

Anil Mahtani, Luis Sanchez,
Enrique Fernandez, Aaron Martinez,
Lentin Joseph

ROS Programming: Building Powerful Robots

Learning Path

Design, build, and simulate complex robots using
the Robot Operating System



Packt

ROS Programming: Building Powerful Robots

Design, build, and simulate complex robots using the Robot Operating System

A learning path in three sections



BIRMINGHAM - MUMBAI

ROS Programming: Building Powerful Robots

Copyright © 2018 Packt Publishing

All rights reserved. No part of this learning path 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 learning path to ensure the accuracy of the information presented. However, the information contained in this learning path is sold without warranty, either express or implied. Neither the author(s), nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this learning path.

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

Content Development Editors: Nithin George Varghese

Graphics: Kirk D'penha

Production Coordinator: Shantanu Zagade

Published on: March 2018

Production reference: 1070318

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-78862-743-6

www.packtpub.com



mapt.io

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

Why subscribe?

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

PacktPub.com

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

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

Table of Contents

[Title page](#)

[Copyright and Credits](#)

[ROS Programming: Building Powerful Robots](#)

[Packt Upsell](#)

[Why subscribe?](#)

[PacktPub.com](#)

[Preface](#)

[Who this learning path is for](#)

[What this learning path covers](#)

[To get the most out of this learning path](#)

[Download the example code files](#)

[Conventions used](#)

[Get in touch](#)

[Reviews](#)

1. Getting Started with ROS

PC installation

Installing ROS Kinetic using repositories

Configuring your Ubuntu repositories

Setting up your source.list file

Setting up your keys

Installing ROS

Initializing rosdep

Setting up the environment

Getting rosinstall

How to install VirtualBox and Ubuntu

Downloading VirtualBox

Creating the virtual machine

Using ROS from a Docker image

Installing Docker

Getting and using ROS Docker images and containers

Installing ROS in BeagleBone Black

Prerequisites

Setting up the local machine and source.list file

Setting up your keys

Installing the ROS packages

Initializing rosdep for ROS

Setting up the environment in the BeagleBone Black

Getting rosinstall for BeagleBone Black

Basic ROS example on the BeagleBone Black

Summary

2. ROS Architecture and Concepts

Understanding the ROS Filesystem level

The workspace

Packages

Metapackages

Messages

Services

Understanding the ROS Computation Graph level

Nodes and nodelets

Topics

Services

Messages

Bags

The ROS master

Parameter Server

Understanding the ROS Community level

Tutorials to practise with ROS

Navigating through the ROS filesystem

Creating our own workspace

Creating an ROS package and metapackage

Building an ROS package

Playing with ROS nodes

Learning how to interact with topics

Learning how to use services

Using Parameter Server

Creating nodes

Building the node

Creating msg and srv files

Using the new srv and msg files

The launch file

Dynamic parameters

Summary

3. Visualization and Debugging Tools

Debugging ROS nodes

Using the GDB debugger with ROS nodes

Attaching a node to GDB while launching ROS

Profiling a node with valgrind while launching ROS

Enabling core dumps for ROS nodes

Logging messages

Outputting logging messages

Setting the debug message level

Configuring the debugging level of a particular node

Giving names to messages

Conditional and filtered messages

Showing messages once, throttling, and other combinations

Using rqt_console and rqt_logger_level to modify the logging level on the fly

Inspecting the system

Inspecting the node's graph online with rqt_graph

Setting dynamic parameters

Dealing with the unexpected

Visualizing nodes diagnostics

Plotting scalar data

Creating a time series plot with rqt_plot

Image visualization

Visualizing a single image

3D visualization

Visualizing data in a 3D world using rqt_rviz

The relationship between topics and frames

Visualizing frame transformations

Saving and playing back data

What is a bag file?

Recording data in a bag file with rosbag

Playing back a bag file

Inspecting all the topics and messages in a bag file

Using the rqt_gui and rqt plugins

Summary

4. The Navigation Stack - Robot Setups

The navigation stack in ROS

Creating transforms

 Creating a broadcaster

 Creating a listener

 Watching the transformation tree

Publishing sensor information

 Creating the laser node

Publishing odometry information

 How Gazebo creates the odometry

 Using Gazebo to create the odometry

 Creating our own odometry

Creating a base controller

 Creating our base controller

Creating a map with ROS

 Saving the map using map_server

 Loading the map using map_server

Summary

5. The Navigation Stack - Beyond Setups

Creating a package

Creating a robot configuration

Configuring the costmaps - global_costmap and local_costmap

Configuring the common parameters

Configuring the global costmap

Configuring the local costmap

Base local planner configuration

Creating a launch file for the navigation stack

Setting up rviz for the navigation stack

The 2D pose estimate

The 2D nav goal

The static map

The particle cloud

The robot's footprint

The local costmap

The global costmap

The global plan

The local plan

The planner plan

The current goal

Adaptive Monte Carlo Localization

Modifying parameters with rqt_reconfigure

Avoiding obstacles

Sending goals

Summary

6. Manipulation with MoveIt!

The MoveIt! architecture

Motion planning

The planning scene

World geometry monitor

Kinematics

Collision checking

Integrating an arm in MoveIt!

What's in the box?

Generating a MoveIt! package with the Setup Assistant

Integration into RViz

Integration into Gazebo or a real robotic arm

Simple motion planning

Planning a single goal

Planning a random target

Planning a predefined group state

Displaying the target motion

Motion planning with collisions

Adding objects to the planning scene

Removing objects from the planning scene

Motion planning with point clouds

The pick and place task

The planning scene

The target object to grasp

The support surface

Perception

Grasping

The pickup action

The place action

The demo mode

Simulation in Gazebo

Summary

7. Using Sensors and Actuators with ROS

Using a joystick or a gamepad

How does joy_node send joystick movements?

Using joystick data to move our robot model

Using Arduino to add sensors and actuators

Creating an example program to use Arduino

Robot platform controlled by ROS and Arduino

Connecting your robot motors to ROS using Arduino

Connecting encoders to your robot

Controlling the wheel velocity

Using a low-cost IMU - 9 degrees of freedom

Installing Razor IMU ROS library

How does Razor send data in ROS?

Creating an ROS node to use data from the 9DoF sensor in our robot

Using robot localization to fuse sensor data in your robot

Using the IMU - Xsens MTi

How does Xsens send data in ROS?

Using a GPS system

How GPS sends messages

Creating an example project to use GPS

Using a laser rangefinder - Hokuyo URG-04lx

Understanding how the laser sends data in ROS

Accessing the laser data and modifying it

Creating a launch file

Using the Kinect sensor to view objects in 3D

How does Kinect send data from the sensors, and how do we see it?

Creating an example to use Kinect

Using servomotors - Dynamixel

How does Dynamixel send and receive commands for the movements?

Creating an example to use the servomotor

Summary

8. Computer Vision

ROS camera drivers support

FireWire IEEE1394 cameras

USB cameras

Making your own USB camera driver with OpenCV

ROS images

Publishing images with ImageTransport

OpenCV in ROS

Installing OpenCV 3.0

Using OpenCV in ROS

Visualizing the camera input images with rqt_image_view

Camera calibration

How to calibrate a camera

Stereo calibration

The ROS image pipeline

Image pipeline for stereo cameras

ROS packages useful for Computer Vision tasks

Visual odometry

Using visual odometry with viso2

Camera pose calibration

Running the viso2 online demo

Performing visual odometry with viso2 with a stereo camera

Performing visual odometry with an RGBD camera

Installing fovi

Using fovi with the Kinect RGBD camera

Computing the homography of two images

Summary

9. Point Clouds

Understanding the PCL

Different point cloud types

Algorithms in PCL

The PCL interface for ROS

My first PCL program

Creating point clouds

Loading and saving point clouds to the disk

Visualizing point clouds

Filtering and downsampling

Registration and matching

Partitioning point clouds

Segmentation

Summary

10. Working with 3D Robot Modeling in ROS

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 file

Visualizing the robot 3D model in RViz

Interacting with pan and tilt joints

Adding physical and collision properties to a URDF model

Understanding robot modeling using xacro

Using properties

Using the math expression

Using macros

Conversion of xacro to URDF

Creating the robot description for a seven DOF robot manipulator

Arm specification

Type of joints

Explaining the xacro model of seven DOF arm

Using constants

Using macros

Including other xacro files

Using meshes in the link

Working with the robot gripper

Viewing the seven DOF arm in RViz

Understanding joint state publisher

Understanding the robot state publisher

Creating a robot model for the differential drive mobile robot

Questions

Summary

11. Simulating Robots Using ROS and Gazebo

Simulating the robotic arm using Gazebo and ROS

The Robotic arm simulation model for Gazebo

Adding colors and textures to the Gazebo robot model

Adding transmission tags to actuate the model

Adding the gazebo_ros_control plugin

Adding a 3D vision sensor to Gazebo

Simulating the robotic arm with Xtion Pro

Visualizing the 3D sensor data

Moving robot joints using ROS controllers in Gazebo

Understanding the ros_control packages

Different types of ROS controllers and hardware interfaces

How the ROS controller interacts with Gazebo

Interfacing joint state controllers and joint position controllers to the arm

Launching the ROS controllers with Gazebo

Moving the robot joints

Simulating a differential wheeled robot in Gazebo

Adding the laser scanner to Gazebo

Moving the mobile robot in Gazebo

Adding joint state publishers in the launch file

Adding the ROS teleop node

Questions

Summary

12. Working with Pluginlib, Nodelets, and Gazebo Plugins

Understanding pluginlib

Creating plugins for the calculator application using pluginlib

Working with pluginlib_calculator package

Step 1 - Creating calculator_base header file

Step 2 - Creating calculator_plugins header file

Step 3 - Exporting plugins using calculator_plugins.cpp

Step 4 - Implementing plugin loader using calculator_loader.cpp

Step 5 - Creating plugin description file: calculator_plugins.xml

Step 6 - Registering plugin with the ROS package system

Step 7 - Editing the CMakeLists.txt file

Step 8: Querying the list of plugins in a package

Step 9 - Running the plugin loader

Understanding ROS nodelets

Creating a nodelet

Step 1 - Creating a package for nodelet

Step 2 - Creating hello_world.cpp nodelet

Step 3 - Explanation of hello_world.cpp

Step 4 - Creating plugin description file

Step 5 - Adding the export tag in package.xml

Step 6 - Editing CMakeLists.txt

Step 7 - Building and running nodelets

Step 8 - Creating launch files for nodelets

Understanding the Gazebo plugins

Creating a basic world plugin

Questions

Summary

13. Writing ROS Controllers and Visualization Plugins

Understanding pr2_mechanism packages

pr2_controller_interface package

Initialization of the controller

Starting the ROS controller

Updating ROS controller

Stopping the controller

pr2_controller_manager

Writing a basic real-time joint controller in ROS

Step 1 ; Creating controller package

Step 2 ; Creating controller header file

Step 3 ; Creating controller source file

Step 4 ; Explanation of the controller source file

Step 5 ; Creating plugin description file

Step 6 ; Updating package.xml

Step 7 ; Updating CMakeLists.txt

Step 8 ; Building controller

Step 9 ; Writing controller configuration file

Step 10 ; Writing launch file for the controller

Step 11 ; Running controller along with PR2 simulation in Gazebo

Understanding ros_control packages

Understanding ROS visualization tool (RViz) and its plugins

Displays panel

RViz toolbar

Views

Time panel

Dockable panels

Writing a RViz plugin for teleoperation

Methodology of building RViz plugin

Step 1 ; Creating RViz plugin package

Step 2 ; Creating RViz plugin header file

Step 3 ; Creating RViz plugin definition

Step 4 ; Creating plugin description file

Step 5 ; Adding export tags in package.xml

Step 6 ; Editing CMakeLists.txt

Step 7 ; Building and loading plugins

Questions

Summary

14. Interfacing I/O Boards, Sensors, and Actuators to ROS

[Understanding the Arduino-ROS interface](#)

[What is the Arduino-ROS interface?](#)

[Understanding the rosserial package in ROS](#)

[Installing rosserial packages on Ubuntu 14.04/15.04](#)

[Understanding ROS node APIs in Arduino](#)

[ROS - Arduino Publisher and Subscriber example](#)

[Arduino-ROS, example - blink LED and push button](#)

[Arduino-ROS, example - Accelerometer ADXL 335](#)

[Arduino-ROS, example - ultrasonic distance sensor](#)

[Equations to find distance using the ultrasonic range sensor](#)

[Arduino-ROS, example - Odometry Publisher](#)

[Interfacing Non-Arduino boards to ROS](#)

[Setting ROS on Odroid-C1 and Raspberry Pi 2](#)

[How to install an OS image to Odroid-C1 and Raspberry Pi 2](#)

[Installation in Windows](#)

[Installation in Linux](#)

[Connecting to Odroid-C1 and Raspberry Pi 2 from a PC](#)

[Configuring an Ethernet hotspot for Odroid-C1 and Raspberry Pi 2](#)

[Installing Wiring Pi on Odroid-C1](#)

[Installing Wiring Pi on Raspberry Pi 2](#)

[Blinking LED using ROS on Odroid-C1 and Raspberry Pi 2](#)

[Push button + blink LED using ROS on Odroid-C1 and Raspberry Pi 2](#)

[Running LED blink in Odroid-C1](#)

[Running button handling and LED blink in Odroid-C1](#)

[Running LED blink in Raspberry Pi 2](#)

[Interfacing Dynamixel actuators to ROS](#)

[Questions](#)

[Summary](#)

15. Programming Vision Sensors using ROS, Open-CV, and PCL

Understanding ROS - OpenCV interfacing packages

Understanding ROS - PCL interfacing packages

Installing ROS perception

Interfacing USB webcams in ROS

Working with ROS camera calibration

Converting images between ROS and OpenCV using cv_bridge

Image processing using ROS and OpenCV

Step 1: Creating ROS package for the experiment

Step 2: Creating source files

Step 3: Explanation of the code

Publishing and subscribing images using image_transport

Converting OpenCV-ROS images using cv_bridge

Finding edges on the image

Visualizing raw and edge detected image

Step 4: Editing the CMakeLists.txt file

Step 5: Building and running example

Interfacing Kinect and Asus Xtion Pro in ROS

Interfacing Intel Real Sense camera with ROS

Working with point cloud to laser scan package

Interfacing Hokuyo Laser in ROS

Interfacing Velodyne LIDAR in ROS

Working with point cloud data

How to publish a point cloud

How to subscribe and process the point cloud

Writing a point cloud data to a PCD file

Read and publish point cloud from a PCD file

Streaming webcam from Odroid using ROS

Questions

Summary

16. Building and Interfacing Differential Drive Mobile Robot Hardware in ROS

Introduction to Chefbot- a DIY mobile robot and its hardware configuration

Flashing Chefbot firmware using Energia IDE

Serial data sending protocol from LaunchPad to PC

Serial data sending protocol from PC to Launchpad

Discussing Chefbot interface packages on ROS

Computing odometry from encoder ticks

Computing motor velocities from ROS twist message

Running robot stand alone launch file using C++ nodes

Configuring the Navigation stack for Chefbot

Configuring the gmapping node

Configuring the Navigation stack packages

Common configuration (local_costmap) and (global_costmap)

Configuring global costmap parameters

Configuring local costmap parameters

Configuring base local planner parameters

Configuring DWA local planner parameters

Configuring move_base node parameters

Understanding AMCL

Understanding RViz for working with the Navigation stack

2D Pose Estimate button

Visualizing the particle cloud

The 2D Nav Goal button

Displaying the static map

Displaying the robot footprint

Displaying the global and local cost map

Displaying the global plan, local plan, and planner plan

The current goal

Obstacle avoidance using the Navigation stack

Working with Chefbot simulation

Building a room in Gazebo

Adding model files to the Gazebo model folder

Sending a goal to the Navigation stack from a ROS node

Questions

Summary

17. Exploring the Advanced Capabilities of ROS-MoveIt!

Motion planning using the move_group C++ interface

Motion planning a random path using MoveIt! C++ APIs

Motion planning a custom path using MoveIt! C++ APIs

Collision checking in robot arm using MoveIt!

Adding a collision object in MoveIt!

Removing a collision object from the planning scene

Checking self collision using MoveIt! APIs

Working with perception using MoveIt! and Gazebo

Grasping using MoveIt!

Working with robot pick and place task using MoveIt!

Creating Grasp Table and Grasp Object in MoveIt!

Pick and place action in Gazebo and real Robot

Understanding Dynamixel ROS Servo controllers for robot hardware interfacing

The Dynamixel Servos

Dynamixel-ROS interface

Interfacing seven DOF Dynamixel based robotic arm to ROS MoveIt!

Creating a controller package for COOL arm robot

MoveIt! configuration of the COOL Arm

Questions

Summary

18. ROS for Industrial Robots

Understanding ROS-Industrial packages

Goals of ROS-Industrial

ROS-Industrial - a brief history

Benefits of ROS-Industrial

Installing ROS-Industrial packages

Block diagram of ROS-Industrial packages

Creating URDF for an industrial robot

Creating MoveIt! configuration for an industrial robot

Updating the MoveIt! configuration files

Testing the MoveIt! configuration

Installing ROS-Industrial packages of universal robotic arm

Installing the ROS interface of universal robots

Understanding the MoveIt! configuration of a universal robotic arm

Working with MoveIt! configuration of ABB robots

Understanding the ROS-Industrial robot support packages

Visualizing the ABB robot model in RViz

ROS-Industrial robot client package

Designing industrial robot client nodes

ROS-Industrial robot driver package

Understanding MoveIt! IKFast plugin

Creating the MoveIt! IKFast plugin for the ABB-IRB6640 robot

Prerequisites for developing the MoveIt! IKFast plugin

OpenRave and IK Fast Module

MoveIt! IK Fast

Installing MoveIt! IKFast package

Installing OpenRave on Ubuntu 14.04.3

Creating the COLLADA file of a robot to work with OpenRave

Generating the IKFast CPP file for the IRB 6640 robot

Creating the MoveIt! IKFast plugin

Questions

Summary

19. Troubleshooting and Best Practices in ROS

[Setting up Eclipse IDE on Ubuntu 14.04.3](#)

[Setting ROS development environment in Eclipse IDE](#)

[Global settings in Eclipse IDE](#)

[ROS compile script for Eclipse IDE](#)

[Adding ROS Catkin package to Eclipse](#)

[Adding run configurations to run ROS nodes in Eclipse](#)

[Best practices in ROS](#)

[ROS C++ coding style guide](#)

[Standard naming conventions used in ROS](#)

[Code license agreement](#)

[ROS code formatting](#)

[ROS code documentation](#)

[Console output](#)

[Best practices in the ROS package](#)

[Important troubleshooting tips in ROS](#)

[Usage of rosrun](#)

[Questions](#)

[Summary](#)

3. ROS Robotics Projects

20. Face Detection and Tracking Using ROS, OpenCV and Dynamixel Servos

Overview of the project

Hardware and software prerequisites

Installing dependent ROS packages

Installing the usb_cam ROS package

Creating a ROS workspace for dependencies

Interfacing Dynamixel with ROS

Installing the ROS dynamixel_motor packages

Creating face tracker ROS packages

The interface between ROS and OpenCV

Working with the face-tracking ROS package

Understanding the face tracker code

Understanding CMakeLists.txt

The track.yaml file

The launch files

Running the face tracker node

The face_tracker_control package

The start_dynamixel launch file

The pan controller launch file

The pan controller configuration file

The servo parameters configuration file

The face tracker controller node

Creating CMakeLists.txt

Testing the face tracker control package

Bringing all the nodes together

Fixing the bracket and setting up the circuit

The final run

Questions

Summary

21. Building a Siri-Like Chatbot in ROS

Social robots

Building social robots

Prerequisites

Getting started with AIML

AIML tags

The PyAIML interpreter

Installing PyAIML on Ubuntu 16.04 LTS

Playing with PyAIML

Loading multiple AIML files

Creating an AIML bot in ROS

The AIML ROS package

Installing the ROS sound_play package

Installing the dependencies of sound_play

Installing the sound_play ROS package

Creating the ros_aiml package

The aiml_server node

The AIML client node

The aiml_tts client node

The AIML speech recognition node

start_chat.launch

start_tts_chat.launch

start_speech_chat.launch

Questions

Summary

22. Controlling Embedded Boards Using ROS

Getting started with popular embedded boards

An introduction to Arduino boards

How to choose an Arduino board for your robot

Getting started with STM32 and TI Launchpads

The Tiva C Launchpad

Introducing the Raspberry Pi

How to choose a Raspberry Pi board for your robot

The Odroid board

Interfacing Arduino with ROS

Monitoring light using Arduino and ROS

Running ROS serial server on PC

Interfacing STM32 boards to ROS using mbed

Interfacing Tiva C Launchpad boards with ROS using Energia

Running ROS on Raspberry Pi and Odroid boards

Connecting Raspberry Pi and Odroid to PC

Controlling GPIO pins from ROS

Creating a ROS package for the blink demo

Running the LED blink demo on Raspberry Pi and Odroid

[Questions](#)

[Summary](#)

23. Teleoperate a Robot Using Hand Gestures

- Teleoperating ROS Turtle using a keyboard
- Teleoperating using hand gestures
- Setting up the project
- Interfacing the MPU-9250 with the Arduino and ROS
 - The Arduino-IMU interfacing code
- Visualizing IMU TF in Rviz
- Converting IMU data into twist messages
- Integration and final run
- Teleoperating using an Android phone
- Questions
- Summary

24. Object Detection and Recognition

Getting started with object detection and recognition

The `find_object_2d` package in ROS

 Installing `find_object_2d`

 Installing from source code

 Running `find_object_2d` nodes using webcams

 Running `find_object_2d` nodes using depth sensors

Getting started with 3D object recognition

Introduction to 3D object recognition packages in ROS

 Installing ORK packages in ROS

 Detecting and recognizing objects from 3D meshes

 Training using 3D models of an object

 Training from captured 3D models

Recognizing objects

Questions

Summary

25. Deep Learning Using ROS and TensorFlow

Introduction to deep learning and its applications

Deep learning for robotics

Deep learning libraries

Getting started with TensorFlow

Installing TensorFlow on Ubuntu 16.04 LTS

TensorFlow concepts

Graph

Session

Variables

Fetches

Feeds

Writing our first code in TensorFlow

Image recognition using ROS and TensorFlow

Prerequisites

The ROS image recognition node

Running the ROS image recognition node

Introducing to scikit-learn

Installing scikit-learn on Ubuntu 16.04 LTS

Introducing to SVM and its application in robotics

Implementing an SVM-ROS application

Questions

Summary

26. ROS on MATLAB and Android

Getting started with the ROS-MATLAB interface

Setting Robotics Toolbox in MATLAB

Basic ROS functions in MATLAB

 Initializing a ROS network

 Listing ROS nodes, topics, and messages

Communicating from MATLAB to a ROS network

Controlling a ROS robot from MATLAB

 Designing the MATLAB GUI application

 Explaining callbacks

 Running the application

Getting started with Android and its ROS interface

 Installing rosjava

 Installing from the Ubuntu package manager

 Installing from source code

 Installing android-sdk from the Ubuntu package manager

 Installing android-sdk from prebuilt binaries

 Installing the ROS-Android interface

 Playing with ROS-Android applications

 Troubleshooting

 Android-ROS publisher-subscriber application

 The teleop application

 The ROS Android camera application

 Making the Android device the ROS master

 Code walkthrough

 Creating basic applications using the ROS-Android interface

 Troubleshooting tips

Questions

Summary

27. Building an Autonomous Mobile Robot

Robot specification and design overview

Designing and selecting the motors and wheels for the robot

Computing motor torque

Calculation of motor RPM

Design summary

Building 2D and 3D models of the robot body

The base plate

The pole and tube design

The motor, wheel, and motor clamp design

The caster wheel design

Middle plate and top plate design

The top plate

3D modeling of the robot

Simulating the robot model in Gazebo

Mathematical model of a differential drive robot

Simulating Chefbot

Building the URDF model of Chefbot

Inserting 3D CAD parts into URDF as links

Inserting Gazebo controllers into URDF

Running the simulation

Mapping and localization

Designing and building actual robot hardware

Motor and motor driver

Motor encoders

Tiva C Launchpad

Ultrasonic sensor

OpenNI depth sensor

Intel NUC

Interfacing sensors and motors with the Launchpad

Programming the Tiva C Launchpad

Interfacing robot hardware with ROS

Running Chefbot ROS driver nodes

Gmapping and localization in Chefbot

Questions

Summary

28. Creating a Self-Driving Car Using ROS

Getting started with self-driving cars

History of autonomous vehicles

Levels of autonomy

Functional block diagram of a typical self-driving car

GPS, IMU, and wheel encoders

Xsens MTi IMU

Camera

Ultrasonic sensors

LIDAR and RADAR

Velodyne HDL-64 LIDAR

SICK LMS 5xx/1xx and Hokuyo LIDAR

Continental ARS 300 radar (ARS)

Delphi radar

On-board computer

Software block diagram of self-driving cars

Simulating the Velodyne LIDAR

Interfacing Velodyne sensors with ROS

Simulating a laser scanner

Explaining the simulation code

Interfacing laser scanners with ROS

Simulating stereo and mono cameras in Gazebo

Interfacing cameras with ROS

Simulating GPS in Gazebo

Interfacing GPS with ROS

Simulating IMU on Gazebo

Interfacing IMUs with ROS

Simulating an ultrasonic sensor in Gazebo

Low-cost LIDAR sensors

Sweep LIDAR

RPLIDAR

Simulating a self-driving car with sensors in Gazebo

Installing prerequisites

Visualizing robotic car sensor data

Moving a self-driving car in Gazebo

Running hector SLAM using a robotic car

Interfacing a DBW car with ROS

Installing packages

Visualizing the self-driving car and sensor data

Communicating with DBW from ROS

Introducing the Udacity open source self-driving car project

MATLAB ADAS toolbox

Questions

Summary

29. Teleoperating a Robot Using a VR Headset and Leap Motion

Getting started with a VR headset and Leap Motion

Project prerequisites

Design and working of the project

Installing the Leap Motion SDK on Ubuntu 14.04.5

Visualizing Leap Motion controller data

Playing with the Leap Motion visualizer tool

Installing the ROS driver for the Leap Motion controller

Testing the Leap Motion ROS driver

Visualizing Leap Motion data in Rviz

Creating a teleoperation node using the Leap Motion controller

Building a ROS-VR Android application

Working with the ROS-VR application and interfacing with Gazebo

Working with TurtleBot simulation in VR

Troubleshooting the ROS-VR application

Integrating ROS-VR application and Leap Motion teleoperation

Questions

Summary

30. Controlling Your Robots over the Web

Getting started with ROS web packages

rosbridge_suite

roslibjs, ros2djs, and ros3djs

The tf2_web_republisher package

Setting up ROS web packages on ROS Kinetic

Installing rosbridge_suite

Setting up rosbridge client libraries

Installing tf2_web_republisher on ROS Kinetic

Teleoperating and visualizing a robot on a web browser

Working of the project

Connecting to rosbridge_server

Initializing the teleop

Creating a 3D viewer inside a web browser

Creating a TF client

Creating a URDF client

Creating text input

Running the web teleop application

Controlling robot joints from a web browser

Installing joint_state_publisher_js

Including the joint state publisher module

Creating the joint state publisher object

Creating an HTML division for sliders

Running the web-based joint state publisher

Prerequisites

Installing prerequisites

Explaining the code

Running the robot surveillance application

Web-based speech-controlled robot

Prerequisites

Enabling speech recognition in the web application

Running a speech-controlled robot application

Questions

Summary

Bibliography

Other Books You May Enjoy

Leave a review - let other readers know what you think

Preface

Robot Operating System is one of the most widely used software frameworks for robotic research and for companies to model, simulate, and prototype robots. Applying your knowledge of ROS to actual robotics is much more difficult than people realize, but this learning path will give you what you need to create your own robotics in no time!

This learning path is packed with hands-on examples that will help you program your robot and give you complete solutions using open source ROS libraries and tools. It also shows you how to use virtual machines and Docker containers to simplify the installation of Ubuntu and the ROS framework, so you can start working in an isolated and control environment without changing your regular computer setup. This also will be the perfect companion for a robotics enthusiast who really wants to do something big in the field.

By the end of the learning path, you'll be able to leverage all the ROS Kinetic features to build a fully fledged robot for all your needs and Finally, you will get to know the best practices to follow when programming using ROS.

Who this learning path is for

This course targets robotic enthusiasts, developers, and researchers who would like to build robot applications using ROS. If you are looking to explore the advanced ROS features in your projects, then this learning path is for you. Basic knowledge of ROS, GNU/Linux, and programming concepts is assumed.

What this learning path covers

Section 1, Effective Robotics Programming with ROS, Third Edition gives you a comprehensive review of ROS, the Robot Operating System framework, which is used nowadays by hundreds of research groups and companies in the robotics industry. More importantly, ROS is also the painless entry point to robotics for nonprofessionals and students. This section will guide you through the installation process of ROS, and soon enough, you will be playing with the basic tools and understanding the different elements of the framework.

The content of the section can be followed without any special devices, and each chapter comes with a series of source code examples and tutorials that you can run on your own computer. This is the only thing you need to follow the section. However, we also show you how to work with hardware so that you can connect your algorithms with the real world. Special care has been taken in choosing devices that are affordable for amateur users, but at the same time, the most typical sensors or actuators in robotics research are covered.

Finally, the potential of ROS is illustrated with the ability to work with whole robots in a real or simulated environment. You will learn how to create your own robot and integrate it with a simulation by using the Gazebo simulator. From here, you will have the chance to explore the different aspects of creating a robot, such as perceiving the world using computer vision or point cloud analysis, navigating through the environment using the powerful navigation stack, and even being able to control robotic arms to interact with your surroundings using the MoveIt! package. By the end of the section, it is our hope that you will have a thorough understanding of the endless possibilities that ROS gives you when developing robotic systems.

Section 2, 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 section 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 section 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 section 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.

Section 3, ROS Robotics Projects is a practical guide to learning ROS by making interesting projects using it. The section assumes that you have some knowledge of ROS.

However, if you do not have any experience with ROS, you can still learn from this section. The first chapter is dedicated to absolute beginners. ROS is widely used in robotics companies, universities, and robot research labs for designing and programming robots. If you would like to work in the robotics software domain or if you want to have a career as a robotics software engineer, this section is perfect for you.

The basic aim of this section is to teach ROS through interactive projects. The projects that we are discussing here can also be reused in your academic or industrial projects. This section handles a wide variety of new technology that can be interfaced with ROS. For example, you will see how to build a self-driving car prototype, how to build a deep-learning application using ROS, and how to build a VR application in ROS. These are only a few highlighted topics; in addition, you will find some 15 projects and applications using ROS and its libraries.

You can work with any project after meeting its prerequisites. Most of the projects can be completed without many dependencies. We are using popular and available hardware components to build most of the projects. So this will help us create almost all of these projects without much difficulty.

To get the most out of this learning path

Section 1, Effective Robotics Programming with ROS, Third Edition was written with the intention that almost everybody can follow it and run the source code examples provided with it. Basically, you need a computer with a Linux distribution. Although any Linux distribution should be fine, it is recommended that you use a version of Ubuntu 16.04 LTS. Then, you will use ROS Kinetic, which is installed according to the instructions given in *Chapter 1, Getting Started with ROS*.

As regards the hardware requirements of your computer, in general, any computer or laptop is enough. However, it is advisable to use a dedicated graphics card in order to run the Gazebo simulator. Also, it will be good to have a good number of peripherals so that you can connect several sensors and actuators, including cameras and Arduino boards.

You will also need Git (the git-core Debian package) in order to clone the repository with the source code provided with this section. Similarly, you are expected to have a basic knowledge of the Bash command line, GNU/Linux tools, and some C/C++ programming skills.

For *Section 2 Mastering ROS for Robotics Programming* and *Section 3 ROS Robotics Projects*, You should have a good PC running Linux distribution, preferably Ubuntu 14.04.3 or Ubuntu 15.04 and Ubuntu 16.04 LTS respectively.

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 section 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.

Download the example code files

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

You can download the code files by following these steps:

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

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

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

The code bundle for the learning path is also hosted on GitHub at <https://github.com/PacktPublishing/ROS-Programming-Building-Powerful-Robots>. We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Conventions used

There are a number of text conventions used throughout this learning path.

CodeInText: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "The `*-ros-pkg` is a community repository for developing high-level libraries easily."

A block of code is set as follows:

```
...  
source /opt/ros/indigo/setup.bash  
source /opt/ros/jade/setup.bash  
source /opt/ros/kinetic/setup.bash  
...
```

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

```
<launch>  
  <node pkg="hokuyo_node" type="hokuyo_node" name="hokuyo_node"/>  
  <node pkg="rviz" type="rviz" name="rviz"  
    args="-d $(find chapter8_tutorials)/config/laser.rviz"/>  
  
  <node pkg="chapter8_tutorials" type="c8_laserscan"  
    name="c8_laserscan" />  
</launch>
```

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

```
$ rosrun openni_camera openni_node  
$ roslaunch openni_launch openni.launch
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Two windows must appear, one with a 3D figure that represents the pose of the IMU and a 2D figure with the Roll, Pitch, and Yaw."



Warnings or important notes appear like this.



Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

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

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

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

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

Reviews

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

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

Effective Robotics Programming with ROS, Third Edition

Find out everything you need to know to build powerful robots with the most up-to-date ROS

Getting Started with ROS

Welcome to the first chapter of this section, where you will learn how to install ROS, the new standard software framework in robotics. This section is an update on *Learning ROS for Robotics Programming - Second Edition*, based in ROS Hydro/Indigo. With ROS, you will learn how to program and control your robots the easy way, using tons of examples and source code that will show you how to use sensors and devices, or how to add new functionalities, such as autonomous navigation, visual perception, and others, to your robot. Thanks to the open source ethos and a community that is developing state-of-the-art algorithms and providing new functionalities, ROS is growing every day.

This section will cover the following topics:

- Installing ROS Kinetic framework on a compatible version of Ubuntu
- The basic operation of ROS
- Debugging and visualizing data
- Programming your robot using this framework
- Connecting sensors, actuators, and devices to create your robot
- Using the navigation stack to make your robot autonomous

In this chapter, we are going to install a full version of ROS Kinetic in Ubuntu. ROS is fully supported and recommended for Ubuntu, and it is experimental for other operative systems. The version used in this section is the 15.10 (Wily Werewolf), and you can download it for free from <http://releases.ubuntu.com/15.10>. Note that you can also use Ubuntu 16.04 (Xenial), following the same steps shown here; indeed, for the BeagleBone Black installation we will use Ubuntu Xenial.

Before starting with the installation, we are going to learn about the origin of ROS and its history.

The **Robot Operating System (ROS)** is a framework that, nowadays, is widely accepted and used in the robotics community. Its main goal is to make the multiple components of a robotics system easy to develop and share so they can work on other robots with minimal changes. This basically allows for code reuse, and improves the quality of the code by having it tested by a large number of users and platforms. ROS was originally developed in 2007 by the **Stanford Artificial Intelligence Laboratory (SAIL)** in support of the Stanford AI Robot project. Since 2008, Willow Garage continued the development, and recently **Open Source Robotics Foundation (OSRF)** began to oversee the maintenance of ROS and partner projects, like Gazebo, including the development of new features.

A lot of research institutions have started to develop in ROS, adding hardware and sharing their code. Also, companies have started to adapt their products to be used in ROS. In the following set of images, you can see some of the platforms that are fully supported. Normally, these platforms are published with a lot of code, examples, and simulators to permit the developers to start work easily. The first three humanoid robots are examples of robots with published code. The last one is an AUV developed by the *University of Las Palmas de Gran Canaria*, and the code has not been published yet. You can find many other examples at <http://wiki.ros.org/Robots>.



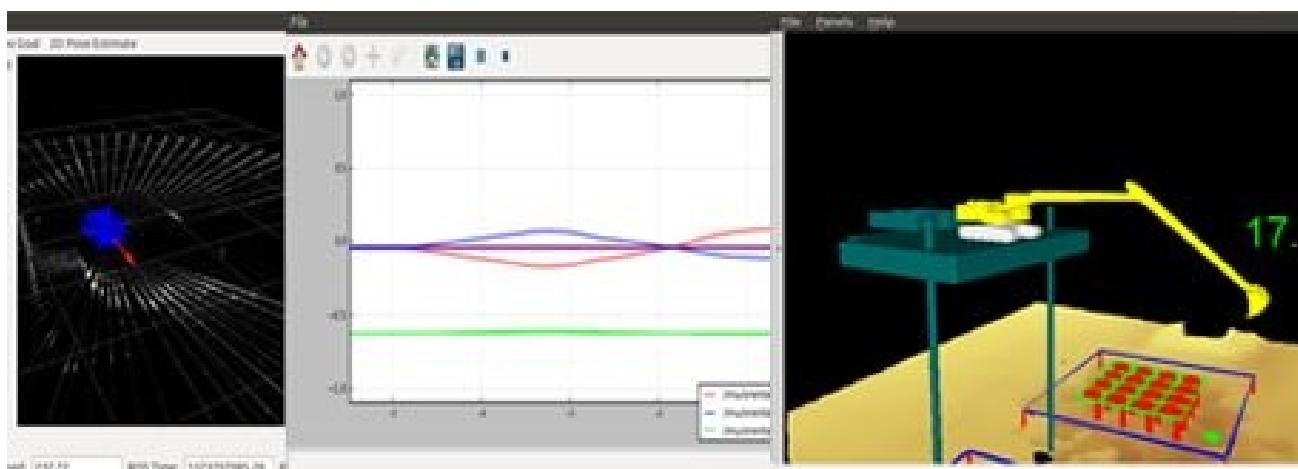
Most of the sensors and actuators used in robotics are supported by ROS as drivers.

Furthermore, some companies benefit from ROS and open hardware to create cheaper and easier to use sensors, as existing software can be used for them at zero cost. The Arduino board is a good example because you can add many different kinds of sensors to this cheap electronic board, such as encoders, light and temperature sensors, and many others, and then expose their measurements to ROS to develop robotic applications.

ROS provides a hardware abstraction, low-level device control with ROS control, implementations of commonly used functionalities and libraries, message passing between processes, and package management with `catkin` and `cmake`.

It uses graph architecture with a centralized topology, where processing takes place in nodes that may receive and send messages to communicate with other nodes on the graph net. A node is any process that can read data from a sensor, control an actuator, or run high level, complex robotic or vision algorithms for mapping or navigating autonomously in the environment.

The `*-ros-pkg` is a community repository for developing high-level libraries easily. Many of the capabilities frequently associated with ROS, such as the navigation library and the `rviz` visualizer, are developed in this repository. These libraries provide a powerful set of tools for working with ROS easily; visualization, simulators, and debugging tools are among the most important features that they have to offer. In the following image you can see two of these tools, the `rviz` and `rqt_plot`. The screenshot in the center is `rqt_plot`, where you can see the plotted data from some sensors. The other two screenshots are `rviz`; in the screenshot you can see a 3D representation of a real robot.



ROS is released under the terms of the **Berkeley Software Distribution (BSD)** license and is an open source software. It is free for commercial and research use. The `ros-pkg` contributed packages are licensed

under a variety of open source licenses.

With ROS, you can take a code from the repositories, improve it, and share it again. This philosophy is the underlying principle of open source software.

ROS has numerous versions, the last one being Indigo. In this section, we are going to use Kinetic because it is the latest version. Now we are going to show you how to install ROS Kinetic. As we mentioned before, the operating system used in the section is Ubuntu, and we are going to use it throughout this section and with all the tutorials. If you use another operating system and you want to follow the section, the best option is to install a virtual machine with a copy of Ubuntu. At the end of this chapter, we will explain how to install a virtual machine to use the ROS inside it, or download a virtual machine with ROS installed.

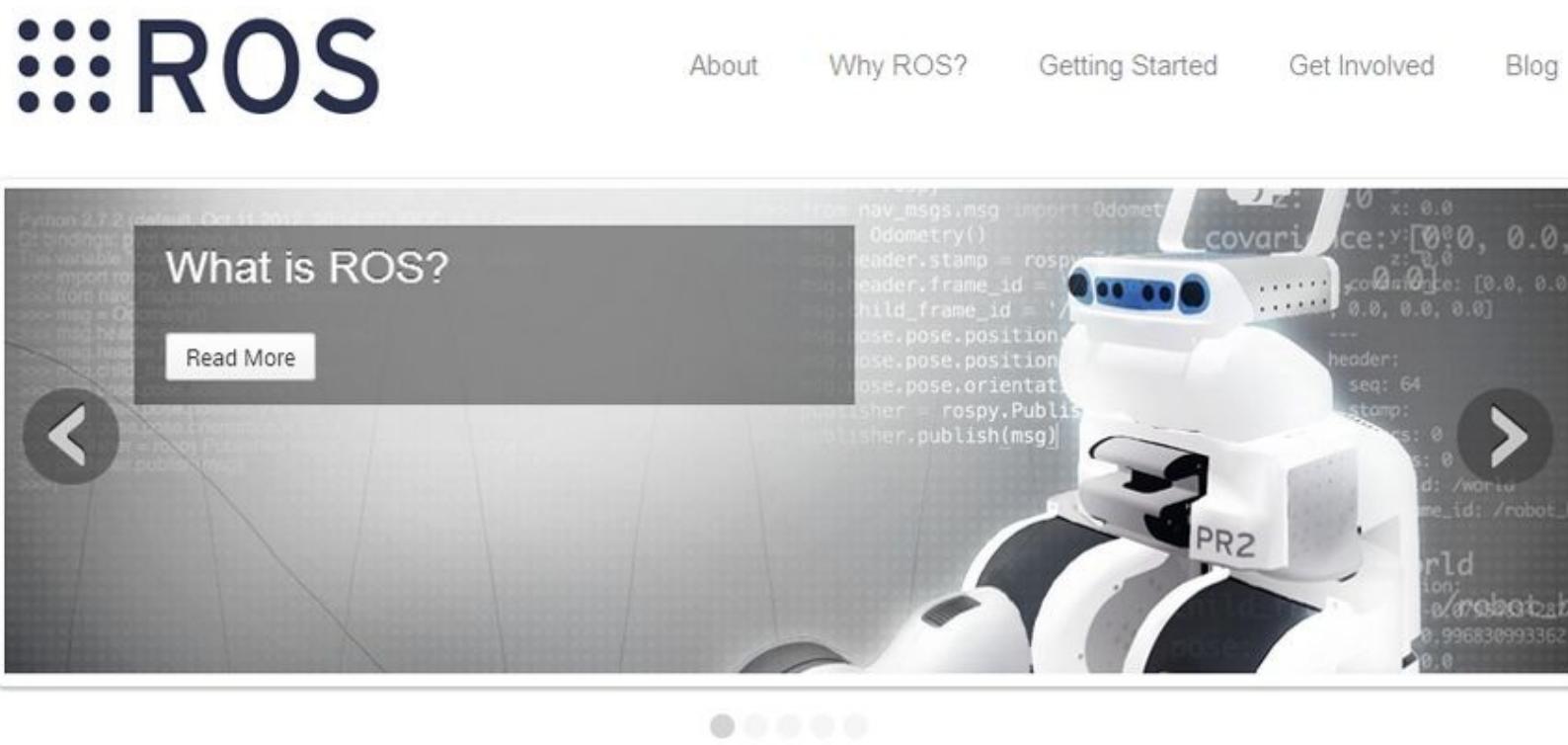
If you want to try installing it on an operating system other than Ubuntu, you can find instructions on how to do so with many other operating systems at <http://wiki.ros.org/kinetic/Installation>.

PC installation

We assume that you have a PC with a copy of Ubuntu 15.10. It will also be necessary to have a basic knowledge of Linux and command tools such as the terminal, Vim, folder creation, and so on. If you need to learn these tools, you can find a lot of relevant resources on the Internet, or you can find books on these topics instead.

Installing ROS Kinetic using repositories

Last year, the ROS web page was updated with a new design and a new organization of contents. The following is a screenshot of the web page:



In the menu, you can find information about ROS and whether ROS is a good choice for your system. You can also find blogs, news, and other features.

Instructions for ROS installation can be found under the Install tab in the Getting Started section.

ROS recommends that you install the system using the repository instead of the source code, unless you are an advanced user and you want to make a customized installation; in that case, you may prefer installing ROS using the source code.

To install ROS using the repositories, we will start by configuring the Ubuntu repository in our system.

Configuring your Ubuntu repositories

In this section, you will learn the steps for installing ROS Kinetic in your computer. This process has been based on the official installation page, which can be found at <http://wiki.ros.org/kinetic/Installation/Ubuntu>.

We assume that you know what an Ubuntu repository is and how to manage it. If you have any doubts about it, refer to <https://help.ubuntu.com/community.Repositories/Ubuntu>.

Before we start the installation, we need to configure our repositories. To do that, the repositories need to allow restricted, universe, and multiverse. To check if your Ubuntu accepts these repositories, click on Ubuntu Software Center in the menu on the left-hand side of your desktop, as shown in the following screenshot:



Click on Edit | Software Sources and you will see the following window. Make sure that all the listed options are checked as shown in the following screenshot (choose the appropriate country for the server from which you download the sources):



Normally these options are marked, so you should not have any problem with this step.

Setting up your source.list file

In this step, you have to select your Ubuntu version. It is possible to install ROS Kinetic in various versions of the operating system. You can use any of them, but we recommend version 15.10 to follow the chapters of this section. Keep in mind that Kinetic works in the Wily Werewolf (15.10) and Xenial Xerus (16.04) versions of Ubuntu. Type the following command to add the repositories:

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

Downloading the example code:

Detailed steps to download the code bundle are mentioned in the Preface of this section. Please have a look.



The code bundle for the section is also hosted on GitHub at https://github.com/rosbook/effective_robatics_programming_with_ros. We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Once you've added the correct repository, your operating system will know where to download programs to install them into your system.

Setting up your keys

This step is to confirm that the origin of the code is correct and that no one has modified the code or programs without the knowledge of the owner. Normally, when you add a new repository, you have to add the keys of that repository, so it's added to your system's trusted list.

```
| $ wget http://packages.ros.org/ros.key -O - | sudo apt-key add -
```

Now we can be sure that the code came from an authorized site and has not been modified.

Installing ROS

We are ready to start the installation now, but before we do that, we'd better make an update to avoid problems with the libraries and versions of software that are not ideal for ROS. This is done with the following command:

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

ROS is huge; sometimes you will install libraries and programs that you will never use. Normally it has four different installations, but this depends on the final use. For

example, if you are an advanced user, you might only need the basic installation for a robot without much space on the hard disk. For this section, we recommend you use the full installation because it will install everything necessary to practice the examples and tutorials.

It doesn't matter if you don't know what are you installing right now — rviz, simulators, navigation, and so on. You will learn everything in the upcoming chapters:

- The easiest (and recommended if you have enough hard disk space) installation is known as `desktop-full`. It comes with ROS, the rqt tools, the rviz visualizer (for 3D), many generic robot libraries, the simulator in 2D (like a stage plan) and 3D (usually Gazebo), the navigation stack (to move, localize, do mapping, and control arms), and also perception libraries using vision, lasers, or RGBD cameras:

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

- If you do not have enough disk space, or if you prefer to install only a few packages, install only the desktop install initially, which only comes with ROS, the rqt tools, rviz, and generic robot libraries. You can install the rest of the packages as and when you need them, for example, by using `aptitude` and looking for `ros-kinetic-*` packages with the following command:

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

- If you only want the bare bones, install `ROS-base`, which is usually recommended for the robot itself, or for computers without a screen or just a **TTY**. It will install the ROS package with the build and communication libraries and no GUI tools at all. With **BeagleBone Black (BBB)**, we will install the system with the following option:

```
| $ sudo apt-get install ros-kinetic-ros-base
```

- Finally, whichever of the previous options you choose, you can also install individual/specific ROS packages (for a given package name):

```
| $ sudo apt-get install ros-kinetic-PACKAGE
```

Initializing rosdep

Before using ROS we need to initialize `rosdep`. The `rosdep` command-line tool helps with the installation of system dependencies for the source code that we are going to compile or install. For this reason, it is required by some of the core components in ROS, so it is installed by default with it. To initialize `rosdep`, you have to run the following commands:

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

Setting up the environment

Congratulations! If you are at this step, you have an installed version of ROS on your system! To start using it, the system needs to know the location of the executable or binary files, as well as the other commands. To do this, normally you need to execute the next script; if you also install another ROS distro, you can work with both just by calling the script of the one you need each time, since this script simply sets your environment. Here, we use the one for ROS Kinetic, but just replace `kinetic` with `indigo` or `jade`, for example, if you want to try other distros:

```
| $ source /opt/ros/kinetic/setup.bash
```

If you type `roscore` in the shell, you will see something starting up. This is the best test for finding out if you have ROS, and if it is installed correctly.

Note that if you open another terminal you also have to source the `setup.bash` file to set the environment variables to detect the ROS packages installed on your system. Otherwise, `roscore` or other ROS commands will not work. This is because the script must be sourced again to configure the environment variables, which include the path where ROS is installed, as well as other packages and additional paths for compiling new code properly.

It is very easy to solve this; you just need to add the script at the end of your `.bashrc` script file so that when you start a new shell, the script will execute and you will have the environment configured.

The `.bashrc` file is within the user home (`/home/USERNAME/.bashrc`). It has the configuration of the shell or terminal, and each time the user opens the terminal, this file is loaded. That way, you can add commands or configuration to make the user's life easy. For this reason, we will add the script at the end of the `.bashrc` file to avoid keying it in each time we open a terminal. We do this with the following command:

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

To see the results, you have to execute the file using the following command, or close the current terminal and open another:

```
| $ source ~/.bashrc
```

Some users need more than a single ROS distribution installed in their system, so you'll have several distros living in the same system and may need to switch between them. Your `~/.bashrc` file must only source the `setup.bash` file of the version you are currently using, since the last call will override the environment set by the others.

For example, you might have the following lines in your `.bashrc` file:

```
...  
source /opt/ros/indigo/setup.bash  
source /opt/ros/jade/setup.bash  
source /opt/ros/kinetic/setup.bash  
...
```

The ROS Kinetic version will be executed in this case. Make sure that the version you are running is the last one in the file. It's also recommended to source a single `setup.bash`.

If you want to check the version used in a terminal, you can do so easily running the `echo $ROS_DISTRO` command.

Getting rosinstall

Now the next step is to install a command tool that will help us install other packages with a single command. This tool is based in Python, but don't worry, you don't need to know Python to use it. You will learn how to use this tool in the upcoming chapters:

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

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

And that's it! You have a complete ROS system installed in your system. When I finish a new installation of ROS, I personally like to test two things: that `roscore` and `turtlesim` both work.

If you want to do the same, type the following commands in different shells:

```
| $ roscore  
| $ rosrun turtlesim turtlesim_node
```

If everything is okay, you will see the following screenshot:



How to install VirtualBox and Ubuntu

VirtualBox is a general-purpose, full virtualizer for x86 hardware, targeted at server, desktop, and embedded use. VirtualBox is free and supports all the major operating systems and pretty much every Linux flavor out there.

If you don't want to change the operating system of your computer to Ubuntu, tools such as VirtualBox help us virtualize a new operating system in our computers without making any changes.

In the following section, we are going to show you how to install VirtualBox and a new installation of Ubuntu. After this virtual installation, you should have a clean installation for restarting your development machine if you have any problems, or to save all the setups necessary for your robot in the machine.

Downloading VirtualBox

The first step is to download the VirtualBox installation file. The latest version at the time of writing this section is 4.3.12; you can download the Linux version of it from <http://download.virtualbox.org/virtualbox/4.3.12/>. If you're using Windows, you can download it from <http://download.virtualbox.org/virtualbox/4.3.12/VirtualBox-4.3.12-93733-Win.exe>.

Once installed, you need to download the image of Ubuntu; for this tutorial we will use a copy of Ubuntu 15.10 from OSBOXES found at <http://www.osboxes.org/ubuntu/>; then we will simply install ROS Kinetic following the same instructions described in the previous section. In particular, the Ubuntu 15.10 image can be downloaded from http://sourceforge.net/projects/osboxes/files/vms/vbox/Ubuntu/15.10/Ubuntu_15.10-64bit.7z/download.

This would download a .7z file. In Linux, it can be uncompressed with the following:

```
| $ 7z x Ubuntu_15.10-64bit.7z
```

If the .7z command is not installed, it can be installed with the following:

```
| $ sudo apt-get install p7zip-full
```

The virtual machine file will go into the 64-bit folder with the name: Ubuntu 15.10 Wily (64bit).vdi

Creating the virtual machine

Creating a new virtual machine with the downloaded file is very easy; just proceed with the following steps. Open VirtualBox and click on New. We are going to create a new virtual machine that will use the `Ubuntu 15.10 wily (64bit).vdi` file downloaded before, which is a hard disk image with Ubuntu 15.10 already installed.

Set the name, type, and version of the virtual machine as shown in the following screenshot:

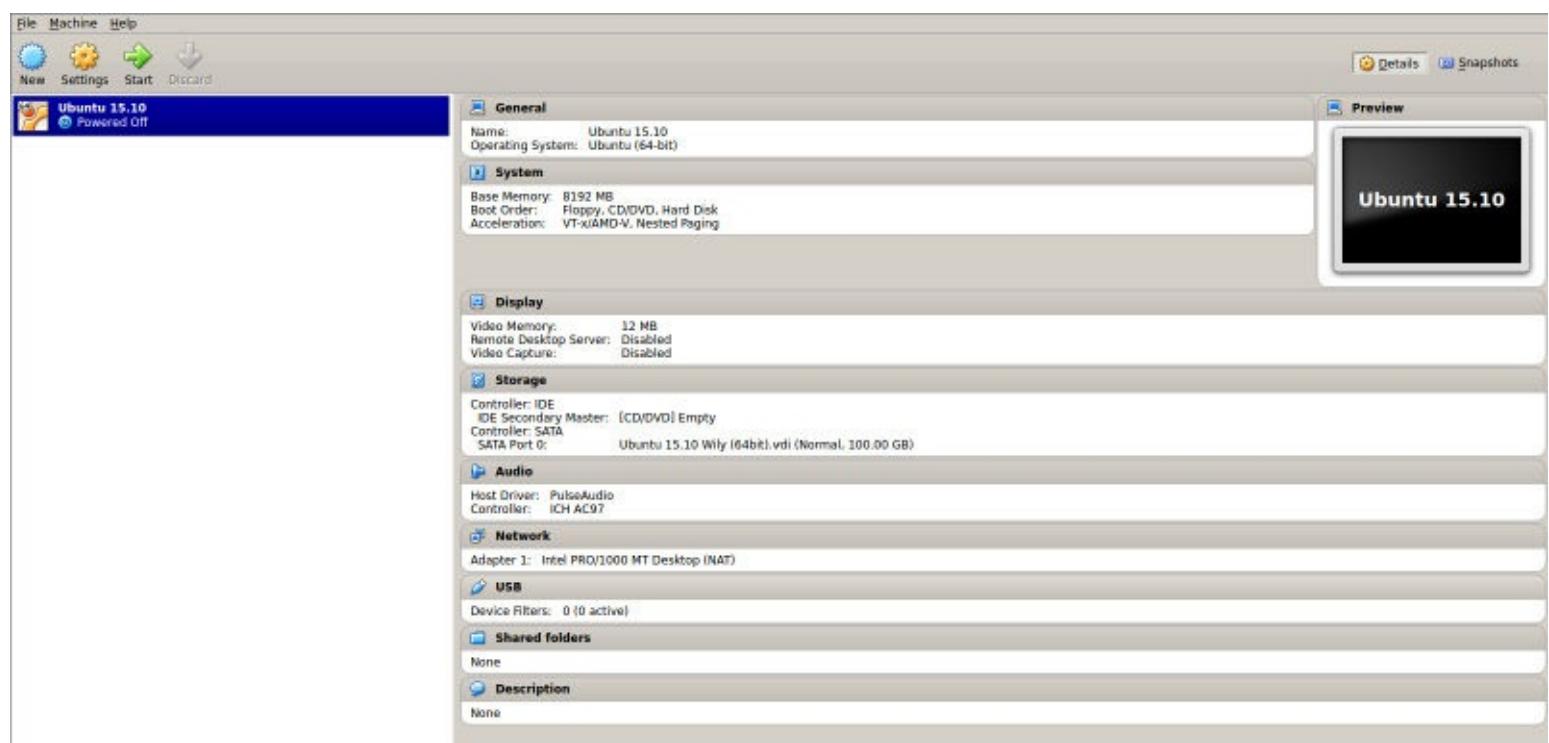


You can configure the parameters of the new virtual machine in the windows that follow. Keep the default configuration and change only the name for the virtual system. This name is how you distinguish this virtual machine from others. For the RAM, we advise that you use as much as possible, but 8 GB should be enough.

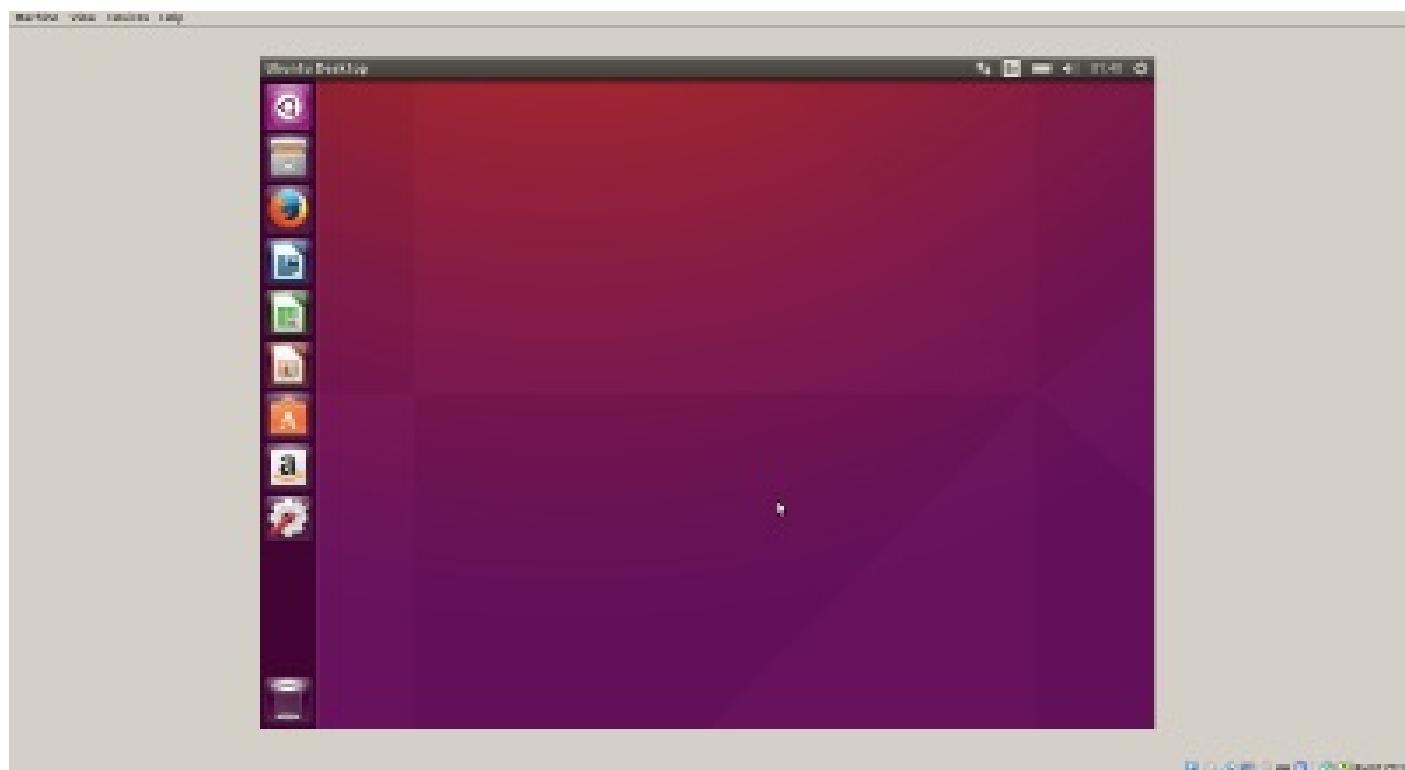
For the hard drive, use the existing virtual hard drive file `ubuntu 15.10 wily (64bit).vdi` downloaded before, as shown in the following screenshot:



After this, you can start your virtual machine by clicking on the Start button. Remember to select the right machine before you start it. In our case, we only have one, but you could have more:



Once the virtual machine starts, you should see another window, as seen in the following screenshot. It is the Ubuntu 15.10 OS with ROS installed (use osboxes.org as the password to log in):



When you finish these steps, install ROS Kinetic as you would on a regular computer following the steps of the previous sections, and you will have a full copy of ROS Kinetic that can be used with this section.

Using ROS from a Docker image

Docker is an open platform that helps to distribute applications and complete systems. In some ways, it is similar to a virtual machine, but it is much faster and more flexible; see <https://www.docker.com> or <https://dockerproject.org> for more information.

Installing Docker

In order to install it in Ubuntu, you only have to run the following:

```
| $ sudo apt-get install docker.io
```

Getting and using ROS Docker images and containers

Docker images are like virtual machines or systems already set up. There are servers that provide images like this, so the users only have to download them. The main server is Docker hub, found at <https://hub.docker.com>. There, it is possible to search for Docker images for different systems and configurations. In our case, we are going to use ROS Kinetic images already available. All ROS Docker images are listed in the official ROS repo images on the web at https://hub.docker.com/_/ros/. The ROS container image is pulled down with the following command:

```
| $ docker pull ros
```

There's a possibility that you may see this error:

```
~$ docker pull ros
FATA[0000] Post http://var/run/docker.sock/v1.18/images/create?fromImage=ros%3
Alatest: dial unix /var/run/docker.sock: permission denied. Are you trying to c
onnect to a TLS-enabled daemon without TLS?
```

You should either update your system or try adding your user to the `docker` group to resolve this:

```
| $ sudo usermod -a -G docker $(whoami)
```

You should see multiple Docker images getting downloaded simultaneously. Each image has a different hash name. This will take some time, especially on slow networks. You will see something like the following once it is done:

```
~$ docker pull ros
latest: Pulling from ros
808ef855e5b6: Pull complete
267903aa9bd1: Pull complete
d28d8a6a946d: Pull complete
ab033c88d533: Pull complete
0b409bffffca0: Pull complete
aa8ec2450c6b: Pull complete
fea18d173ca4: Pull complete
5c9bb5cbe512: Pull complete
ae87b758dd0d: Pull complete
9caddeb3affd3: Pull complete
9c28b2d84bd7: Pull complete
0c7cd879039b: Pull complete
e8530b0325b8: Pull complete
8ab2cb273ccb: Pull complete
c7411052df49: Pull complete
ec05b0e2ef74: Pull complete
c366f9bb95b3: Pull complete
e795c4487953: Pull complete
Digest: sha256:078fbdb221da8a3126eff2e283655f5a58e0342de272e38ef94631a1017568b86
Status: Downloaded newer image for ros:latest
```

The ROS Kinetic distribution can be pulled down using the corresponding tag, using the following command:

```
| $ docker pull ros:kinetic-robot
```

Although you do not need to know it, the images and containers are stored by Docker in `/var/lib/docker` by default.

Once the container is downloaded, we can run it interactively with the following command:

```
| $ docker run -it ros
```

This will be like entering a session inside the Docker container. This command will create a new container from the main image. Inside it we have a full Ubuntu system with ROS Kinetic already installed. We can install additional packages and run ROS nodes, as in a regular system. With `docker ps -a`, you can check all the containers available and the image they come from.

We have to set up the ROS environment inside the container in order to start using ROS. That is, we have to run the following command:

```
| $ source /opt/ros/kinetic/setup.bash
```

Docker containers can be stopped from other terminals using `docker stop`, and they can also be removed with `docker rm`. Docker also allows you to configure the container to expose the network, as well as mounting a host folder as volumes into it. In addition to this, it also supports a Python API, and has many other features. All this can be found in the official documentation at <https://docs.docker.com>. However, in principle, `docker run` should be enough, and we can even SSH into a running Docker container, as a regular machine, using its name. We can also open another terminal for a running container with the following command (where `NAME` is the name of the Docker container, that you can find using `docker ps -a`):

```
| $ docker exec -it NAME bash
```

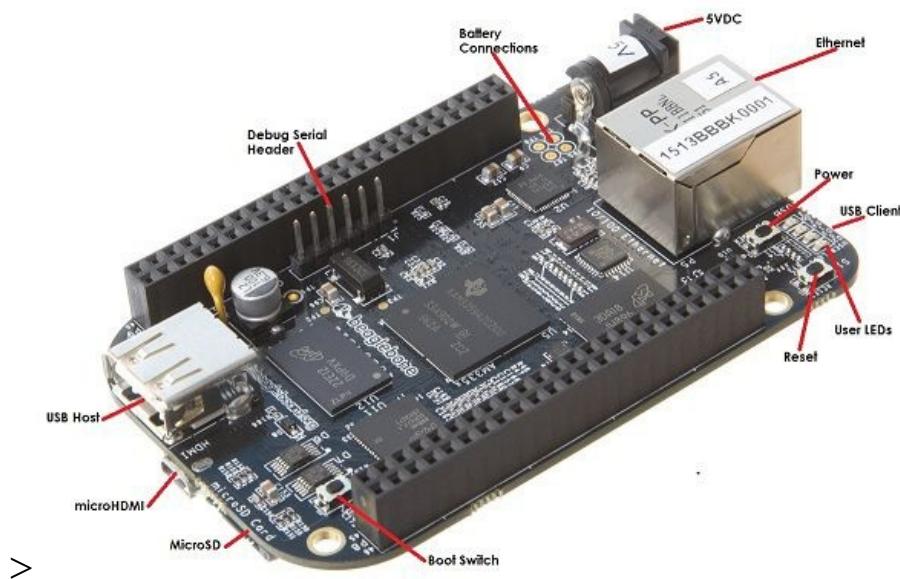
You can also create your own Docker images using `docker build` and specify what should be installed in them in `Dockerfile`. You can even publish them online with `docker push`, contributing them to the community or simply sharing your working setup. This section comes with a working Docker image and `Dockerfile` to build it, and you can find this by running `docker build` from the same folder where `Dockerfile` is. This Docker image is basically an extension of the ROS Kinetic one with the code of the section. The instructions to download and install it would be on the GitHub repository with the rest of the code.

Installing ROS in BeagleBone Black

BeagleBone Black is a low-cost development platform based on an ARM Cortex A8 processor. This board is fabricated with a Linux distribution called **Ångström**. Ångström was developed by a small group who wanted to unify Linux distribution for embedded systems. They wanted an operating system that was stable and user-friendly.

Texas Instruments designed BeagleBone Black thinking that the community of developers needed an on-board computer with some **general purpose input/output (GPIO)** pins. The BeagleBone Black platform is an evolution of the original BeagleBone. The main features of the board are an ARM Cortex A8 processor at 1 GHz with 512 MB RAM, and with Ethernet, USB, and HDMI connections and two headers of 46 GPIO pins.

This GPIO can be set up as digital I/O, ADC, PWM, or for communication protocol like I2C, SPI, or UART. The GPIO is an easy way to communicate with sensors and actuators directly from the BeagleBone without intermediaries. The following is a labeled image of BeagleBone:



When the BeagleBone board came out, it was not possible to install ROS on the Ångström distribution. For this reason, it was common to install an operating system based on Ubuntu on the BeagleBone. There are different versions of Ubuntu ARM compatible with the BeagleBone Black and ROS; we recommend that you use an image of Ubuntu ARM 16.04 Xenial armhf on the platform to work with ROS.

Now, an ROS version for Ångström distribution is ready to be installed; you can do so following the installation steps given at <http://wiki.ros.org/kinetic/Installation/Angstrom>. Despite this possibility, we have chosen to install ROS on Ubuntu ARM because these distributions are more common and can be used on other ARM-based boards such as UDOO, ODROIDU3, ODROIDX2, or Gumstick.

ARM technology is booming with the use of mobile devices such as smartphones and tablets. Apart from the increasing computer power of the ARM cortex, the great level of integration and low consumption has made this technology suitable for autonomous robotic systems. In the last few years, multiple ARM

platforms for developers have been launched in the market. Some of them have features similar to the BeagleBone Black, such as the Raspberry PI or the Gumstick Overo. Additionally, more powerful boards like GumstickDuoVero with a Dual Core ARM Cortex A9 or some quad core boards like OdroidU3, OdroidX2 or UDOO are now available.

Prerequisites

Before installing ROS on BeagleBone Black, we have to achieve some prerequisites. As this section is focused on ROS, we will list them without going into detail. There is a lot of information about BeagleBone Black and Ubuntu ARM available on websites, forums, and books that you can check out.

First, we have to install an Ubuntu ARM distribution compatible with ROS, so an image of Ubuntu ARM is needed. You can obtain an Ubuntu 16.04 Xenial armhf using `wget` with the following command:

```
| $ wget https://rcn-ee.com/rootfs/2016-10-06/elinux/ubuntu-16.04.1-console-armhf-2016-10-06.tar.xz
```

From the URL <https://rcn-ee.com/rootfs> you can look for newer versions too. This version is the one mentioned in the official documentation at <http://elinux.org/BeagleBoardUbuntu>.

Once the image is downloaded, the next step is installing it on a microSD card. First, unpack the image with the following commands:

```
| $ tar xf ubuntu-16.04.1-console-armhf-2016-10-06.tar.xz  
| $ cd ubuntu-16.04.1-console-armhf-2016-10-06
```

Insert a microSD of 2 GB or more on the card reader of your computer and install the Ubuntu image on it with the following setup script:

```
| $ sudo ./setup_sdcard.sh --mmc DEVICE --dtb BOARD
```

In the preceding script, `DEVICE` is the device where the microSD appears on the system, for example `/dev/sdb`, and `BOARD` is the board name. For the BeagleBone Black it would be `beaglebone`. So, assuming the microSD is in `/dev/mmcblk0`, and you are using a BeagleBone Black, the command would be as follows:

```
| $ sudo ./setup_sdcard.sh --mmc /dev/mmcblk0 --dtb beaglebone
```

If you do not know the device assigned to the microSD, you can use the following command:

```
| $ sudo ./setup_sdcard.sh --probe-mmc
```

Once we have Ubuntu ARM on our platform, the BeagleBone Black network interfaces must be configured to provide access to the network, so you will have to configure the network settings, such as the IP, DNS, and gateway.

Remember that the easiest way could be mounting the SD card in another computer and editing `/etc/network/interfaces`.

Another easy way consists on using an Ethernet cable and running the DHCP client to obtain an IP address.

```
| $ sudo dhclient eth0
```

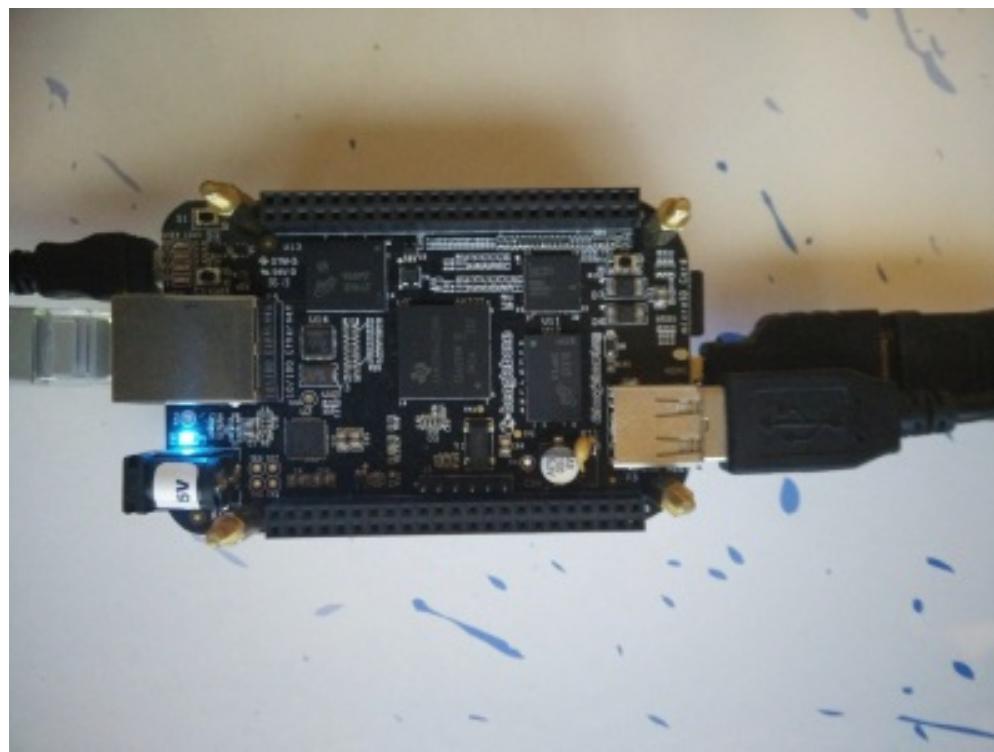
For this, you need to boot up the BeagleBone Black with the microSD. For that, you need to keep the **S2**

button press before you power on the board, either with the DC or USB connector. After some minutes the system will show the login prompt. Log in with the user `ubuntu` and password `tempwd` (default ones) and then run the DHCP client command shown above with the Ethernet cable connected. Then, you can check the IP address assigned using (look at the `inet addr: value`):

```
| $ ifconfig eth0
```

In our setup, we have connected to the BeagleBone Black the following devices (as shown in the image below):

- HDMI (with microHDMI adapter) cable to see terminal prompt on the screen during the network setup; after that we can SSH into the board
- Keyboard connected through USB
- Power supplied through the microUSB connector
- Ethernet cable to access the Internet, as explained so far



After setting up the network, we should install the packages, programs, and libraries that ROS will need. Now that the network is up, we can also SSH into the board with (assuming the IP address assigned to it is `192.168.1.6`):

```
| $ ssh ubuntu@192.168.1.6
```

We are going to follow the instructions in <http://wiki.ros.org/indigo/Installation/UbuntuARM>, but with the changes required so they work for ROS kinetic (note they are still for indigo on the web). The first step consists on setting up the repository sources so we can install ROS.

```
| $ sudo vi /etc/apt/sources.list
```

And add restricted to the sources, so you have something like this:

```
| deb http://ports.ubuntu.com/ubuntu-ports/ xenial main restricted universe multiverse
```

```

#deb-src http://ports.ubuntu.com/ubuntu-ports/ xenial main restricted universe multiverse
deb http://ports.ubuntu.com/ubuntu-ports/ xenial-updates main restricted universe multiverse
#deb-src http://ports.ubuntu.com/ubuntu-ports/ xenial-updates main restricted universe
multiverse
#Kernel source (repos.rcn-ee.com) : https://github.com/RobertCNelson/linux-stable-rcn-ee
#
#git clone https://github.com/RobertCNelson/linux-stable-rcn-ee
#cd ./linux-stable-rcn-ee
#git checkout `uname -r` -b tmp
#
deb [arch=armhf] http://repos.rcn-ee.com/ubuntu/ xenial main
#deb-src [arch=armhf] http://repos.rcn-ee.com/ubuntu/ xenial main

```

Then, update the sources running:

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

The operating system for BeagleBone Black is set up for micro SD cards with 1-4 GB. This memory space is very limited if we want to use a large part of the ROS Kinetic packages. In order to solve this problem, we can use SD cards with more space and expand the file system to occupy all the space available with re-partitioning.

So if we want to work with a bigger memory space, it is recommended to expand the BeagleBone Black memory file system. This process is further explained at http://elinux.org/Beagleboard:Expanding_File_System_Partition_On_A_microSD.

Although the following steps are not needed in general, they are here in case you need them in your particular case. You can proceed by following these commands:

1. We need to become a super user, so we will type the following command and our password:

```
| $ sudo su
```

2. We will look at the partitions of our SD card:

```
| $ fdisk /dev/mmcblk0
```

3. On typing **p**, the two partitions of the SD card will be shown:

```
| $ p
```

4. After this, we will delete one partition by typing **d** and then we will type **2** to indicate that we want to delete `/dev/mmcblk0p2`:

```
| $ d
| $ 2
```

5. On typing **n**, a new partition will be created; if we type **p** it will be a primary partition. We will indicate that we want to number it as the second partition by typing **2**:

```
| $ n
| $ p
| $ 2
```

6. You can write these changes by typing **w** if everything is right, or eliminate the changes with **Ctrl + Z**:

| \$ w

7. We should reboot the board after finishing:

| \$ reboot

8. Once again, become a super user once the reboot is complete:

| \$ sudo su

9. Finally, run the following command to execute the expansion of the memory file system of the operating system:

| \$ resize2fs /dev/mmcblk0p2

Now we should be ready to install ROS. At this point, the process of installation is pretty similar to the PC installation previously explained in this chapter, so we should be familiar with it. We will see that the main difference when installing ROS on BeagleBone Black is that we can't install the ROS full-desktop; we must install it package by package.

Setting up the local machine and source.list file

Now you will start setting up your local machine:

```
| $ sudo update-locale LANG=C LANGUAGE=C LC_ALL=C LC_MESSAGES=POSIX
```

After this, we will configure the source lists depending on the Ubuntu version that we have installed in BeagleBone Black. The number of Ubuntu versions compatible with BeagleBone Black is limited, and only active builds can be found for Ubuntu 16.40 Xenial armhf, the most popular version of Ubuntu ARM. Run the following to install the Ubuntu armhf repositories:

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

Setting up your keys

As explained previously, this step is needed to confirm that the origin of the code is correct and that no one has modified the code or programs without the knowledge of the owner:

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

Installing the ROS packages

Before the installation of ROS packages, we must update the system to see all the packages on the ROS repository just added:

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

This part of the installation is slightly different for the BeagleBone Black. There are a lot of libraries and packages in ROS, and not all of them compile fully on an ARM, so it is not possible to make a full desktop installation. It is recommended that you install them package by package to ensure that they will work on an ARM platform.

You can try to install ROS-base, known as ROS Bare Bones. ROS-base installs the ROS package along with the build and communications libraries but does not include the GUI tools (press *ENTER (Y)* when prompted):

```
| $ sudo apt-get install ros-kinetic-ros-base
```

We can install specific ROS packages with the following command:

```
| $ sudo apt-get install ros-kinetic-PACKAGE
```

If you need to find the ROS packages available for BeagleBone Black, you can run the following command:

```
| $ apt-cache search ros-kinetic
```

For example, the following packages are the basics (already installed as ros-base dependencies) that work with ROS and can be installed individually using `apt-get install`:

```
$ sudo apt-get install ros-kinetic-ros  
$ sudo apt-get install ros-kinetic-roslaunch  
$ sudo apt-get install ros-kinetic-rosparam  
$ sudo apt-get install ros-kinetic-rosservice
```

Although, theoretically, not all the packages of ROS are supported by BeagleBone Black, in practice, we have been able to migrate entire projects developed on PC to BeagleBone Black. We tried a lot of packages, and the only one that we could not install was rviz, which is indeed not recommended to run on it.

Initializing rosdep for ROS

The `rosdep` command-line tool must be installed and initialized before you can use ROS. This allows you to easily install libraries and solve system dependencies for the source you want to compile, and is required to run some core components in ROS. You can use the following commands to install and initialize `rosdep`:

```
$ sudo apt-get install python-rosdep  
$ sudo rosdep init  
$ rosdep update
```

Setting up the environment in the BeagleBone Black

If you have arrived at this step, congratulations, because you have installed ROS in your BeagleBone Black. The ROS environment variables can be added to your bash, so they will be added every time a shell is launched:

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

We have to be careful if we have more than one version of ROS in our system. The `bashrc` setup must use the variables of the version being used only.

If we want to set up the environment in the current shell, we will run the following command:

```
| $ source /opt/ros/kinetic/setup.bash
```

Getting rosinstall for BeagleBone Black

Rosinstall is a common command-line tool in ROS that helps us to install packages easily. It can be installed on Ubuntu with the following command:

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

Basic ROS example on the BeagleBone Black

As a basic example, you can run a ROS core on one terminal on the BeagleBone Black:

```
| $ roscore
```

And from another terminal publish a pose message (note you can press *Tab Tab* after `geometry_msgs/Pose` and it'd autocomplete the message fields, that then you need to change since it'd have the default values):

```
$ rostopic pub /dummy geometry_msgs/Pose
Position:
x: 1.0
y: 2.0
z: 3.0
Orientation:
x: 0.0
y: 0.0
z: 0.0
w: 1.0 -r 10
```

Now, from your laptop (in the same network), you can set `ROS_MASTER_URI` to point to the BeagleBone Black (IP `192.168.1.6` in our case):

```
| $ export ROS_MASTER_URI=http://192.168.1.6:11311
```

And now you should be able to see the pose published from the BeagleBone Black on your laptop doing:

```
$ rostopic echo -n1 /dummy
Position:
x: 1.0
y: 2.0
z: 3.0
Orientation:
x: 0.0
y: 0.0
z: 0.0
w: 1.0
---
```

If you use a PoseStamped, you can even visualize it on rviz.

From this point, you can check multiple projects at <http://wiki.ros.org/BeagleBone>, as well as another installation option, which uses the Angstrom OS instead of Ubuntu, but it does not support ROS Kinetic at the moment.

Summary

In this chapter, we have installed ROS Kinetic on different physical and virtual devices (PC, VirtualBox, and BeagleBone Black) in Ubuntu. With these steps, you should have everything needed to start working with ROS installed on your system, and you can also practice the examples in this section. You also have the option of installing ROS using the source code. This option is for advanced users and we recommend you use only the installation from the apt repositories as it is more common and normally does not give any errors or problems.

It is a good idea to play around with ROS and its installation on a virtual machine. That way, if you have problems with the installation or with something else, you can reinstall a new copy of your operating system and start again.

Normally, with virtual machines, you will not have access to real hardware, such as sensors or actuators. In any case, you can use it for testing the algorithms.

ROS Architecture and Concepts

Once you have installed ROS, you're probably thinking, *OK, I have installed it, so now what?* In this chapter, you will learn the structure of ROS and the parts it is made up of. Furthermore, you will start to create nodes and packages and use ROS with examples using **Turtlesim**.

The ROS architecture has been designed and divided into three sections or levels of concepts:

- The Filesystem level
- The Computation Graph level
- The Community level

The first level is the **Filesystem level**. In this level, a group of concepts are used to explain how ROS is internally formed, the folder structure, and the minimum number of files that it needs to work.

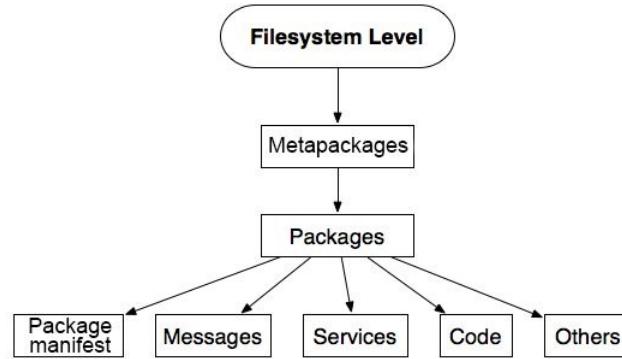
The second level is the **Computation Graph level**, where communication between processes and systems happens. In this section, we will see all the concepts and mechanisms that ROS has to set up systems, handle all the processes, and communicate with more than a single computer, and so on.

The third level is the **Community level**, which comprises a set of tools and concepts to share knowledge, algorithms, and code between developers. This level is of great importance; as with most open source software projects, having a strong community not only improves the ability of newcomers to understand the intricacies of the software, as well as solve the most common issues, it is also the main force driving its growth.

Understanding the ROS Filesystem level

The ROS Filesystem is one of the strangest concepts to grasp when starting to develop projects in ROS, but with time and patience, the reader will easily become familiar with it and realize its value for managing projects and its dependencies.

The main goal of the ROS Filesystem is to centralize the build process of a project, while at the same time provide enough flexibility and tooling to decentralize its dependencies.



Similar to an operating system, an ROS program is divided into folders, and these folders have files that describe their functionalities:

- **Packages:** Packages form the atomic level of ROS. A package has the minimum structure and content to create a program within ROS. It may have ROS runtime processes (nodes), configuration files, and so on.
- **Package manifests:** Package manifests provide information about a package, licenses, dependencies, compilation flags, and so on. A package manifest is managed with a file called `package.xml`.
- **Metapackages:** When you want to aggregate several packages in a group, you will use metapackages. In **ROS Fuerte**, this form for ordering packages was called *Stacks*. To maintain the simplicity of ROS, the stacks were removed, and now, metapackages make up this function. In ROS, there exists a lot of these metapackages; for example, the navigation stack.
- **Metapackage manifests:** Metapackage manifests (`package.xml`) are similar to a normal package, but with an `export` tag in XML. It also has certain restrictions in its structure.
- **Message (msg) types:** A message is the information that a process sends to other processes. ROS has a lot of standard types of messages. Message descriptions are stored in `my_package/msg/MyMessageType.msg`.
- **Service (srv) types:** Service descriptions, stored in `my_package/srv/MyServiceType.srv`, define the request and response data structures for services provided by each process in ROS.

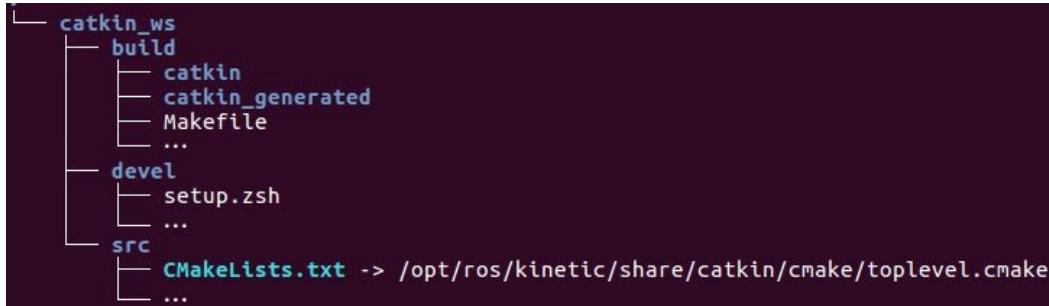
In the following screenshot, you can see the content of the `turtlesim` package. What you see is a series of files and folders with code, images, launch files, services, and messages. Keep in mind that the screenshot was edited to show a short list of files; the real package has more:

```
turtlesim
├── CHANGELOG.rst
├── CMakeLists.txt
└── images
    └── kinetic.png
include
└── turtlesim
launch
└── multisim.launch
msg
└── Color.msg
    └── Pose.msg
package.xml
src
└── turtle.cpp
    └── turtle_frame.cpp
    └── turtlesim
        └── turtlesim.cpp
srv
└── Kill.srv
    └── SetPen.srv
    └── Spawn.srv
    └── TeleportAbsolute.srv
    └── TeleportRelative.srv
```

The workspace

In general terms, the workspace is a folder which contains packages, those packages contain our source files and the environment or workspace provides us with a way to compile those packages. It is useful when you want to compile various packages at the same time and it is a good way of centralizing all of our developments.

A typical workspace is shown in the following screenshot. Each folder is a different space with a different role:



- **The source space:** In the source space (the `src` folder), you put your packages, projects, clone packages, and so on. One of the most important files in this space is `CMakeLists.txt`. The `src` folder has this file because it is invoked by `cmake` when you configure the packages in the workspace. This file is created with the `catkin_init_workspace` command.
- **The build space:** In the `build` folder, `cmake` and `catkin` keep the cache information, configuration, and other intermediate files for our packages and projects.
- **Development (devel) space:** The `devel` folder is used to keep the compiled programs. This is used to test the programs without the installation step. Once the programs are tested, you can install or export the package to share with other developers.

You have two options with regard to building packages with `catkin`. The first one is to use the standard `CMake` workflow. With this, you can compile one package at a time, as shown in the following commands:

```
| $ cmake packageToBuild/
| $ make
```

If you want to compile all your packages, you can use the `catkin_make` command line, as shown in the following commands:

```
| $ cd workspace
| $ catkin_make
```

Both commands build the executable in the build space directory configured in ROS.

Another interesting feature of ROS is its overlays. When you are working with a package of ROS, for example, Turtlesim, you can do it with the installed version, or you can download the source file and compile it to use your modified version.

ROS permits you to use your version of this package instead of the installed version. This is very useful information if you are working on an upgrade of an installed package. You might not understand the utility of this at the moment, but in the following chapters we will use this feature to create our own plugins.

Packages

Usually when we talk about packages, we refer to a typical structure of files and folders. This structure looks as follows:

- `include/package_name/`: This directory includes the headers of the libraries that you would need.
- `msg/`: If you develop nonstandard messages, put them here.
- `scripts/`: These are executable scripts that can be in Bash, Python, or any other scripting language.
- `src/`: This is where the source files of your programs are present. You can create a folder for nodes and nodelets or organize it as you want.
- `srv/`: This represents the service (`srv`) types.
- `CMakeLists.txt`: This is the `CMake` build file.
- `package.xml`: This is the package manifest.

To create, modify, or work with packages, ROS gives us tools for assistance, some of which are as follows:

- `rospack`: This command is used to get information or find packages in the system.
- `catkin_create_pkg`: This command is used when you want to create a new package.
- `catkin_make`: This command is used to compile a workspace.
- `rosdep`: This command installs the system dependencies of a package.
- `rqt_dep`: This command is used to see the package dependencies as a graph. If you want to see the package dependencies as a graph, you will find a plugin called `package graph` in `rqt`. Select a package and see the dependencies.

To move between packages and their folders and files, ROS gives us a very useful package called `rosbash`, which provides commands that are very similar to Linux commands. The following are a few examples:

- `roscd`: This command helps us change the directory. This is similar to the `cd` command in Linux.
- `rosed`: This command is used to edit a file.
- `rosdp`: This command is used to copy a file from a package.
- `rosd`: This command lists the directories of a package.
- `rosls`: This command lists the files from a package. This is similar to the `ls` command in Linux.

Every package must contain a `package.xml` file, as it is used to specify information about the package. If you find this file inside a folder, it is very likely that this folder is a package or a metapackage.

If you open the `package.xml` file, you will see information about the name of the package, dependencies, and so on. All of this is to make the installation and the distribution of these packages easier .

Two typical tags that are used in the `package.xml` file are `<build_depend>` and `<run_depend>`.

The `<build_depend>` tag shows which packages must be installed before installing the current package. This is because the new package might use functionality contained in another package.

The `<run_depend>` tag shows the packages that are necessary for running the code of the package. The following screenshot is an example of the `package.xml` file:

```
?xml version="1.0"?
<package>
  <name>example</name>
  <version>0.0.1</version>
  <description>
    this is a example.
  </description>
  <maintainer email="test@test.com">test</maintainer>
  <license>BSD</license>

  <url type="website">http://www.test.com</url>
  <author>test</author>

  <buildtool_depend>catkin</buildtool_depend>
  <build_depend>geometry_msgs</build_depend>

  <run_depend>geometry_msgs</run_depend>
</package>
~
```

Metapackages

As we have shown earlier, metapackages are special packages with only one file inside; this file is `package.xml`. This package does not have other files, such as code, includes, and so on.

Metapackages are used to refer to others packages that are normally grouped following a feature-like functionality, for example, navigation stack, `ros_tutorials`, and so on.

You can convert your stacks and packages from ROS Fuerte to Kinetic and `catkin` using certain rules for migration. These rules can be found at http://wiki.ros.org/catkin/migrating_from_rosbuild.

In the following screenshot, you can see the content from the `package.xml` file in the `ros_tutorials` metapackage. You can see the `<export>` tag and the `<run_depend>` tag. These are necessary in the package manifest, which is also shown in the following screenshot:

```
<?xml version="1.0"?>
<package>
  ...
  <buildtool_depend>catkin</buildtool_depend>
  ...
  <run_depend>roscpp_tutorials</run_depend>
  <run_depend>rospy_tutorials</run_depend>
  <run_depend>turtlesim</run_depend>
  ...
  <export>
    <metapackage/>
  </export>
  ...
</package>
```

If you want to locate the `ros_tutorials` metapackage, you can use the following command:

```
| $ rosstack find ros_tutorials
```

The output will be a path, such as `/opt/ros/kinetic/share/ros_tutorials`.

To see the code inside, you can use the following command line:

```
| $ vim /opt/ros/kinetic/ros_tutorials/package.xml
```

Remember that Kinetic uses metapackages, not stacks, but the `rosstack find` command-line tool is also capable of finding metapackages.

Messages

ROS uses a simplified message description language to describe the data values that ROS nodes publish. With this description, ROS can generate the right source code for these types of messages in several programming languages.

ROS has a lot of messages predefined, but if you develop a new message, it will be in the `msg/` folder of your package. Inside that folder, certain files with the `.msg` extension define the messages.

A message must have two main parts: *fields* and *constants*. Fields define the type of data to be transmitted in the message, for example, `int32`, `float32`, and `string`, or new types that you created earlier, such as `type1` and `type2`. Constants define the name of the fields.

An example of an `msg` file is as follows:

```
int32 id  
float32 vel  
string name
```

In ROS, you can find a lot of standard types to use in messages, as shown in the following table list:

| Primitive type | Serialization | C++ | Python |
|-----------------------|---------------------|-------------------------|---------------------|
| <code>bool</code> (1) | 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> | | <code>int64_t</code> | <code>long</code> |

| | | | |
|----------|-------------------------------|---------------|----------------|
| | signed 64-bit int | | |
| uint64 | unsigned 64-bit int | uint64_t | long |
| float32 | 32-bit IEEE float | float | float |
| float64 | 64-bit IEEE float | double | float |
| string | ascii string (4) | std::string | string |
| time | secs/nsecs signed 32-bit ints | ros::Time | rospy.Time |
| duration | secs/nsecs signed 32-bit ints | ros::Duration | rospy.Duration |

A special type in ROS is the header type. This is used to add the time, frame, and sequence number. This permits you to have the messages numbered, to see who is sending the message, and to have more functions that are transparent for the user and that ROS is handling.

The header type contains the following fields:

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

You can see the structure using the following command:

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

Thanks to the header type, it is possible to record the timestamp and frame of what is happening with the robot, as we will see in upcoming chapters.

ROS provides certain tools to work with messages. The `rosmsg` tool prints out the message definition information and can find the source files that use a message type.

In upcoming sections, we will see how to create messages with the right tools.

Services

ROS uses a simplified service description language to describe ROS service types. This builds directly upon the ROS `msg` format to enable request/response communication between nodes. Service descriptions are stored in `.srv` files in the `srv/` subdirectory of a package.

To call a service, you need to use the package name, along with the service name; for example, you will refer to the `sample_package1/srv/sample1.srv` file as `sample_package1/sample1`.

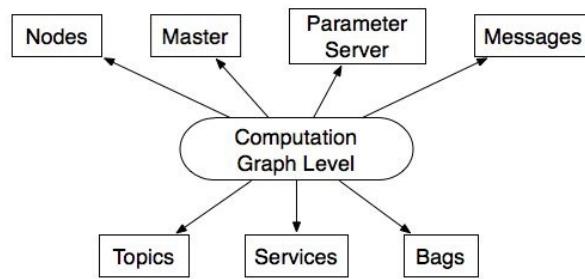
Several tools exist to perform operations on services. The `rossrv` tool prints out the service descriptions and packages that contain the `.srv` files and finds source files that use a service type.

If you want to create a service, ROS can help you with the service generator. These tools generate code from an initial specification of the service. You only need to add the `gensrv()` line to your `CMakeLists.txt` file.

In upcoming sections, you will learn how to create your own services.

Understanding the ROS Computation Graph level

ROS creates a network where all the processes are connected. Any node in the system can access this network, interact with other nodes, see the information that they are sending, and transmit data to the network:



The basic concepts in this level are nodes, the master, Parameter Server, messages, services, topics, and bags, all of which provide data to the graph in different ways and are explained in the following list:

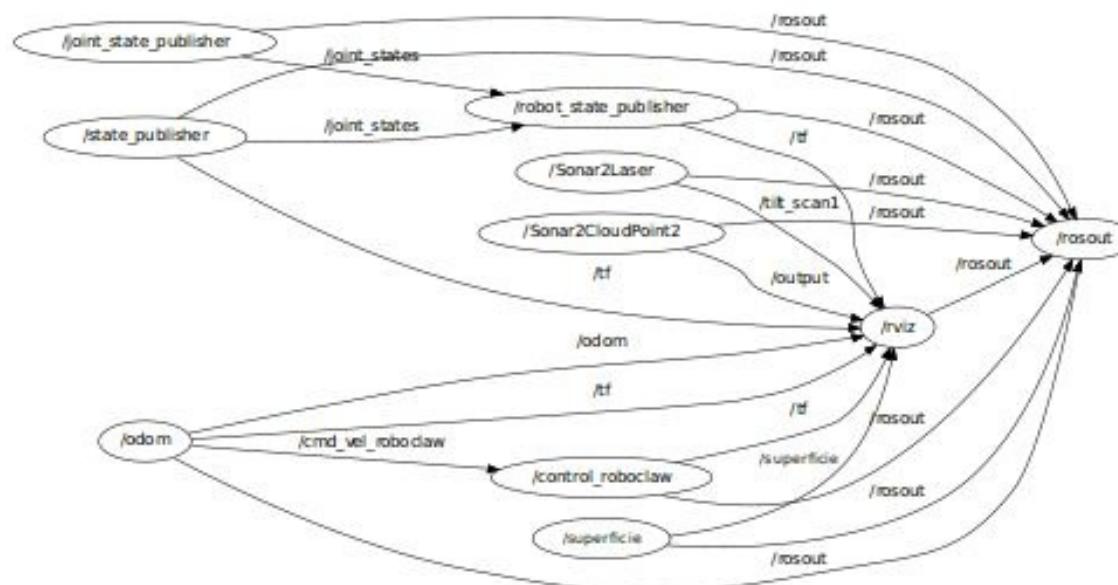
- **Nodes:** Nodes are processes where computation is done. If you want to have a process that can interact with other nodes, you need to create a node with this process to connect it to the ROS network. Usually, a system will have many nodes to control different functions. You will see that it is better to have many nodes that provide only a single functionality, rather than have a large node that makes everything in the system. Nodes are written with an ROS client library, for example, `roscpp` or `rospy`.
- **The master:** The master provides the registration of names and the lookup service to the rest of the nodes. It also sets up connections between the nodes. If you don't have it in your system, you can't communicate with nodes, services, messages, and others. In a distributed system, you will have the master in one computer, and you can execute nodes in this or other computers.
- **Parameter Server:** Parameter Server gives us the possibility of using keys to store data in a central location. With this parameter, it is possible to configure nodes while it's running or to change the working parameters of a node.
- **Messages:** Nodes communicate with each other through messages. A message contains data that provides information to other nodes. ROS has many types of messages, and you can also develop your own type of message using standard message types.
- **Topics:** Each message must have a name to be routed by the ROS network. When a node is sending data, we say that the node is publishing a topic. Nodes can receive topics from other nodes by simply subscribing to the topic. A node can subscribe to a topic even if there aren't any other nodes publishing to this specific topic. This allows us to decouple the production from the consumption. It's important that topic names are unique to avoid problems and confusion between topics with the same name.
- **Services:** When you publish topics, you are sending data in a many-to-many fashion, but when you need a request or an answer from a node, you can't do it with topics. Services give us the possibility of interacting with nodes. Also, services must have a unique name. When a node has a service, all

the nodes can communicate with it, thanks to ROS client libraries.

- **Bags:** Bags are a format to save and play back the ROS message data. Bags are an important mechanism to store data, such as sensor data, that can be difficult to collect but is necessary to develop and test algorithms. You will use bags a lot while working with complex robots.

In the following diagram, you can see the graphic representation of this level. It represents a real robot working in real conditions. In the graph, you can see the nodes, the topics, which node is subscribed to a topic, and so on. This graph does not represent messages, bags, Parameter Server, and services. It is necessary for other tools to see a graphic representation of them. The tool used to create the graph is `rqt_graph`; you will learn more about it in Chapter 3, *Visualization and Debugging Tools*.

These concepts are implemented in the `ros_comm` repository.



Nodes and nodelets

Nodes are executable that can communicate with other processes using topics, services, or the Parameter Server. Using nodes in ROS provides us with fault tolerance and separates the code and functionalities, making the system simpler.

ROS has another type of node called **nodelets**. These special nodes are designed to run multiple nodes in a single process, with each nodelet being a thread (light process). This way, we avoid using the ROS network among them, but permit communication with other nodes. With that, nodes can communicate more efficiently, without overloading the network. Nodelets are especially useful for camera systems and 3D sensors, where the volume of data transferred is very high.

A node must have a unique name in the system. This name is used to permit the node to communicate with another node using its name without ambiguity. A node can be written using different libraries, such as `roscpp` and `rospy`; `roscpp` is for C++ and `rospy` is for Python. Throughout this section, we will use `roscpp`.

ROS has tools to handle nodes and give us information about it, such as `rosnode`. The `rosnode` tool is a command-line tool used to display information about nodes, such as listing the currently running nodes. The supported commands are as follows:

- `rosnode info NODE`: This prints information about a node
- `rosnode kill NODE`: This kills a running node or sends a given signal
- `rosnode list`: This lists the active nodes
- `rosnode machine hostname`: This lists the nodes running on a particular machine or lists machines
- `rosnode ping NODE`: This tests the connectivity to the node
- `rosnode cleanup`: This purges the registration information from unreachable nodes

In upcoming sections, you will learn how to use these commands with examples.

A powerful feature of ROS nodes is the possibility of changing parameters while you start the node. This feature gives us the power to change the node name, topic names, and parameter names. We use this to reconfigure the node without recompiling the code so that we can use the node in different scenes.

An example of changing a topic name is as follows:

```
| $ rosrun book_tutorials tutorialX topic1:=/level1/topic1
```

This command will change the topic name `topic1` to `/level1/topic1`. It's likely that you will not understand this at this stage, but you will find the utility of it in upcoming chapters.

To change parameters in the node, you can do something similar to changing the topic name. For this, you only need to add an underscore (_) to the parameter name; for example:

```
| $ rosrun book_tutorials tutorialX _param:=9.0
```

The preceding command will set `param` to the float number `9.0`.

Bear in mind that you cannot use names that are reserved by the system. They are as follows:

- `__name`: This is a special, reserved keyword for the name of the node
- `__log`: This is a reserved keyword that designates the location where the node's log file should be written
- * `__ip` and `__hostname`: These are substitutes for `ROS_IP` and `ROS_HOSTNAME`
- * `__master`: This is a substitute for `ROS_MASTER_URI`
- * `__ns`: This is a substitute for `ROS_NAMESPACE`

Topics

Topics are buses used by nodes to transmit data. Topics can be transmitted without a direct connection between nodes, which means that the production and consumption of data is decoupled. A topic can have various subscribers and can also have various publishers, but you should be careful when publishing the same topic with different nodes as it can create conflicts.

Each topic is strongly typed by the ROS message type used to publish it, and nodes can only receive messages from a matching type. A node can subscribe to a topic only if it has the same message type.

The topics in ROS can be transmitted using TCP/IP and UDP. The TCP/IP-based transport is known as **TCPROS** and uses the persistent TCP/IP connection. This is the default transport used in ROS.

The UDP-based transport is known as **UDPROS** and is a low-latency, lossy transport, so it is best suited to tasks such as teleoperation.

ROS has a tool to work with topics called `rostopic`. It is a command-line tool that gives us information about the topic or publishes data directly on the network.

This tool has the following parameters:

- `rostopic bw /topic`: This displays the bandwidth used by the topic.
- `rostopic echo /topic`: This prints messages to the screen.
- `rostopic find message_type`: This finds topics by their type.
- `rostopic hz /topic`: This displays the publishing rate of the topic.
- `rostopic info /topic`: This prints information about the topic, such as its message type, publishers, and subscribers.
- `rostopic list`: This prints information about active topics.
- `rostopic pub /topic type args`: This publishes data to the topic. It allows us to create and publish data in whatever topic we want, directly from the command line.
- `rostopic type /topic`: This prints the topic type, that is, the type of message it publishes.

We will learn how to use this command-line tool in upcoming sections.

Services

When you need to communicate with nodes and receive a reply in an RPC fashion, you cannot do it with topics; you need to do it with services.

Services are developed by the user and standard services don't exist for nodes. The files with the source code of the services are stored in the `srv` folder.

Similar to topics, services have an associated service type that is the package resource name of the `.srv` file. As with other ROS filesystem-based types, the service type is the package name and the name of the `.srv` file. For example, the `chapter2_tutorials/srv/chapter2_srv1.srv` file has the `chapter2_tutorials/chapter2_srv1` service type.

ROS has two command-line tools to work with services: `rossrv` and `rosservice`. With `rossrv`, we can see information about the services' data structure, and it has exactly the same usage as `rosmsg`.

With `rosservice`, we can list and query services. The supported commands are as follows:

- `rosservice call /service args`: This calls the service with the arguments provided
- `rosservice find msg-type`: This finds services by service type
- `rosservice info /service`: This prints information about the service
- `rosservice list`: This lists the active services
- `rosservice type /service`: This prints the service type
- `rosservice uri /service`: This prints the ROSRPC URI service

Messages

A node publishes information using messages which are linked to topics. The message has a simple structure that uses standard types or types developed by the user.

Message types use the following standard ROS naming convention; the name of the package, then / and then the name of the .msg file. For example, std_msgs/ msg/String.msg has the std_msgs/String message type.

ROS has the `rosmsg` command-line tool to get information about messages.

The accepted parameters are as follows:

- `rosmsg show`: This displays the fields of a message
- `rosmsg list`: This lists all messages
- `rosmsg package`: This lists all of the messages in a package
- `rosmsg packages`: This lists all of the packages that have the message
- `rosmsg users`: This searches for code files that use the message type
- `rosmsg md5`: This displays the MD5 sum of a message

Bags

A **bag** is a file created by ROS with the `.bag` format to save all of the information of the messages, topics, services, and others. You can use this data later to visualize what has happened; you can play, stop, rewind, and perform other operations with it.

The bag file can be reproduced in ROS just as a real session can, sending the topics at the same time with the same data. Normally, we use this functionality to debug our algorithms.

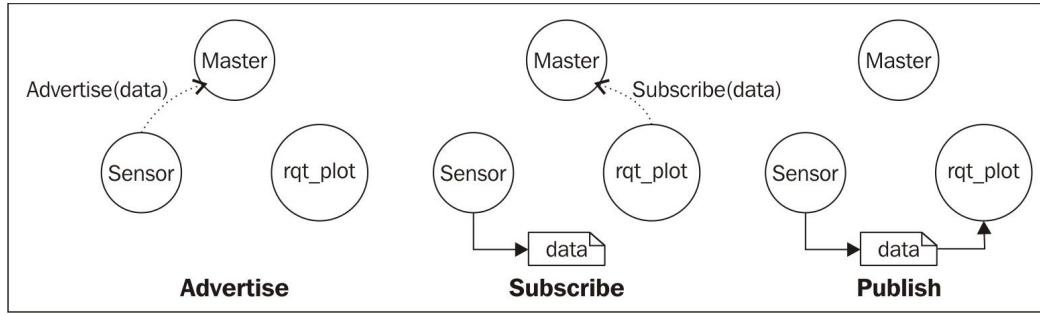
To use bag files, we have the following tools in ROS:

- `rosbag`: This is used to record, play, and perform other operations
- `rqt_bag`: This is used to visualize data in a graphic environment
- `rostopic`: This helps us see the topics sent to the nodes

The ROS master

The **ROS master** provides naming and registration services to the rest of the nodes in the ROS system. It tracks publishers and subscribers to topics as well as services. The role of the master is to enable individual ROS nodes to locate one another.

Once these nodes have located each other, they communicate with each other in a peer-to-peer fashion. You can see in a graphic example the steps performed in ROS to advertise a topic, subscribe to a topic, and publish a message, in the following diagram:



The master also provides Parameter Server. The master is most commonly run using the `roscore` command, which loads the ROS master, along with other essential components.

Parameter Server

Parameter Server is a shared, multivariable dictionary that is accessible via a network. Nodes use this server to store and retrieve parameters at runtime.

Parameter Server is implemented using XMLRPC and runs inside the ROS master, which means that its API is accessible via normal XMLRPC libraries. XMLRPC is a **Remote Procedure Call (RPC)** protocol that uses XML to encode its calls and HTTP as a transport mechanism.

Parameter Server uses XMLRPC data types for parameter values, which include the following:

- 32-bit integers
- Booleans
- Strings
- Doubles
- ISO8601 dates
- Lists
- Base64-encoded binary data

ROS has the `rosparam` tool to work with Parameter Server. The supported parameters are as follows:

- `rosparam list`: This lists all the parameters in the server
- `rosparam get parameter`: This gets the value of a parameter
- `rosparam set parameter value`: This sets the value of a parameter
- `rosparam delete parameter`: This deletes a parameter
- `rosparam dump file`: This saves Parameter Server to a file
- `rosparam load file`: This loads a file (with parameters) on Parameter Server

Understanding the ROS Community level

The ROS Community level concepts are the ROS resources that enable separate communities to exchange software and knowledge. These resources include the following:

- **Distributions:** ROS distributions are collections of versioned metapackages that you can install. ROS distributions play a similar role to Linux distributions. They make it easier to install a collection of software, and they also 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 Wiki is the main forum for documenting information about ROS. Anyone can sign up for an account, contribute their own documentation, provide corrections or updates, write tutorials, and more.
- **Bug ticket system:** If you find a problem or want to propose a new feature, ROS has this resource to do it.
- **Mailing lists:** The ROS user-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:** Users can ask questions on forums using this resource.
- **Blog:** You can find regular updates, photos, and news at <http://www.ros.org/news>.

Tutorials to practise with ROS

It is time for you to practise what you have learned until now. In upcoming sections, you will see examples for you to practise along with the creation of packages, using nodes, using Parameter Server, and moving a simulated robot with Turtlesim.

Navigating through the ROS filesystem

As explained before, ROS provides a number of command-line tools for navigating through the filesystem. In this subsection, we will explain the most used ones, with examples.

To get information about the packages and stacks in our environment, such as their paths, dependencies, and so on, we can use `rospack` and `rosstack`. On the other hand, to move through packages and stacks, as well as listing their contents, we will use `roscd` and `rosls`.

For example, if you want to find the path of the `turtlesim` package, you can use the following command:

```
| $ rospack find turtlesim
```

Which will then result in the following output:

```
| /opt/ros/kinetic/share/turtlesim
```

The same thing happens with the metapackages that you have installed in the system. An example of this is as follows:

```
| $ rosstack find ros_comm
```

You will obtain the path for the `ros_comm` metapackage as follows:

```
| /opt/ros/kinetic/share/ros_comm
```

To list the files inside the pack or stack, you will use the following command:

```
| $ rosfs turtlesim
```

The following is the output of the preceding command:

```
| cmake images srv package.xml msg
```

Changing the current working directory, essentially moving to a new folder, can be done with `roscd` as follows:

```
| $ roscd turtlesim  
| $ pwd
```

The new path will be as follows:

```
| /opt/ros/kinetic/share/turtlesim
```

Creating our own workspace

Before proceeding to further examples, we are going to create our own workspace. In this workspace, we will centralize all the code used throughout this section.

To see the workspace that ROS is using, use the following command:

```
| $ echo $ROS_PACKAGE_PATH
```

You will see output similar to the following:

```
| /opt/ros/kinetic/share:/opt/ros/kinetic/stacks
```

The folder that we are going to create is in `~/dev/catkin_ws/src/`. To add this folder, we use the following commands:

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

Once we've created the workspace folder, there are no packages inside-only a `CMakeList.txt` file. The next step is building the workspace. To do this, we use the following commands:

```
| $ cd ~/dev/catkin_ws  
| $ catkin_make
```

Now, if you type the `ls` command in the directory, you can see new folders created with the previous command. These are the `build` and `devel` folders.

To finish the configuration, use the following command:

```
| $ source devel/setup.bash
```

This step is only for reloading the `setup.bash` file. You will obtain the same result if you close and open a new shell. You should have this command at the end in your `~/.bashrc` file because we used it in Chapter 1, *Getting Started with ROS*; otherwise, you can add it using the following command:

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

Creating an ROS package and metapackage

As explained before, creating packages can be done manually, but to avoid the tedious work involved, we will use the `catkin_create_pkg` command-line tool.

We will create the new package in our recently initialized workspace using the following commands:

```
| $ cd ~/dev/catkin_ws/src  
| $ catkin_create_pkg chapter2_tutorials std_msgs roscpp
```

The format of this command includes the name of the package and the dependencies that will have the package, in our case, `std_msgs` and `roscpp`. This is shown in the following command:

```
| catkin_create_pkg [package_name] [dependency1] ... [dependencyN]
```

The following dependencies are included:

- `std_msgs`: This contains common message types representing primitive data types and other basic message constructs, such as `multiarray`.
- `roscpp`: This is a C++ implementation of ROS. It provides a client library that enables C++ programmers to quickly interface with ROS topics, services, and parameters.

If all the parameters are correct, the output will look as follows:

```
Created file chapter2_tutorials/package.xml  
Created file chapter2_tutorials/CMakeLists.txt  
Created folder chapter2_tutorials/include/chapter2_tutorials  
Created folder chapter2_tutorials/src  
Successfully created files in /home/aaronmr/dev/catkin_ws/src/chapter2_tutorials. Please adjust the values in package.xml.
```

As we saw earlier, you can use the `rospack`, `roscd`, and `ros1s` commands to retrieve information about the new package. The following are some of the operations which can be performed:

- `rospack profile`: This command informs you about the newly-added packages to the ROS system. It is useful after installing any new package.
- `rospack find chapter2_tutorials`: This command helps us find the path.
- `rospack depends chapter2_tutorials`: This command helps us see the dependencies.
- `ros1s chapter2_tutorials`: This command helps us see the content.
- `roscd chapter2_tutorials`: This command changes the actual path.

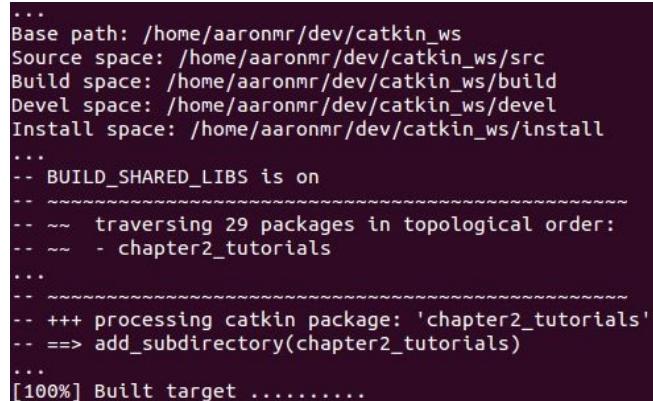
Building an ROS package

Once you have your package created and you have some code, it is necessary to build the package. When the package is built, the code contained in it is compiled; this includes not only the code added by the user, but also the code generated from the messages and services.

To build a package, we will use the `catkin_make` tool, as follows:

```
| $ cd ~/dev/catkin_ws/  
| $ catkin_make
```

In a few seconds, you will see something similar to the following screenshot:



```
...  
Base path: /home/aaronmr/dev/catkin_ws  
Source space: /home/aaronmr/dev/catkin_ws/src  
Build space: /home/aaronmr/dev/catkin_ws/build  
Devel space: /home/aaronmr/dev/catkin_ws/devel  
Install space: /home/aaronmr/dev/catkin_ws/install  
...  
-- BUILD_SHARED_LIBS is on  
-- ~~~~~  
-- ~~ traversing 29 packages in topological order:  
-- ~~ - chapter2_tutorials  
...  
-- ~~~~~  
-- +++ processing catkin package: 'chapter2_tutorials'  
-- ==> add_subdirectory(chapter2_tutorials)  
...  
[100%] Built target .....
```

If you don't encounter any failures, the package is compiled.

Remember that you should run the `catkin_make` command in the `workspace` folder. If you try to do it in any other folder, the command will fail. An example of this is provided in the following command lines:

```
| $ roscd chapter2_tutorials/  
| $ catkin_make
```

When you are in the `chapter2_tutorials` folder and try to build the package using `catkin_make`, you will get the following error:

```
| The specified base path "/home/your_user/dev/catkin_ws/src/chapter2_tutorials" contains a CMakeLists.txt but  
| "catkin_make" must be invoked in the root of workspace
```

If you execute `catkin_make` in the `catkin_ws` folder, you will obtain a good compilation. Finally, compiling a single package using `catkin_make` can be done using the following command:

```
| $ catkin_make --pkg <package name>
```

Playing with ROS nodes

As we explained in the *Nodes and nodelets* section, nodes are executable programs and, once built, these executables can be found in the `devel` space. To practise and learn about nodes, we are going to use a typical package called `turtlesim`.

If you have installed the desktop installation, you will have the `turtlesim` package preinstalled; if not, install it with the following command:

```
| $ sudo apt-get install ros-kinetic-ros-tutorials
```

Before starting this tutorial, it is important that you open a terminal and start `roscore` using the following command:

```
| $ roscore
```

To get information on nodes, we have the `rosnode` tool. To see what parameters are provided, type the following command:

```
| $ rosnode
```

You will obtain a list of accepted parameters, as shown in the following screenshot:

```
rosnode is a command-line tool for printing information about ROS Nodes.

Commands:
  rosnode ping    test connectivity to node
  rosnode list    list active nodes
  rosnode info    print information about node
  rosnode machine list nodes running on a particular machine or list machines
  rosnode kill    kill a running node
  rosnode cleanup purge registration information of unreachable nodes

Type rosnode <command> -h for more detailed usage, e.g. 'rosnode ping -h'
```

If you want a more detailed explanation of the use of these parameters, use the following command:

```
| $ rosnode <param> -h
```

Now that `roscore` is running, we are going to get information about the nodes that are running, using the following command:

```
| $ rosnode list
```

You will see that the only node running is `/rosout`. This is normal, as this node is usually launched with `roscore`.

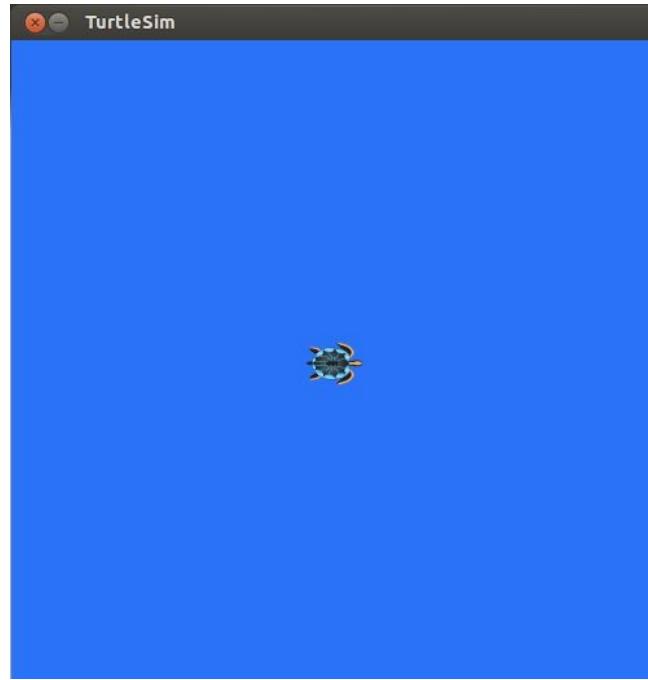
We can get all the information about this node using the parameters provided by the `rosnode` tool. Try to use the following commands for more information:

```
$ rosnode info
$ rosnode ping
$ rosnode machine
$ rosnode kill
$ rosnode cleanup
```

Now we are going to start a new node with `rosrun` using the following command:

```
| $ rosrun turtlesim turtlesim_node
```

We will then see a new window appear with a little turtle in the middle, as shown in the following screenshot:



If we see the node list now, we will see a new node with the name `/turtlesim`. You can see information about the node using `rosnode info nameNode`.

You can see a lot of information that can be used to debug your programs, using the following command:

```
| $ rosnode info /turtlesim
```

The preceding command line prints the following information:

```
-----  
Node [/turtlesim]  
Publications:  
  * /turtle1/color_sensor [turtlesim/Color]  
  * /rosout [rosgraph_msgs/Log]  
  * /turtle1/pose [turtlesim/Pose]  
  
Subscriptions:  
  * /turtle1/cmd_vel [geometry_msgs/Twist]  
  
Services:  
  * /turtle1/teleport_absolute  
  * /turtlesim/get_loggers  
  * /turtlesim/set_logger_level  
  * /reset  
  * /spawn  
  * /clear  
  * /turtle1/set_pen  
  * /turtle1/teleport_relative  
  * /kill  
  
contacting node http://daneel:38674/ ...  
Pid: 3881  
Connections:  
  * topic: /rosout  
    * to: /rosout  
    * direction: outbound  
    * transport: TCPROS  
  * topic: /turtle1/cmd_vel  
    * to: /teleop_turtle (http://daneel:44645/)  
    * direction: inbound  
    * transport: TCPROS
```

In the information, we can see the Publications (*topics*), Subscriptions (*topics*), and services (*srv*) that the node has and the unique name of each.

Now, let's see how you interact with the node using topics and services.

Learning how to interact with topics

To interact and get information about topics, we have the `rostopic` tool. This tool accepts the following parameters:

- `rostopic bw TOPIC`: This displays the bandwidth used by topics
- `rostopic echo TOPIC`: This prints messages to the screen
- `rostopic find TOPIC`: This finds topics by their type
- `rostopic hz TOPIC`: This displays the publishing rate of topics
- `rostopic info TOPIC`: This prints information about active topics
- `rostopic list`: This lists the active topics
- `rostopic pubs TOPIC`: This publishes data to the topic
- `rostopic type TOPIC`: This prints the topic type

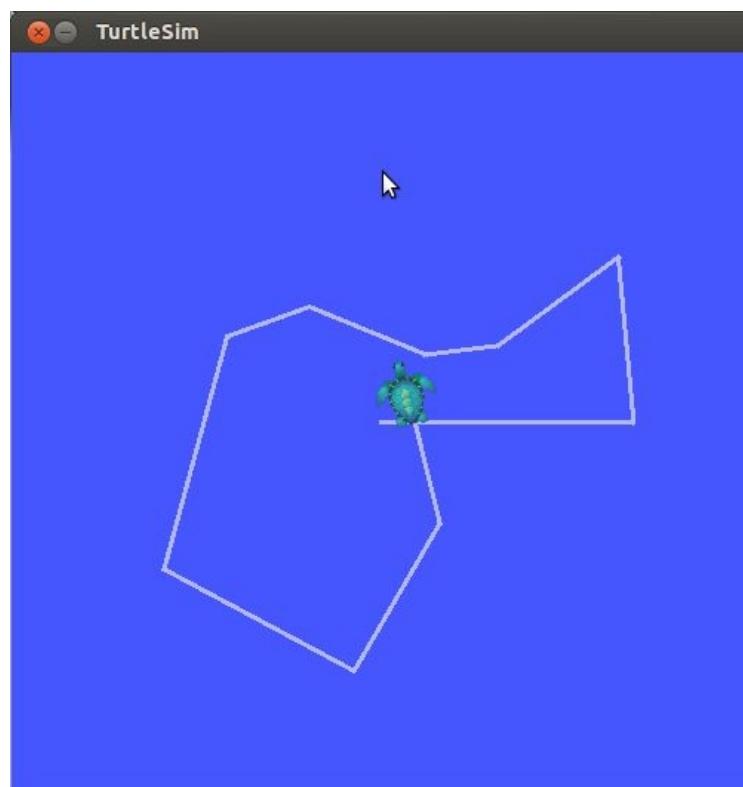
If you want to see more information on these parameters, use `-h`, as follows:

```
| $ rostopic bw -h
```

With the `pub` parameter, we can publish topics that can be subscribed to by any node. We only need to publish the topic with the correct name. We will do this test later; we are now going to use a node that will do this work for us:

```
| $ rosrun turtlesim turtle_teleop_key
```

With this node, we can move the turtle using the arrow keys, as illustrated in the following screenshot:



Why does the turtle move when `turtle_teleop_key` is executed?

Checking the information provided by `rosnode` about the `/teleop_turtle` and `/turtlesim` nodes, we can see that there exists a topic called `/turtle1/cmd_vel` [geometry_msgs/Twist] in the Publications section of the node `/teleop_turtle`, and in the Subscriptions section of the `/turtlesim` node, there is `/turtle1/cmd_vel` [geometry_msgs/Twist]:

```
| $ rosnode info /teleop_turtle
```

```
-----  
Node [/teleop_turtle]  
Publications:  
* /turtle1/cmd_vel [geometry_msgs/Twist]  
* /rosout [rosgraph_msgs/Log]  
  
Subscriptions: None  
  
Services:  
* /teleop_turtle/get_loggers  
* /teleop_turtle/set_logger_level  
  
contacting node http://daneel:44645/ ...  
Pid: 4156  
Connections:  
* topic: /rosout  
  * to: /rosout  
  * direction: outbound  
  * transport: TCPROS  
* topic: /turtle1/cmd_vel  
  * to: /turtlesim  
  * direction: outbound  
  * transport: TCPROS
```

This means that the first node is publishing a topic that the second node can subscribe to. You can see the topic list using the following command line:

```
| $ rostopic list
```

The output will be as follows:

```
/rosout  
/rosout_agg  
/turtle1/colour_sensor  
/turtle1/cmd_vel  
/turtle1/pose
```

With the `echo` parameter, you can see the information sent by the node. Run the following command line and use the arrow keys to see the data that is being sent:

```
| $ rostopic echo /turtle1/cmd_vel
```

You will see something similar to the following output:

```
---  
linear:  
x: 0.0  
y: 0.0  
z: 0.0  
angular:  
x: 0.0  
y: 0.0  
z: 2.0  
---
```

You can see the type of message sent by the topic using the following command line:

```
| $ rostopic type /turtle1/cmd_vel
```

You will see something similar to the following output:

```
| Geometry_msgs/Twist
```

If you want to see the message fields, you can do it with the following command:

```
| $ rosmsg show geometry_msgs/Twist
```

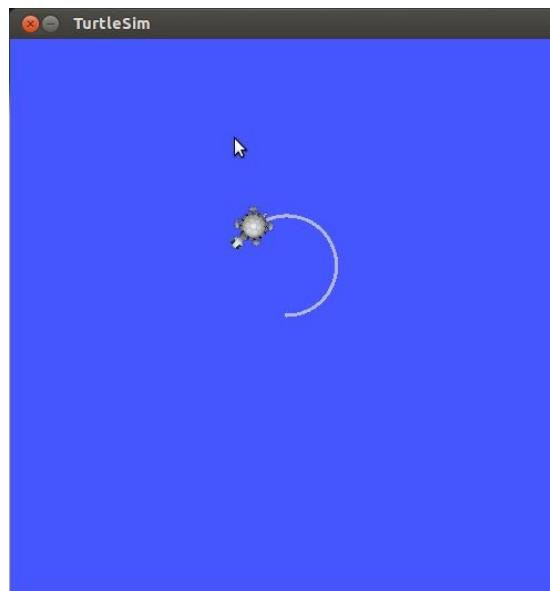
You will see something similar to the following output:

```
geometry_msgs/Vector3 linear
  float64 x
  float64 y
  float64 z
geometry_msgs/Vector3 angular
  float64 x
  float64 y
  float64 z
```

These tools are useful because, with this information, we can publish topics using the `rostopic pub [topic] [msg_type] [args]` command:

```
$ rostopic pub /turtle1/cmd_vel geometry_msgs/Twist -r 1 -- "linear:
x: 1.0
y: 0.0
z: 0.0
angular:
x: 0.0
y: 0.0
z: 1.0"
```

You will see the turtle doing a curve, as shown in the following screenshot:



Learning how to use services

Services are another way through which nodes can communicate with each other. Services allow nodes to send a request and receive a response.

The tool that we are going to use to interact with services is called `rosservice`. The accepted parameters for this command are as follows:

- `rosservice args /service`: This prints the service arguments
- `rosservice call /service`: This calls the service with the arguments provided
- `rosservice find msg-type`: This finds services by their service type
- `rosservice info /service`: This prints information about the service
- `rosservice list`: This lists the active services
- `rosservice type /service`: This prints the service type
- `rosservice uri /service`: This prints the ROSRPC URI service

We are going to list the services available for the `turtlesim` node using the following command, so if it is not working, run `roscore` and run the `turtlesim` node:

```
| $ rosservice list
```

You will obtain the following output:

```
/clear
/kill
/reset
/rosout/get_loggers
/rosout/set_logger_level
/spawn
/teleop_turtle/get_loggers
/teleop_turtle/set_logger_level
/turtle1/set_pen
/turtle1/teleport_absolute
/turtle1/teleport_relative
/turtlesim/get_loggers
/turtlesim/set_logger_level
```

If you want to see the type of any service, for example, the `/clear` service, use the following command:

```
| $ rosservice type /clear
```

You will see something similar to the following output:

```
| std_srvs/Empty
```

To invoke a service, you will use `rosservice call [service] [args]`. If you want to invoke the `/clear` service, use the following command:

```
| $ rosservice call /clear
```

In the `turtlesim` window, you will now see that the lines created by the movements of the turtle will be deleted.

Now we are going to try another service, for example, the `/spawn` service. This service will create another turtle in another location with a different orientation.

To start with, we are going to see the following type of message:

```
| $ rosservice type /spawn | rossrv show
```

You will see something similar to the following output:

```
float32 x
float32 y
float32 theta
string name
...
string name
```

The preceding command is the same as the following commands. If you want to know why these lines are the same, search in Google about *piping Linux*:

```
| $ rosservice type /spawn
```

You will see something similar to the following output:

```
| turtlesim/Spawn
```

Type in the following command:

```
| $ rossrv show turtlesim/Spawn
```

You will see something similar to the following output:

```
float32 x
float32 y
float32 theta
string name
...
string name
```

With these fields, we know how to invoke the service. We need the position of `x` and `y`, the orientation (`theta`), and the name of the new turtle:

```
| $ rosservice call /spawn 3 3 0.2 "new_turtle"
```

We then obtain the following result:



Using Parameter Server

Parameter Server is used to store data that is accessible to all nodes. ROS has a tool called `rosparam` to manage Parameter Server. The accepted parameters are as follows:

- `rosparam set parameter value`: This sets the parameter
- `rosparam get parameter`: This gets the parameter
- `rosparam load file`: This loads parameters from the file
- `rosparam dump file`: This dumps parameters to the file
- `rosparam delete parameter`: This deletes the parameter
- `rosparam list`: This lists the parameter names

For example, we can see the parameters in the server that are used by all nodes:

```
| $ rosparam list
```

We obtain the following output:

```
/background_b  
/background_g  
/background_r  
/rosdistro  
/roslaunch/uris/host_aaronmr_laptop_60878  
/rosversion  
/run_id
```

The background parameters are of the `turtlesim` node. These parameters change the color of the windows that are initially blue. If you want to read a value, you will use the `get` parameter:

```
| $ rosparam get /background_b
```

To set a new value, you will use the `set` parameter:

```
| $ rosparam set /background_b 100
```

Another important feature of `rosparam` is the `dump` parameter. With this parameter, you can save or load the contents of Parameter Server.

To save Parameter Server, use `rosparam dump [file_name]` as follows:

```
| $ rosparam dump save.yaml
```

To load a file with new data for Parameter Server, use `rosparam load [file_name] [namespace]` as follows:

```
| $ rosparam load load.yaml namespace
```

Creating nodes

In this section, we are going to learn how to create two nodes: one to publish data and the other to receive this data. This is the basic way of communicating between two nodes, that is, to handle data and do something with this data.

Navigate to the `chapter2_tutorials/src/` folder using the following command:

```
| $ rosdep chapter2_tutorials/src/
```

Create two files with the names `example1_a.cpp` and `example1_b.cpp`. The `example1_a.cpp` file will send the data with the node name, and the `example1_b.cpp` file will show the data in the shell. Copy the following code inside the `example1_a.cpp` file or download it from the repository:

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

int main(int argc, char **argv)
{
    ros::init(argc, argv, "example1_a");
    ros::NodeHandle n;
    ros::Publisher chatter_pub =
        n.advertise<std_msgs::String>("message", 1000);
    ros::Rate loop_rate(10);
    while (ros::ok())
    {
        std_msgs::String msg;
        std::stringstream ss;
        ss << " I am the example1_a node ";
        msg.data = ss.str();
        //ROS_INFO("%s", msg.data.c_str());
        chatter_pub.publish(msg);
        ros::spinOnce();
        loop_rate.sleep();
    }
    return 0;
}
```

Here is a further explanation of the preceding code. The headers to be included are `ros/ros.h`, `std_msgs/String.h`, and `sstream`. Here, `ros/ros.h` includes all the files necessary for using the node with ROS, and `std_msgs/String.h` includes the header that denotes the type of message that we are going to use:

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

At this point, we initialize the node and set the name; remember that the name must be unique:

```
| ros::init(argc, argv, "example1_a");
```

This is the handler of our process; it allows us to interact with the environment:

```
| ros::NodeHandle n;
```

We instantiate a publisher and tell the master the name of the topic and the type. The name is `message`, and the second parameter is the buffer size. If the topic is publishing data quickly, the buffer will keep at least

1000 messages:

```
| ros::Publisher chatter_pub =  
| n.advertise<std_msgs::String>("message", 1000);
```

The next step is to set the data sending frequency, which in this case is 10 Hz:

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

The `ros::ok()` line stops the node if *Ctrl + C* is pressed or if ROS stops all the nodes:

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

In this part, we create a variable for the message with the correct type to send the data:

```
| std_msgs::String msg;  
| std::stringstream ss;  
| ss<< " I am the example1_a node ";  
| msg.data = ss.str();
```

We continue by sending the message, in this case, the semantic is to publish a message, using the previously defined publisher:

```
| chatter_pub.publish(msg);
```

The `spinOnce` function takes care of handling all of ROS's internal events and actions, such as reading from subscribed topics; `spinOnce` performs one iteration in the main loop of ROS in order to allow the user to perform actions between iterations, in contrast with the `spin` function, which runs the main loop without interruption:

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

Finally, we sleep for the required time to get a 10 Hz frequency:

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

Now we will create the other node. Copy the following code inside the `example1_b.cpp` file or download it from the repository:

```
#include "ros/ros.h"  
#include "std_msgs/String.h"  
  
void chatterCallback(const std_msgs::String::ConstPtr& msg)  
{  
    ROS_INFO("I heard: [%s]", msg->data.c_str());  
}  
  
int main(int argc, char **argv)  
{  
    ros::init(argc, argv, "example1_b");  
    ros::NodeHandle n;  
    ros::Subscriber sub = n.subscribe("message", 1000,  
        chatterCallback);  
    ros::spin();  
    return 0;  
}
```

Let's explain the code. Include the headers and the type of message to use for the topic:

```
| #include "ros/ros.h"  
| #include "std_msgs/String.h"
```

The following type of function is a `callback` and happens in response to an action, which in this case is the reception of a `String` message. This function allows us to do something with the data; in this case, we display it in the terminal:

```
| void messageCallback(const std_msgs::String::ConstPtr& msg)  
{  
    ROS_INFO("I heard: [%s]", msg->data.c_str());  
}
```

We create a subscriber and start to listen to the topic with the name `message`. The buffer will be of `1000`, and the function to handle the message will be `messageCallback`:

```
| ros::Subscriber sub = n.subscribe("message", 1000,  
    messageCallback);
```

The `ros::spin()` line is the main loop where the node starts to read the topic and when a message arrives, `messageCallback` is called. When the user presses `Ctrl + C`, the node exits the loop and ends:

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

Building the node

As we are using the `chapter2_tutorials` package, we are going to edit the `CMakeLists.txt` file. You can use your favorite editor or the `rosed` tool. This will open the file with the Vim editor:

```
| $ rosed chapter2_tutorials CMakeLists.txt
```

At the end of the file, we will copy the following lines:

```
include_directories(  
  include  
  ${catkin_INCLUDE_DIRS}  
)  
  
add_executable(example1_a src/example1_a.cpp)  
add_executable(example1_b src/example1_b.cpp)  
  
add_dependencies(example1_a  
  chapter2_tutorials_generate_messages_cpp)  
add_dependencies(example1_b  
  chapter2_tutorials_generate_messages_cpp)  
  
target_link_libraries(example1_a ${catkin_LIBRARIES})  
target_link_libraries(example1_b ${catkin_LIBRARIES})
```

Now, to build the package and compile all the nodes, use the `catkin_make` tool as follows:

```
| $ cd ~/dev/catkin_ws/  
| $ catkin_make --pkg chapter2_tutorials
```

If ROS is not running on your computer, you will have to use the following command:

```
| $ roscore
```

You can check whether ROS is running using the `rosvnode list` command as follows:

```
| $ rosvnnode list
```

Now run both nodes in different shells:

```
| $ rosrun chapter2_tutorials example1_a  
| $ rosrun chapter2_tutorials example1_b
```

If you check the shell where the `example1_b` node is running, you will see something similar to the following screenshot:

```
...  
[ INFO] [1403252419.452448698]: I heard: [ I am the example1_a node ]  
[ INFO] [1403252419.552163326]: I heard: [ I am the example1_a node ]  
[ INFO] [1403252419.653701929]: I heard: [ I am the example1_a node ]  
[ INFO] [1403252419.752261663]: I heard: [ I am the example1_a node ]  
[ INFO] [1403252419.854459847]: I heard: [ I am the example1_a node ]  
...
```

Everything that is happening can be viewed in the following diagram. You can see that the `example1_a` node is publishing the `message` topic, and the `example1_b` node is subscribing to the topic:



You can use `rosnode` and `rostopic` to debug and see what the nodes are doing. Try the following commands:

```
$ rosnode list
$ rosnode info /example1_a
$ rosnode info /example1_b
$ rostopic list
$ rostopic info /message
$ rostopic type /message
$ rostopic bw /message
```

Creating msg and srv files

In this section, we are going to learn how to create `msg` and `srv` files for use in our nodes. They are files where we put a specification about the type of data to be transmitted and the values of this data. ROS will use these files to create the necessary code for us to implement the `msg` and `srv` files to be used in our nodes.

Let's start with the `msg` file first.

In the example used in the *Building the node* section, we created two nodes with a standard type message. Now we are going to learn how to create custom messages with the tools that ROS has.

First, create a new `msg` folder in our `chapter2_tutorials` package; create a new `chapter2_msg1.msg` file and add the following lines:

```
| int32 A  
| int32 B  
| int32 C
```

Now, edit `package.xml` and remove `<!-- -->` from the `<build_depend>message_generation</build_depend>` and `<run_depend>message_runtime</run_depend>` lines.

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

```
| find_package(catkin REQUIRED COMPONENTS  
|   roscpp  
|   std_msgs  
|   message_generation  
)
```

Find the next lines, uncomment and add the name of the new message as follows:

```
| ## Generate messages in the 'msg' folder  
| add_message_files(  
|   FILES  
|   chapter2_msg1.msg  
)  
  
| ## Generate added messages and services with any dependencies  
| listed here  
| generate_messages(  
|   DEPENDENCIES  
|   std_msgs  
)
```

And now you can compile using the following lines:

```
| $ cd ~/dev/catkin_ws/  
| $ catkin_make
```

To check whether all is OK, you can use the `rosmsg` command:

```
| $ rosmsg show chapter2_tutorials/chapter2_msg1
```

If you see the same content as that of the `chapter2_msg1.msg` file, all is OK.

Now we are going to create a `srv` file. Create a new folder in the `chapter2_tutorials` folder with the name `srv`, create a new `chapter2_srv1.srv` file and add the following lines:

```
| int32 A  
| int32 B  
| int32 C  
| ---  
+ int32 sum
```

To compile the new `msg` and `srv` files, you have to uncomment the following lines in the `package.xml` and `CMakeLists.txt` files. These lines permit the configuration of the messages and services and tell ROS how and what to build.

First of all, open the `package.xml` folder from your `chapter2_tutorials` package as follows:

```
| $ roscd chapter2_tutorials package.xml
```

Search for the following lines and uncomment them:

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

Open `CMakeLists.txt` using the following command:

```
| $ roscd chapter2_tutorials CMakeLists.txt
```

Find the following lines, uncomment them and complete them with the correct data:

```
| catkin_package(  
|   CATKIN_DEPENDS message_runtime  
)
```

To generate messages, you need to add the `message_generation` line in the `find_package` section:

```
| find_package(catkin REQUIRED COMPONENTS  
|   roscpp  
|   std_msgs  
|   message_generation  
)
```

Add the names of the message and service files in the `add_message_files` section, as follows:

```
| ## Generate messages in the 'msg' folder  
| add_message_files(  
|   FILES  
|     chapter2_msg1.msg  
)  
  
| ## Generate services in the 'srv' folder  
| add_service_files(  
|   FILES  
|     chapter2_srv1.srv  
)
```

Uncomment the `generate_messages` section to make sure that the generation of messages and services can be done:

```
| ## Generate added messages and services with any dependencies  
| listed here
```

```
        generate_messages(  
            DEPENDENCIES  
            std_msgs  
)
```

You can test whether all is OK using the `rossrv` tool as follows:

```
|   $ rossrv show chapter2_tutorials/chapter2_srv1
```

If you see the same content as that of the `chapter2_srv1.srv` file, all is OK.

Using the new srv and msg files

First, we are going to learn how to create a service and how to use it in ROS. Our service will calculate the sum of three numbers. We need two nodes: a server and a client.

In the `chapter2_tutorials` package, create two new nodes with the following names: `example2_a.cpp` and `example2_b.cpp`. Remember to put the files in the `src` folder.

In the first file, `example2_a.cpp`, add the following code:

```
#include "ros/ros.h"
#include "chapter2_tutorials/chapter2_srv1.h"

bool add(chapter2_tutorials::chapter2_srv1::Request &req,
          chapter2_tutorials::chapter2_srv1::Response &res)
{
    res.sum = req.A + req.B + req.C;
    ROS_INFO("request: A=%ld, B=%ld C=%ld", (int)req.A, (int)req.B,
             (int)req.C);
    ROS_INFO("sending back response: [%ld]", (int)res.sum);
    return true;
}

int main(int argc, char **argv)
{
    ros::init(argc, argv, "add_3_ints_server");
    ros::NodeHandle n;
    ros::ServiceServer service = n.advertiseService("add_3_ints",
                                                    add);
    ROS_INFO("Ready to add 3 ints.");
    ros::spin();
    return 0;
}
```

Let's explain the code. These lines include the necessary headers and the `srv` file that we created:

```
#include "ros/ros.h"
#include "chapter2_tutorials/chapter2_srv1.h"
```

This function will add three variables and send the result to the other node:

```
bool add(chapter2_tutorials::chapter2_srv1::Request &req,
          chapter2_tutorials::chapter2_srv1::Response &res)
```

Here, the service is created and advertised over ROS:

```
ros::ServiceServer service = n.advertiseService("add_3_ints",
                                                add);
```

In the second file, `example2_b.cpp`, add this code:

```
#include "ros/ros.h"
#include "chapter2_tutorials/chapter2_srv1.h"
#include <cstdlib>

int main(int argc, char **argv)
{
    ros::init(argc, argv, "add_3_ints_client");
    if (argc != 4)
    {
```

```

ROS_INFO("usage: add_3_ints_client A B C ");
return 1;
}

ros::NodeHandle n;
ros::ServiceClient client =
  n.serviceClient<chapter2_tutorials::chapter2_srv1>("add_3_ints");
chapter2_tutorials::chapter2_srv1 srv;
srv.request.A = atol(argv[1]);
srv.request.B = atol(argv[2]);
srv.request.C = atol(argv[3]);
if (client.call(srv))
{
  ROS_INFO("Sum: %ld", (long int)srv.response.sum);
}
else
{
  ROS_ERROR("Failed to call service add_3_ints");
  return 1;
}
return 0;
}

```

As usual, let's explain the code. Create a client for the service with the name `add_3_ints`:

```

ros::ServiceClient client =
n.serviceClient<chapter2_tutorials::chapter2_srv1>("add_3_ints");

```

Here, we create an instance of our `srv` request type and fill all the values to be sent. If you remember, the message has three fields:

```

chapter2_tutorials::chapter2_srv1 srv;
srv.request.A = atol(argv[1]);
srv.request.B = atol(argv[2]);
srv.request.C = atol(argv[3]);

```

With this line, the service is called and the data is sent. If the call succeeds, `call()` will return `true`, and if not, `call()` will return `false`:

```

if (client.call(srv))

```

To build the new nodes, edit `CMakeList.txt` and add the following lines:

```

add_executable(example2_a src/example2_a.cpp)
add_executable(example2_b src/example2_b.cpp)

add_dependencies(example2_a
  chapter2_tutorials_generate_messages_cpp)add_dependencies(example2_b
  chapter2_tutorials_generate_messages_cpp)
target_link_libraries(example2_a ${catkin_LIBRARIES})
target_link_libraries(example2_b ${catkin_LIBRARIES})

```

Now execute the following command:

```

$ cd ~/dev/catkin_ws
$ catkin_make

```

To start the nodes, execute the following command lines:

```

$ rosrun chapter2_tutorials example2_a
$ rosrun chapter2_tutorials example2_b 1 2 3

```

You should see something similar to this output:

```

| Node example2_a

```

```
[ INFO] [1355256113.014539262]: Ready to add 3 ints.
[ INFO] [1355256115.792442091]: request: A=1, B=2 C=3
[ INFO] [1355256115.792607196]: sending back response: [6]
Node example2_b
[ INFO] [1355256115.794134975]: Sum: 6
```

Now we are going to create nodes with our custom `msg` file. The example is the same, that is, `example1_a.cpp` and `example1_b.cpp`, but with the new message, `chapter2_msg1.msg`.

The following code snippet is present in the `example3_a.cpp` file:

```
#include "ros/ros.h"
#include "chapter2_tutorials/chapter2_msg1.h"
#include <iostream>

int main(int argc, char **argv)
{
    ros::init(argc, argv, "example3_a");
    ros::NodeHandle n;
    ros::Publisher pub =
        n.advertise<chapter2_tutorials::chapter2_msg1>("message", 1000);
    ros::Rate loop_rate(10);
    while (ros::ok())
    {
        chapter2_tutorials::chapter2_msg1 msg;
        msg.A = 1;
        msg.B = 2;
        msg.C = 3;
        pub.publish(msg);
        ros::spinOnce();
        loop_rate.sleep();
    }
    return 0;
}
```

The following code snippet is present in the `example3_b.cpp` file:

```
#include "ros/ros.h"
#include "chapter2_tutorials/chapter2_msg1.h"

void messageCallback(const
    chapter2_tutorials::chapter2_msg1::ConstPtr& msg)
{
    ROS_INFO("I heard: [%d] [%d] [%d]", msg->A, msg->B, msg->C);
}

int main(int argc, char **argv)
{
    ros::init(argc, argv, "example3_b");
    ros::NodeHandle n;
    ros::Subscriber sub = n.subscribe("message", 1000,
        messageCallback);
    ros::spin();
    return 0;
}
```

If we run both nodes now, we will see something similar to the following output:

```
...
[ INFO] [1355270835.920368620]: I heard: [1] [2] [3]
[ INFO] [1355270836.020326372]: I heard: [1] [2] [3]
[ INFO] [1355270836.120367449]: I heard: [1] [2] [3]
[ INFO] [1355270836.220266466]: I heard: [1] [2] [3]
...
```

The launch file

The `launch` file is a useful feature in ROS for launching more than one node. In these sections, we have created nodes and we have been executing them in different shells. Imagine working with 20 nodes and the nightmare of executing each one in a shell!

With the `launch` file, we can do it in the same shell by launching a configuration file with the extension `.launch`.

To practise using this utility, we are going to create a new folder in our package as follows:

```
$ rosed chapter2_tutorials/  
$ mkdir launch  
$ cd launch  
$ vim chapter2.launch
```

Now put the following code inside the `chapter2.launch` file:

```
<?xml version="1.0"?>  
<launch>  
  <node name ="example1_a" pkg="chapter2_tutorials"  
        type="example1_a"/>  <node name ="example1_b" pkg="chapter2_tutorials"  
        type="example1_b"/>  
</launch>
```

This file is simple, although you can write a very complex file if you want, for example, to control a complete robot, such as PR2 or Robonaut. Both are real robots and they are simulated in ROS.

The file has a `launch` tag; inside this tag, you can see the `node` tag. The `node` tag is used to launch a node from a package, for example, the `example1_a` node from the `chapter2_tutorials` package.

This launch file will execute two nodes-the first two examples of this chapter. If you remember, the `example1_a` node sends a message to the `example1_b` node.

To launch the file, you can use the following command:

```
$ rosrun chapter2_tutorials chapter2.launch
```

You will see something similar to the following screenshot on your screen:

```

started roslaunch server http://127.0.0.1:40930/
SUMMARY
=====
PARAMETERS
  * /rosdistro
  * /rosversion

NODES
/
  example1_a (chapter2_tutorials/example1_a)
  example1_b (chapter2_tutorials/example1_b)

auto-starting new master
process[master]: started with pid [19889]
ROS_MASTER_URI=http://localhost:11311

setting /run_id to b334800a-f940-11e3-989f-080027b05884
process[rosout-1]: started with pid [19902]
started core service [/rosout]
process[example1_a-2]: started with pid [19914]
process[example1_b-3]: started with pid [19925]

```

The running nodes are listed in the screenshot. You can also see the running nodes using the following command:

```
| $ rosnode list
```

You will see the three nodes listed as follows:

```
/example1_a
/example1_b
/rosout
```

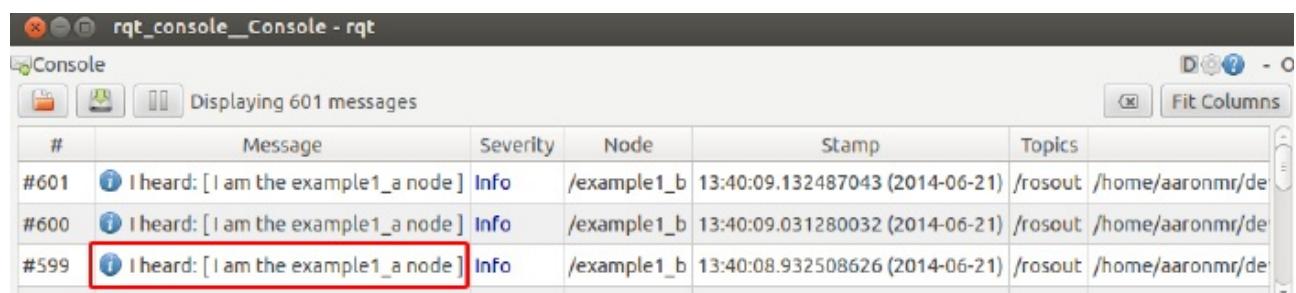
When you launch a launch file, it is not necessary to execute it before the `roscore` command; `roslaunch` does it for us.

Remember that the `example1_b` node prints on the screen the message received from the other node. If you take a look, you won't see anything. This is because `example1_b` prints the message using `ROS_INFO`, and when you run only a node in a shell, you can see it, but when you run a launch file, you can't.

Now, to see the message printed on the screen, you can use the `rqt_console` utility. You will learn more about this utility in the following chapters. Now run the following command:

```
| $ rqt_console
```

You will see the message sent by `example1_b`, as shown in the following screenshot:



On the line, you can see the message, the node that has sent it, and the path of the source file.

Dynamic parameters

Another utility in ROS is the **Dynamic Reconfigure** utility. Normally, when you are programming a new node, you initialize the variables with data that can only be changed within the node. If you want to change these values dynamically from outside the node, you can use Parameter Server, services, or topics. If you are working in a PID node to control a motor, for example, you should use the Dynamic Reconfigure utility.

In this section, you will learn how to configure a basic node with this feature. Add the necessary lines in the `CMakeLists.txt` and `package.xml` files.

To use Dynamic Reconfigure, you should write a configuration file and save it in the `cfg` folder in your package. Create the folder and a new file as follows:

```
$ rosdep chapter2_tutorials
$ mkdir cfg
$ vim chapter2.cfg
```

Write the following code in the `chapter2.cfg` file:

```
#!/usr/bin/env python
PACKAGE = "chapter2_tutorials"

from dynamic_reconfigure.parameter_generator_catkin import *

gen = ParameterGenerator()

gen.add("double_param", double_t, 0, "A double parameter", .1, 0,
1)gen.add("str_param", str_t, 0, "A string parameter",
"Chapter2_dynamic_reconfigure")
gen.add("int_param", int_t, 0, "An Integer parameter", 1, 0, 100)
gen.add("bool_param", bool_t, 0, "A Boolean parameter", True)

size_enum = gen.enum([ gen.const("Low", int_t, 0, "Low is 0"),
gen.const("Medium", int_t, 1, "Medium is 1"),
gen.const("High", int_t, 2, "High is 2")],
"Select from the list")

gen.add("size", int_t, 0, "Select from the list", 1, 0, 3,
edit_method=size_enum)

exit(gen.generate(PACKAGE, "chapter2_tutorials", "chapter2_"))
```

Let's explain the code. These lines initialize ROS and import the parameter generator:

```
#!/usr/bin/env python
PACKAGE = "chapter2_tutorials"

from dynamic_reconfigure.parameter_generator_catkin import *
```

The following line initializes the parameter generator, and thanks to that, we can start to add parameters in the following lines:

```
gen = ParameterGenerator()
gen.add("double_param", double_t, 0, "A double parameter", .1, 0,
1)gen.add("str_param", str_t, 0, "A string parameter",
"Chapter2_dynamic_reconfigure")
gen.add("int_param", int_t, 0, "An Integer parameter", 1, 0, 100)
gen.add("bool_param", bool_t, 0, "A Boolean parameter", True)
```

These lines add different parameter types and set the default values, description, range, and so on. The parameter has the following arguments:

```
| gen.add(name, type, level, description, default, min, max)
```

- `name`: This is the name of the parameter
- `type`: This is the type of the value stored
- `level`: This is a bitmask that is passed to the callback
- `description`: This is a short description of the parameter
- `default`: This is the default value when the node starts
- `min`: This is the minimum value for the parameter
- `max`: This is the maximum value for the parameter

The names of the parameters must be unique, and the values have to be in the range and have `min` and `max` values:

```
| exit(gen.generate(PACKAGE, "chapter2_tutorials", "chapter2_"))
```

The last line generates the necessary files and exits the program. Notice that the `.cfg` file was written in Python. This section is for C++ snippets, but we will sometimes use Python snippets.

It is necessary to change the permissions for the file because the file will be executed by ROS. To make the file executable and runnable by any user, we will use the `chmod` command with the `a+x` parameter as follows:

```
| $ chmod a+x cfg/chapter2.cfg
```

Open `CMakeList.txt` and add the following lines:

```
find_package(catkin REQUIRED COMPONENTS
roscpp
  std_msgs
  message_generation
  dynamic_reconfigure
)
generate_dynamic_reconfigure_options(
  cfg/chapter2.cfg
)
add_dependencies(example4 chapter2_tutorials_gencfg)
```

Now we are going to write our new node with Dynamic Reconfigure support. Create a new file in your `src` folder as follows:

```
| $ roscd chapter2_tutorials
| $ vim src/example4.cpp
```

Write the following code snippet in the file:

```
#include <ros/ros.h>
#include <dynamic_reconfigure/server.h>
#include <chapter2_tutorials/chapter2Config.h>

void callback(chapter2_tutorials::chapter2Config &config, uint32_t
level) {
  ROS_INFO("Reconfigure Request: %d %f %s %s %d",
  config.level, config.servo1, config.servo2, config.servo3,
  config.servo4, config.servo5);
```

```

    config.int_param,
    config.double_param,
    config.str_param.c_str(),
    config.bool_param?"True":"False",
    config.size);
}

int main(int argc, char **argv) {
    ros::init(argc, argv, "example4_dynamic_reconfigure");

    dynamic_reconfigure::Server<chapter2_tutorials::chapter2Config>
        server; dynamic_reconfigure::Server<chapter2_tutorials::chapter2Config>::
        CallbackType f;

    f = boost::bind(&callback, _1, _2);
    server.setCallback(f);

    ros::spin();
    return 0;
}

```

Let's explain the code and note the important lines. As usual, these lines include the headers for ROS, Parameter Server, and our `config` file created earlier:

```

#include <ros/ros.h>
#include <dynamic_reconfigure/server.h>
#include <chapter2_tutorials/chapter2Config.h>

```

The `callback` function will print the new values for the parameters. The way to access the parameters is, for example, `config.int_param`. The name of the parameter must be the same as the one that you configured in the `example2.cfg` file:

```

void callback(chapter2_tutorials::chapter2Config &config, uint32_t
level) {
    ROS_INFO("Reconfigure Request: %d %f %s %s %d",
            config.int_param,
            config.double_param,
            config.str_param.c_str(),
            config.bool_param?"True":"False",
            config.size);
}

```

To continue, the server is initialized in the line where we pass the `chapter2_config` configuration file:

```

dynamic_reconfigure::Server<chapter2_tutorials::chapter2Config>
server; dynamic_reconfigure::Server<chapter2_tutorials::chapter2Config>::
CallbackType f;

f = boost::bind(&callback, _1, _2);
server.setCallback(f);

```

Now we send the `callback` function to the server. When the server gets a reconfiguration request, it will call the `callback` function.

Once we are done with the explanation, we need to add lines to the `CMakeLists.txt` file as follows:

```

add_executable(example4 src/example4.cpp)
add_dependencies(example4 chapter2_tutorials_gencfg)
target_link_libraries(example4 ${catkin_LIBRARIES})

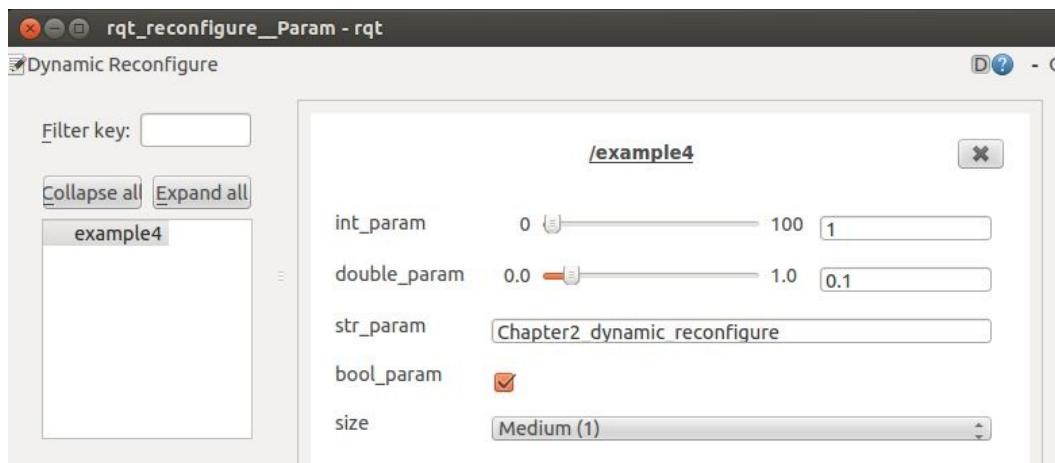
```

Now you have to compile and run the node and the Dynamic Reconfigure GUI as follows:

```
$ roscore
```

```
$ rosrun chapter2_tutorials example4  
$ rosrun rqt_reconfigure rqt_reconfigure
```

When you execute the last command, you will see a new window where you can dynamically modify the parameters of the node, as shown in the following screenshot:



Each time you modify a parameter with the slider, the checkbox, and so on, you will see the changes made in the shell where the node is running. You can see an example in the following screenshot:

```
[ INFO] [1403367196.752115948]: Reconfigure Request: 20 0.800000 qwert True 1  
[ INFO] [1403367196.942722848]: Reconfigure Request: 20 0.800000 qwerty True 1  
[ INFO] [1403367196.973132691]: Reconfigure Request: 20 0.800000 qwerty True 1  
[ INFO] [1403367197.183714401]: Reconfigure Request: 20 0.800000 qwertyu True 1  
[ INFO] [1403367197.217819018]: Reconfigure Request: 20 0.800000 qwertyu True 1  
[ INFO] [1403367203.160337570]: Reconfigure Request: 1 0.800000 qwertyu True 1  
[ INFO] [1403367203.188864110]: Reconfigure Request: 1 0.800000 qwertyu True 1
```

Thanks to Dynamic Reconfigure, you can program and test your nodes more efficiently and faster. Using the program with hardware is a good choice and you will learn more about it in the following chapters.

Summary

This chapter provided you with general information about the ROS architecture and how it works. You saw certain concepts, tools, and samples of how to interact with nodes, topics, and services. In the beginning, all of these concepts might look complicated and without use, but in upcoming chapters, you will start to understand their applications.

It is useful to practise using these terms and tutorials before continuing because in upcoming chapters we will assume that you know all of the concepts and uses.

Remember that if you have queries about something and you cannot find the solution in this section, you can use the official resources of ROS from <http://www.ros.org>. Additionally, you can ask the ROS Community questions at <http://answers.ros.org>.

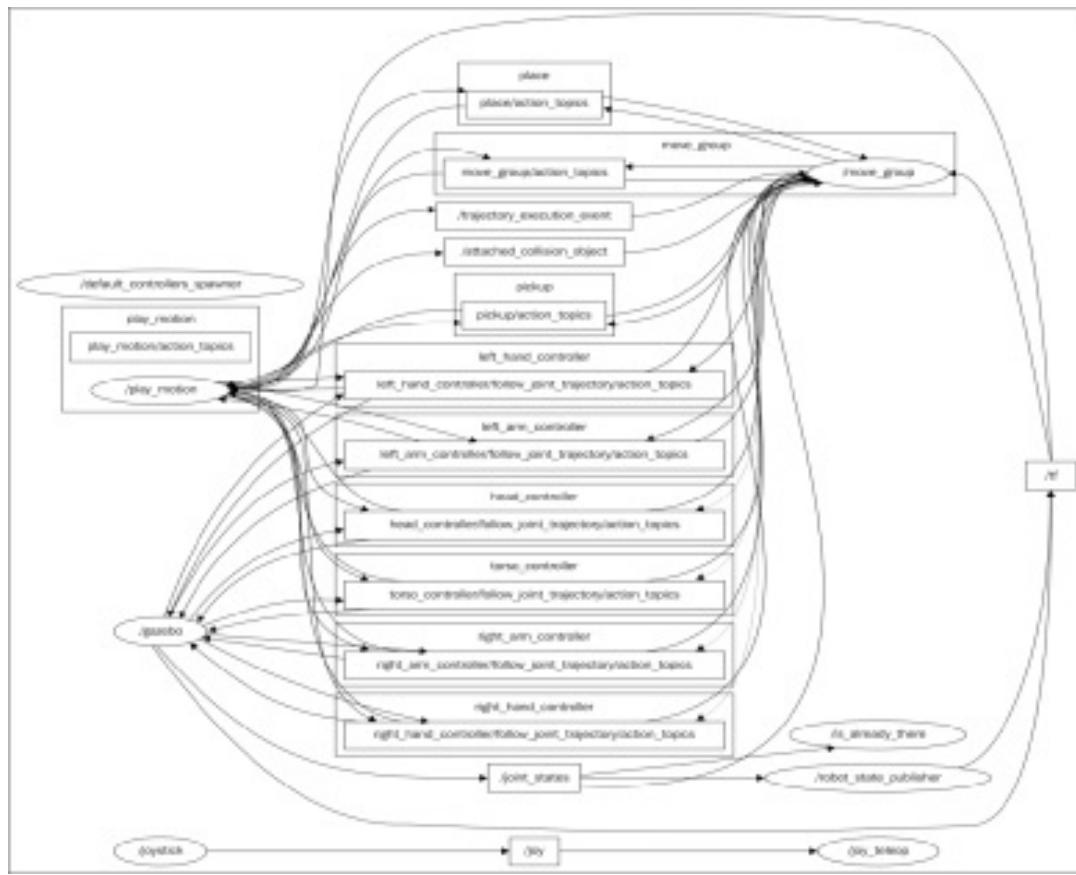
In the next chapter, you will learn how to debug and visualize data using ROS tools. This will help you to find problems, know whether what ROS is doing is correct, and better define what your expectations from it are.

Visualization and Debugging Tools

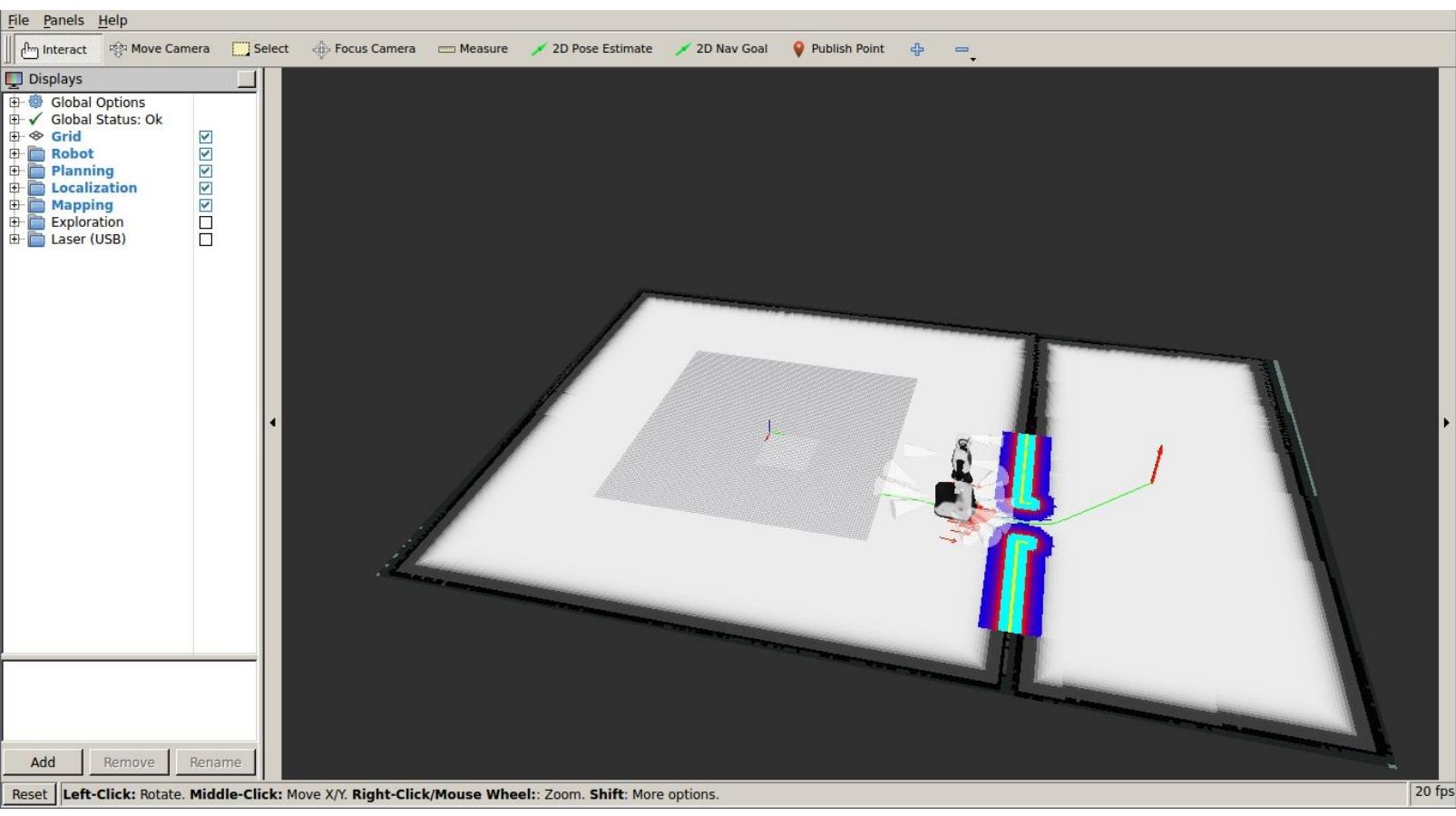
ROS has a good number of tools which allow the user and the developer to visualize and debug their code in order to detect and solve issues with both hardware and software. This comprises a message logging system similar to `log4cxx`, diagnostic messages, and also visualization and inspection tools, which provide a comprehensive list of the running nodes as well as how they are interconnected.

In this chapter, we will also show you how to debug an ROS node with the GDB debugger. The message logging API will be explained, and advice will be given on how to choose the logging level. Then, we will explain the set of ROS tools that allows us to inspect which processes are running and what information is communicated between them. For instance, the following figure shows a tool that visualizes the graph of the system, where the nodes are the processes running and the edges represent the data workflow through communication topics. This tool is `rqt_graph`, and in this case, it shows the nodes and topics which compose the **REEM robot** software system running on a Gazebo simulation.

You can see multiple controllers for the arms, torso, head, **MoveIt!** `move_group` node, pick and place action servers, and the `play_motion` node for pre-recorded movements. Other nodes publish `joint_states`, spawn the robot controllers, and control the joystick to move the mobile base.



Similarly, this chapter will show you how to plot scalar data in a time series, visualize images from a video stream, and represent different types of data in a 3D representation using (the widely known) `rviz` (or `rqt_rviz`), as shown in the following screenshot:



The preceding screenshot shows the REEM robot, which can be run in simulation with the following command:

```
| $ rosrun reem_2dnav_gazebo reem_navigation.launch
```

Note that before you install it, you need to follow the instructions provided at <http://wiki.ros.org/Robots/REEM>. The following sections in this chapter will cover the following topics on visualization and debugging: debugging our code in ROS; using logging messages in our code, with different severity levels, names, conditions, and throttling options. Here, we will explain the `rqt_logger_level` and `rqt_console` interfaces, which allow you to set the severity level of a node and visualize the message, respectively. We will also inspect the state of the ROS system by listing the nodes running, the topics, services, and actions they use to transfer messages among them, and the parameters declared in the ROS master server. We will explain `rqt_graph`, which shows nodes and topics in a directed graph representation, and `rqt_reconfigure`, which allows you to change dynamic parameters. We will also take a look at visualizing diagnostics information using the `unite_monitor` and `robot_monitor` interfaces, as well as plotting scalar data from messages using `rqt_plot`.

For non-scalar data, we will explain other `rqt` tools available in ROS, such as `rqt_image_view` to visualize images and `rqt_rviz` to show multiple data in a 3D representation. We will also show you how to visualize markers and interactive markers, and what frames are and how they are integrated into ROS messages and visualization tools. We will also explain how to use `rqt_tf_tree` to visualize the **Transform Frame (tf)** tree, along with how to save messages and replay them for simulation or evaluation purposes. We will also cover the `rqt_bag` interface.

Finally, other `rqt_gui` interfaces will be explained, as well as how to arrange them in a single GUI.

Most of the `rqt` tools can be run by simply inputting their name in the terminal, such as `rqt_console`, but in some cases this does not work and we must use `rosrun rqt_reconfigure rqt_reconfigure`, which always works; note that the name seems to be repeated, but it is actually the package and node names, one after the other.

Debugging ROS nodes

ROS nodes can be debugged as regular programs. They run as a process in the operating system and have a PID. Therefore, you can debug them as with any program using standard tools, such as `gdb`. Similarly, you can check for memory leaks with `memcheck` or profile the performance of your algorithm with `callgrind`. However, remember that in order to run a node, you must run the following command:

```
$ rosrun chapter3_tutorials example1
```

Unfortunately, you cannot simply run the command through `gdb` in the following way:

```
$ gdb rosrun chapter3_tutorials example1
```

In the following sections, we will explain how to call these tools for an ROS node to overcome this issue. Later, we will see how to add logging messages to our code in order to make it simple to diagnose problems; in practice, using logging messages helps to diagnose basic (and not so basic) problems without the need to debug the binaries. Similarly, we will discuss ROS introspection tools, which allow you to easily detect broken connections between nodes. Finally, even though we will provide a bottom-up overview, in practice we usually follow a top-down approach to diagnosing issues.

Using the GDB debugger with ROS nodes

In order to debug a C/C++ node with the `gdb` debugger, all we need to know is the location of the node executable. With the ROS Kinetic and `catkin` packages, the node executable is placed inside the `devel/lib/<package>` folder within the workspace. For example, in order to run the `example1` node from the `chapter3_tutorials` package in `gdb`, we have to proceed as follows, starting from the workspace folder (`/home/<user>/book_ws`):

```
| $ cd devel/lib/chapter3_tutorials
```

If you have run `catkin_make install`, you can also navigate to the `install/lib/chapter3_tutorials` directory using the following command:

```
| $ cd install/lib/chapter3_tutorials
```

Now we can run the node executable inside `gdb` with the following command:

```
| $ gdb example1
```



Remember that you must have roscore running before you start your node because it will need the master/server running.

Once roscore is running, you can start your node in `gdb` by pressing the R key (and Enter), and you can also list the associated source code with the L key as well as set breakpoints or any of the functionalities that `gdb` comes with. If everything is correct, you should see the following output in the `gdb` terminal after running the node:

```
(gdb) r
Starting program: /home/luis/devel/catkin_ws/devel/lib/chapter3_tutorials/example1
[Thread debugging using libthread_db enabled]
using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
[New Thread 0x7ffff170d700 (LWP 6618)]
[New Thread 0x7ffff0f0c700 (LWP 6619)]
[New Thread 0x7ffffebfff700 (LWP 6620)]
[New Thread 0x7ffffeb7fe700 (LWP 6625)]
[DEBUG] [1476313631.940149636]: This is a simple DEBUG message!
[DEBUG] [1476313631.940214159]: This is a DEBUG message with an argument: 3.1400
00
[DEBUG] [1476313631.940246937]: This is DEBUG stream message with an argument: 3
.14
[Thread 0x7ffffeb7fe700 (LWP 6625) exited]
[Thread 0x7ffff170d700 (LWP 6618) exited]
[Thread 0x7ffff0f0c700 (LWP 6619) exited]
[Thread 0x7ffffebfff700 (LWP 6620) exited]
[Inferior 1 (process 6613) exited normally]
(gdb) ■
```

Attaching a node to GDB while launching ROS

In many cases, we might get a `launch` file that takes care of starting the node, as you can see in the following example:

```
<launch>
  <node pkg="chapter3_tutorials" type="example1" name="example1"/>
</launch>
```

In order to attach it to `gdb`, we must add `launch-prefix="xterm -e gdb --args"` as follows:

```
<launch>
  <node pkg="chapter3_tutorials" type="example1" name="example1"
    launch-prefix="xterm -e gdb --args"/>
</launch>
```

Similarly, you can also add `output="screen"` to make the node output appear on the terminal. With this launch prefix, a new xterm terminal will be created with the node attached to `gdb`. Set breakpoints if needed, and then press the **C** or **R** key to run the node and debug it. One of the common uses you will find of this simple workflow is to obtain a **backtrace (bt)** if the node crashes.

Profiling a node with valgrind while launching ROS

Additionally, we can use the same attribute to attach the node to diagnosis tools. For example, we can run our program through `valgrind` (see <http://valgrind.org> for further information) to detect memory leaks using `memcheck` and perform profiling analysis using `callgrind`. Contrary to attaching to `gdb`, we do not need to start `xterm`:

```
<launch>
  <node pkg="chapter3_tutorials" type="example1"
    name="example1" output="screen"
    launch-prefix="valgrind"/>
</launch>
```

Enabling core dumps for ROS nodes

Although ROS nodes are actually regular executables, there is a tricky point to enabling core dumps, which can later be used in a `gdb` session. First of all, we have to set an unlimited core size; the current value can be checked with `ulimit -c`. Note that this is also required for any executable and not just ROS nodes:

```
| $ ulimit -c unlimited
```

Then, to allow core dumps to be created, we must set the core filename to use the pid process by default. Otherwise, they will not be created because at `$ROS_HOME`, there is already a `core` directory to prevent core dumps. Therefore, in order to create core dumps with the name and path `$ROS_HOME/core.PID`, we must run the following command:

```
| $ echo 1 | sudo tee /proc/sys/kernel/core_uses_pid
```

Logging messages

It is good practice to include messages that indicate what the program is doing; we must do it without compromising the efficiency of our software and the clarity of its output. In ROS, we have an API which covers both features, built on top of `log4cxx` (a port of the well-known `log4j` logger library). In brief, we have several levels of messages, which might have a name (named messages) and depend on a condition or even throttle. All of them have a negligible footprint on performance if they are masked by the current verbosity level (even at compile time). They also have full integration with other ROS tools to visualize and filter the messages from all the nodes running.

Outputting logging messages

ROS comes with a great number of functions/macros to output logging messages. It supports different levels, conditions, STL streams, throttling, and other features that we will see in this section. To start with something simple, an information message is printed with this code in C++:

```
| $ ROS_INFO("My INFO message.");
```

In order to have access to these logging functions/macros, the following header is required:

```
| #include <ros/ros.h>
```

This includes the following header (where the logging API is defined):

```
| #include <ros/console.h>
```

As a result of running a program with the preceding message, we will get the following output:

```
| [ INFO] [1356440230.837067170]: My INFO message.
```

All messages are printed with their level and the current timestamp (your output might differ for this reason) before the actual message, with both between square brackets. The timestamp is the epoch time, that is, the number of seconds and nanoseconds since January 1, 1970. Then, we have our message-always with a new line.

This function allows parameters in the same way as the C `printf` function does. For example, we can print the value of a floating point number in the variable `val` with this code:

```
| floatval = 1.23;
| ROS_INFO("My INFO message with argument: %f", val);
```

C++ STL streams are also supported with `*_STREAM` functions. Therefore, the previous instruction is equivalent to the following using streams:

```
| ROS_INFO_STREAM("My INFO message with argument: " <<val);
```

Note that we did not specify any stream since the API takes care of that by redirecting to `cout/cerr`, a file, or both.

Setting the debug message level

ROS supports the following logging levels (in increasing order of relevance):

- DEBUG
- INFO
- WARN
- ERROR
- FATAL

These levels are part of the function used to output messages, so it's something you can easily experiment with, simply by using the syntax below:

```
| ROS_<LEVEL>[_<OTHER>]
```

Each message is printed with a particular color. The colors are as follows:

| | |
|-------|-----------|
| DEBUG | in green |
| INFO | in white |
| WARN | in yellow |
| ERROR | in red |
| FATAL | in purple |

Each message level is meant to be used for a different purpose. Here, we suggest the uses for each of the levels:

- **DEBUG**: Use them only when debugging; this information should not be displayed in a deployed application, as it's only for testing purposes
- **INFO**: These should be standard messages to indicate significant steps or what the node is doing
- **WARN**: As the name suggests, use them to provide a warning that something might be wrong, missed, or abnormal, but that the application can still run regardless
- **ERROR**: These can be used to indicate errors, although the node can still recover from them; however, they set a certain expectation regarding the node's behavior
- **FATAL**: These messages usually expose errors that prevent the node from continuing its operation

Configuring the debugging level of a particular node

By default, only messages of `INFO` or higher levels are shown. ROS uses these levels to filter the messages printed by a particular node. There are many ways to do so. Some of them are set at the time of compilation and some messages aren't even compiled below a given verbosity level; others can be changed before execution using a configuration file, and it is also possible to change the logging level dynamically using the `rqt_console` and `rqt_logger_level` tools.

It is possible to set the logging level at compile time in our source code, but this is very uncommon and not recommended as it requires us to modify the source code to change the logging level.

Nevertheless, in some cases we need to remove the overhead of all the logging functions below a given level. In those cases, we won't be able to see those messages later because they get removed from the code and are not simply disabled. To do so, we must set `ROSCONSOLE_MIN_SEVERITY` to the minimum severity level desired, or even none, in order to avoid any message (even `FATAL`). The macros are as follows:

```
| ROSCONSOLE_SEVERITY_DEBUG  
| ROSCONSOLE_SEVERITY_INFO  
| ROSCONSOLE_SEVERITY_WARN  
| ROSCONSOLE_SEVERITY_ERROR  
| ROSCONSOLE_SEVERITY_FATAL  
| ROSCONSOLE_SEVERITY_NONE
```

The `ROSCONSOLE_MIN_SEVERITY` macro is defined in `<ros/console.h>` to the `DEBUG` level if not given. Therefore, we can pass it as a build argument (with `-D`) or put it before all the headers. For example, to show only `ERROR` (or higher) messages, we will put this in our source code:

```
| #define ROSCONSOLE_MIN_SEVERITY ROSCONSOLE_SEVERITY_ERROR
```

Alternatively, we can set this to all the nodes in a package, setting this macro in `CMakeLists.txt` by adding this line:

```
| add_definitions(-DROSCONSOLE_MIN_SEVERITY  
| =ROSCONSOLE_SEVERITY_ERROR)
```

On the other hand, we have the more flexible solution of setting the minimum logging level in a configuration file. We create a `config` folder with a file, such as `chapter3_tutorials.config`, and this content (edit the file provided since it is set to `DEBUG`):

```
| log4j.logger.ros.chapter3_tutorials=ERROR
```

Then, we must set the `ROSCONSOLE_CONFIG_FILE` environment variable to point to our file. We can do this on a launch file that also runs the node. Therefore, we will extend the `launch` file shown earlier to do so with the `env` (environment variable) element, as shown here:

```
| <launch>  
| <!-- Logger config -->  
| <env name="ROSCONSOLE_CONFIG_FILE"
```

```
value="$(find  
chapter3_tutorials)/config/chapter3_tutorials.config"/>  
  
<!-- Example 1 -->  
<node pkg="chapter3_tutorials" type="example1" name="example1"  
output="screen"/>  
</launch>
```

The environment variable takes the configuration file shown previously, which contains the logging level specification for each named logger; in this case, it is for the package name.

Giving names to messages

By default, ROS assigns several names to the node loggers. The messages discussed until now will be named after the node's name. In complex nodes, we can give a name to those messages of a given module or functionality. This is done with `ROS_<LEVEL>[_STREAM]_NAMED` functions (see the `example2` node):

```
| ROS_INFO_STREAM_NAMED(  
|   "named_msg",  
|   "My named INFO stream message; val = " <>val  
| );
```

With named messages, we can set different initial logging levels for each named message using the configuration file and modify them individually later. We must use the name of the messages as children of the package in the specification; for example, for `named_msg` messages, we will use the following code:

```
| log4j.logger.ros.chapter3_tutorials.named_msg=ERROR
```

Conditional and filtered messages

Conditional messages are printed only when a given condition is satisfied. To use them, we have the `ROS_<LEVEL>[_STREAM]_COND[_NAMED]` functions; note that they can be named messages as well (see the `example2` node for more examples and combinations):

```
ROS_INFO_STREAM_COND(
    val< 0.,
    "My conditional INFO stream message; val (" <<val<< ") < 0"
);
```

Filtered messages are similar to conditional messages in essence, but they allow us to specify a user-defined filter that extends `ros::console::FilterBase`; we must pass a pointer to such a filter in the first argument of a macro with the format `ROS_<LEVEL>[_STREAM]_FILTER[_NAMED]`. The following example is taken from the `example2` node:

```
struct MyLowerFilter : public ros::console::FilterBase {
    MyLowerFilter(const double& val) : value(val) {}
    inline virtual bool isEnabled() { return value < 0.; }
    double value;
};

MyLowerFilter* filter_lower;

ROS_INFO_STREAM_FILTER(&filter_lower,
    "My filter INFO stream message; val (" <<val<< ") < 0"
);
```

Showing messages once, throttling, and other combinations

It is also possible to control how many times a given message is shown. We can print it only once with `ROS_<LEVEL>[_STREAM]_ONCE[_NAMED]`:

```
| for(int i = 0; i< 10; ++i ) {  
|   ROS_INFO_STREAM_ONCE("My once INFO stream message; i = " <<i);  
| }
```

This code from the `example2` node will show the message only once.

However, it is usually better to show the message with a certain frequency. For that, we have throttle messages. They have the same format as the once message, but here `ONCE` is replaced with `THROTTLE`. They also include a first argument, which is period in seconds, that is, it is printed only every period seconds:

```
| for(int i = 0; i< 10; ++i ) {  
|   ROS_INFO_STREAM_THROTTLE(2,  
|     "My throttle INFO stream message; i = " <<i);  
|   ros::Duration( 1 ).sleep();  
| }
```

Finally, note that named, conditional, and once/throttle messages can be used together with all the available levels.

Nodelets also have some support in terms of logging messages. Since they have their own namespace, they have a specific name to differentiate the message of one nodelet from another. Simply put, all the macros shown until now are valid, but instead of `ros_*`, we have `NODELET_*`. These macros will only compile inside nodelets. In addition, they operate by setting up a named logger with the name of the nodelet running so that you can differentiate between the outputs of two nodelets of the same type running in the same nodelet manager. They also have an advantage in that you can turn one specific nodelet into the debug level instead of all the nodelets of a specific type.

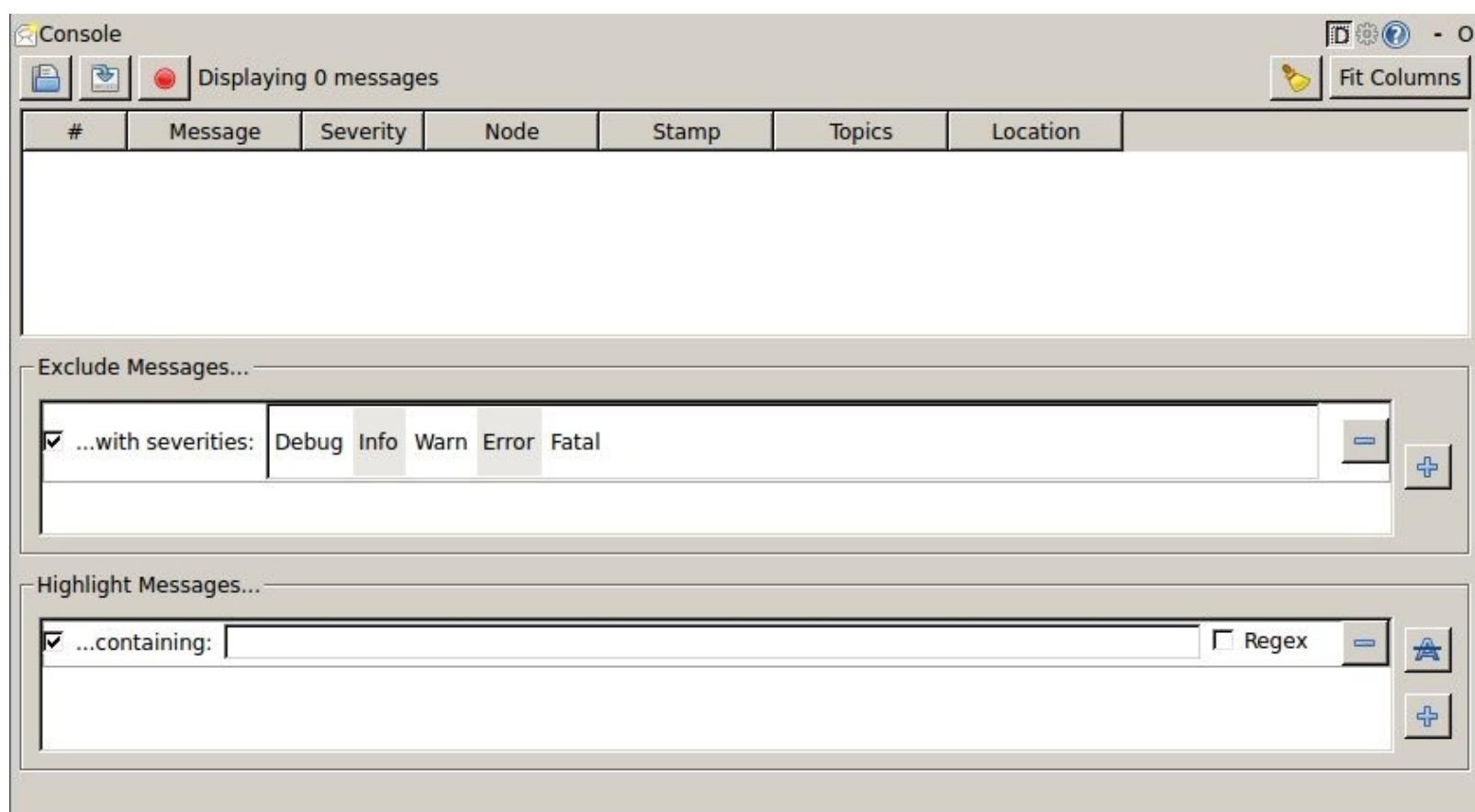
Using rqt_console and rqt_logger_level to modify the logging level on the fly

ROS provides a series of tools to manage logging messages. In ROS Kinetic, we have two separate GUIs: `rqt_logger_level` to set the logging level of the nodes or named loggers and `rqt_console` to visualize, filter, and analyze the logging messages.

In order to test this, we are going to use `example3`. Run `roscore` and `rqt_console` to see the logging messages:

```
| $ rosrun rqt_console rqt_console
```

The following window will open:



Now, run the node:

```
| $ rosrun chapter3_tutorials example3
```

You will start seeing the logging messages, as the following screenshot shows. Note that `roscore` must be running and that you must press the recording button on `rqt_console`.

In `rqt_console`, the messages are collected and shown in a table where different columns separate the timestamp, the message itself, the severity level, and the node that produced the message, alongside other information. You can fit the columns automatically by pressing the `Resize columns` button. If you double-click on a message, you can see all the information, including the line of code that generated it, as shown in the following screenshot:

This interface allows you to pause, save, and load previous/saved logging messages. We can clear the list

of messages and filter them. In ROS Kinetic, excluding those that are filtered, messages have specific interfaces depending on the filter criteria. For instance, nodes can be filtered with a single rule, where we select the nodes we want to exclude. Additionally, in the same way, we can set highlighting filters. This is shown in the following screenshot:

The screenshot shows the rqt_console application window. At the top, it displays "Displaying 8 of 22 messages". Below this is a table with columns: #, Message, Severity, Node, Stamp, Topics, and Location. The data in the table is as follows:

| # | Message | Severity | Node | Stamp | Topics | Location |
|-----|-------------|----------|-----------|-----------------|---------|-----------------|
| #21 | FATAL me... | Fatal | /example3 | 22:20:02.943... | /rosout | /home/enriqu... |
| #20 | ERROR m... | Error | /example3 | 22:20:02.943... | /rosout | /home/enriqu... |
| #15 | FATAL me... | Fatal | /example3 | 22:20:01.943... | /rosout | /home/enriqu... |
| #14 | ERROR m... | Error | /example3 | 22:20:01.943... | /rosout | /home/enriqu... |
| #10 | FATAL me... | Fatal | /example3 | 22:20:00.943... | /rosout | /home/enriqu... |
| #9 | ERROR m... | Error | /example3 | 22:20:00.943... | /rosout | /home/enriqu... |
| #5 | FATAL me... | Fatal | /example3 | 22:19:59.943... | /rosout | /home/enriqu... |
| #4 | ERROR m... | Error | /example3 | 22:19:59.943... | /rosout | /home/enriqu... |

Below the table are two sections for filtering:

- Exclude Messages...**: A checkbox labeled "...with severities:" followed by buttons for Debug, Info, Warn, Error, and Fatal. The Error and Fatal buttons are highlighted.
- Highlight Messages...**: A checkbox labeled "...containing:" followed by a text input field, a button for "Messages matching ANY of these rules will be highlighted", a "Regex" checkbox, and a plus sign icon for adding rules.

As an example, the messages from the previous image are filtered by excluding those with a severity different to Error and Fatal.

In order to set the severity of the loggers, we must run the following command:

```
| $ rosrun rqt_logger_level rqt_logger_level
```

Here, we can select the node, then the named logger, and finally its severity. Once we modify it, the new messages received with a severity below the desired level will not appear in rqt_console:

Shown in the following screenshot is an example where we have set the severity level to the minimum (Debug) for the named logger, `ros.chapter3_tutorials.named_msg`, of the `example3` node; remember that the named loggers are created by the `*_NAMED` logging functions

As you can see, every node has several internal loggers by default, which are related to the ROS communication API, among others; in general, you should not reduce their severity.

Inspecting the system

When our system is running, we might have several nodes and many more topics publishing messages amongst each other. We might also have nodes providing actions or services. For large systems, it is important to have tools that let us see what is running at a given time. ROS provides basic but very powerful tools with this goal in mind, from the CLI to GUI applications.

Listing nodes, topics, services, and parameters from our perspective, we should start with the most basic level of introspection. We are going to see how to obtain the list of nodes running and topics and services available at a given time:

| Obtain the list of all | Command |
|-------------------------------|------------------------------|
| Nodes running | <code>rosnode list</code> |
| Topics of all nodes running | <code>rostopic list</code> |
| Services of all nodes running | <code>rosservice list</code> |
| Parameters on the server | <code>rosparam list</code> |

We recommend that you go back to *Chapter 2, ROS Architecture and Concepts*, to see how these commands also allow us to obtain the message type sent by a particular topic, as well as its fields, using `rosmg show`.

Any of these commands can be combined with regular bash commands, such as `grep`, to look for the desired nodes, topics, services, or parameters. For example, action goal topics can be found using the following command:

```
$ rostopic list | grep goal
```

The `grep` command looks for text or patterns in a list of files or the standard output.

Additionally, ROS provides several GUIs for you to play with topics and services. First, `rqt_top` shows the nodes running in an interface, similar to a **table of processes (ToP)**, which allows us to rapidly see all the nodes and resources they are using. For the following screenshot, we have used the REEM simulation with the navigation stack running as an example:

Process Monitor

| Node | PID | CPU % | Mem % | Num Threads |
|------------------------------|-------|--------|-------|-------------|
| .../rqt_gui_py_node_10852 | 10852 | 13.10 | 1.07 | 5 |
| /rosout | 6271 | 1.00 | 0.12 | 5 |
| /robot_state_publisher | 6311 | 15.20 | 0.19 | 6 |
| /play_motion | 6398 | 20.20 | 0.54 | 9 |
| /move_group | 6365 | 25.30 | 0.72 | 19 |
| /move_base | 6575 | 26.30 | 0.43 | 11 |
| /map_server | 6408 | 9.10 | 0.14 | 5 |
| /joystick | 6319 | 9.10 | 0.11 | 5 |
| /joy_teleop | 6314 | 6.10 | 0.20 | 6 |
| /is_already_there | 6399 | 7.10 | 0.20 | 6 |
| /gazebo | 6296 | 139.00 | 3.78 | 69 |
| /default_controllers_spawner | 6299 | 6.10 | 0.20 | 5 |
| /amcl | 6438 | 19.20 | 0.26 | 7 |

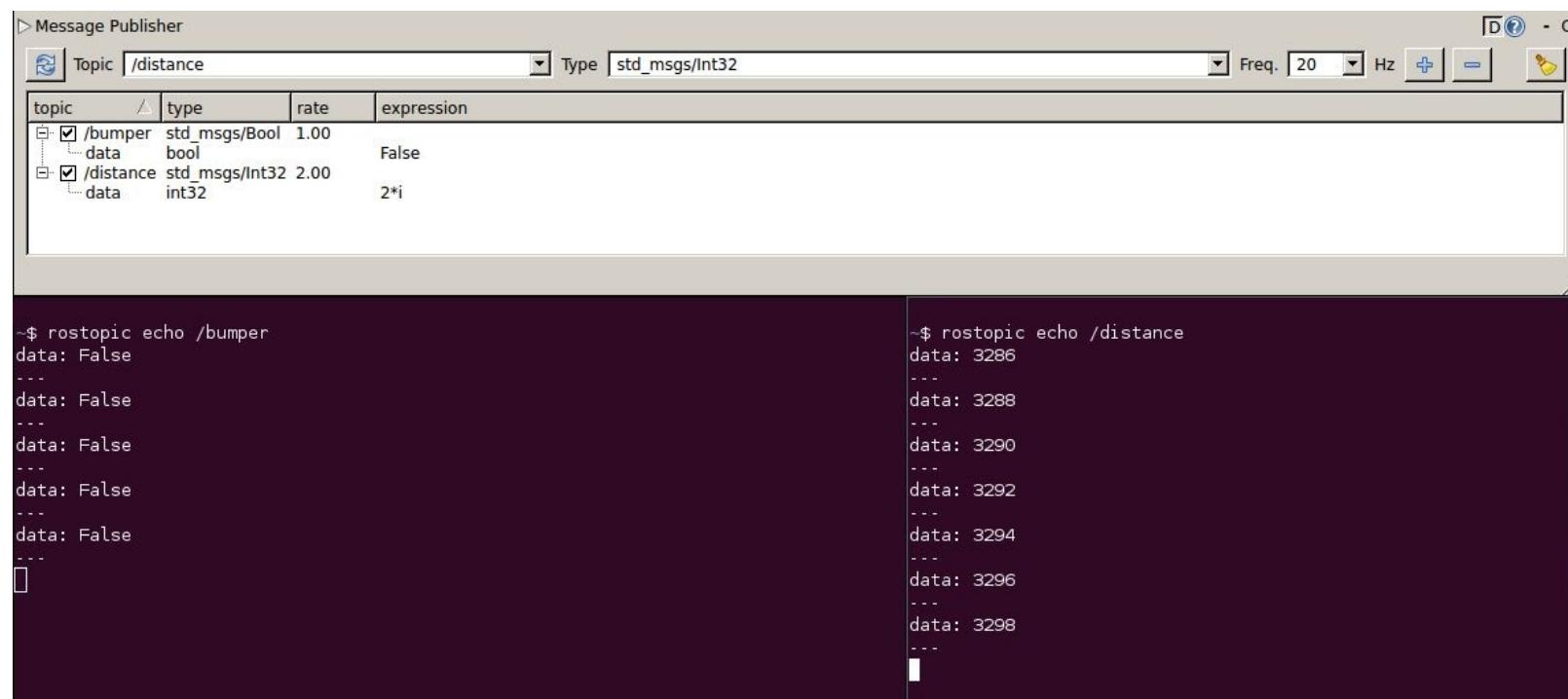
Kill Node

On the other hand, `rqt_topic` provides us with information about topics, including publishers, subscribers, the publishing rate, and messages published. You can view the message fields and select the topics you want to subscribe to in order to analyze their bandwidth and rate (Hz) and see the latest message published; note that latched topics do not usually publish continuously, so you will not see any information about them. The following screenshot shows this:

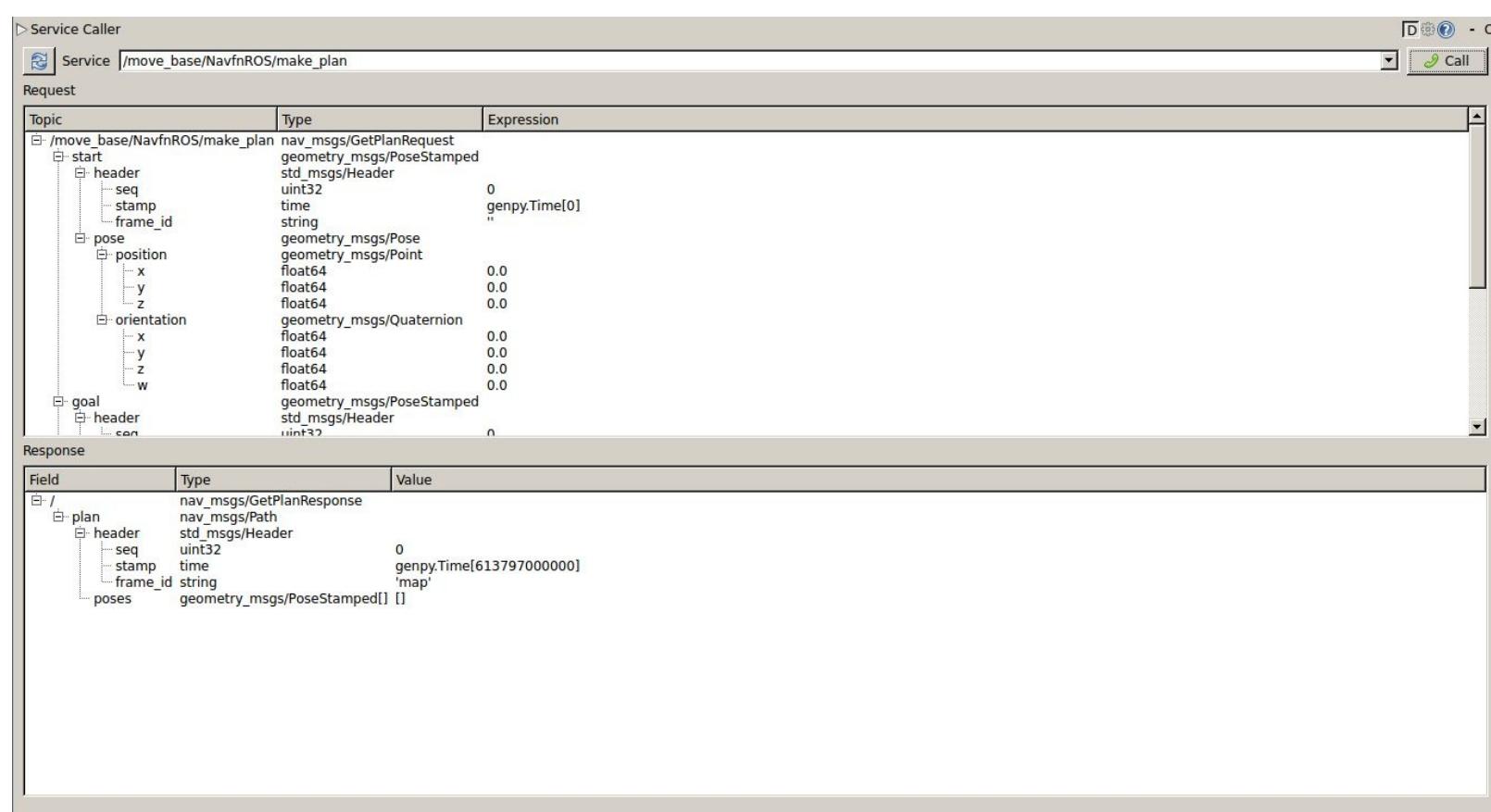
Topic Monitor

| Topic | Type | Bandwidth | Hz | Value |
|--------------------------------|---|-----------|---------|--|
| .../amcl_pose | geometry_msgs/PoseWithCovarianceStamped | unknown | unknown | |
| header | std_msgs/Header | | | |
| frame_id | string | | | 'map' |
| seq | uint32 | | | 0 |
| stamp | time | | | genpy.Time[5917000000] |
| pose | geometry_msgs/PoseWithCovariance | | | (0.20457495899314834, -0.001707561741432752, ...) |
| covariance | float64[36] | | | |
| pose | geometry_msgs/Pose | | | |
| orientation | geometry_msgs/Quaternion | | | |
| w | float64 | | | 0.9999948663862841 |
| x | float64 | | | 0.0 |
| y | float64 | | | 0.0 |
| z | float64 | | | 0.003204247349652341 |
| position | geometry_msgs/Point | | | |
| x | float64 | | | 0.005243756470619018 |
| y | float64 | | | 0.023378910660500424 |
| z | float64 | | | 0.0 |
| | dynamical_reconfigure/ConfigDescription | | | not monitored |
| | dynamical_reconfigure/Config | | | not monitored |
| | moveit_msgs/AttachedCollisionObject | | | not monitored |
| | sensor_msgs/Camerainfo | | | not monitored |
| .../back_camera/image | sensor_msgs/Image | 10.22MB/s | 9.34 | |
| | sensor_msgs/CompressedImage | | | not monitored |
| | dynamical_reconfigure/ConfigDescription | | | not monitored |
| | dynamical_reconfigure/Config | | | not monitored |
| | sensor_msgs/CompressedImage | | | not monitored |
| | dynamical_reconfigure/ConfigDescription | | | not monitored |
| | dynamical_reconfigure/Config | | | not monitored |
| | dynamic_reconfigure/ConfigDescription | | | not monitored |
| | dynamic_reconfigure/Config | | | not monitored |
| | dynamic_reconfigure/Config | | | not monitored |
| | dynamic_reconfigure/Config | | | not monitored |
| | dynamic_reconfigure/Config | | | not monitored |
| | sensor_msgs/Imu | 16.80KB/s | 50.00 | |
| | geometry_msgs/Vector3 | | | (-1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0) |
| header | std_msgs/Header | | | |
| linear_acceleration | geometry_msgs/Vector3 | | | |
| x | float64 | | | 0.0 |
| y | float64 | | | 0.0 |
| z | float64 | | | 0.0 |
| linear_acceleration_covariance | float64[9] | | | (-1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0) |
| orientation | geometry_msgs/Quaternion | | | |
| orientation_covariance | float64[9] | | | (0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0) |
| /bumper_states | gazebo_msgs/ContactsState | | | can not get message class for type "gazebo_msgs/C... |
| /clock | rostopic_msgs/Clock | | | not monitored |
| /diagnostics | diagnostic_msgs/DiagnosticArray | | | not monitored |

Similarly, `rqt_publisher` allows us to manage multiple instances of rostopic pub commands in a single interface. It also supports Python expressions for the published messages and fixed values. In the following screenshot, we will see two example topics being published (we will see the messages using `rostopic echo <topic>` in two different terminals):



Note that `rqt_service_caller` does the same thing for multiple instances of `rosservice call` commands. In the following screenshot, we will call the `/move_base/NavfnROS/make_plan` service, which is where we have to set up the request; for empty services, this is not needed, due to the `/global_localization` service from the `/amcl` node. After clicking on the Call button, we will obtain the response message. For this example, we have used the REEM simulation with the navigation stack running:



Inspecting the node's graph online with rqt_graph

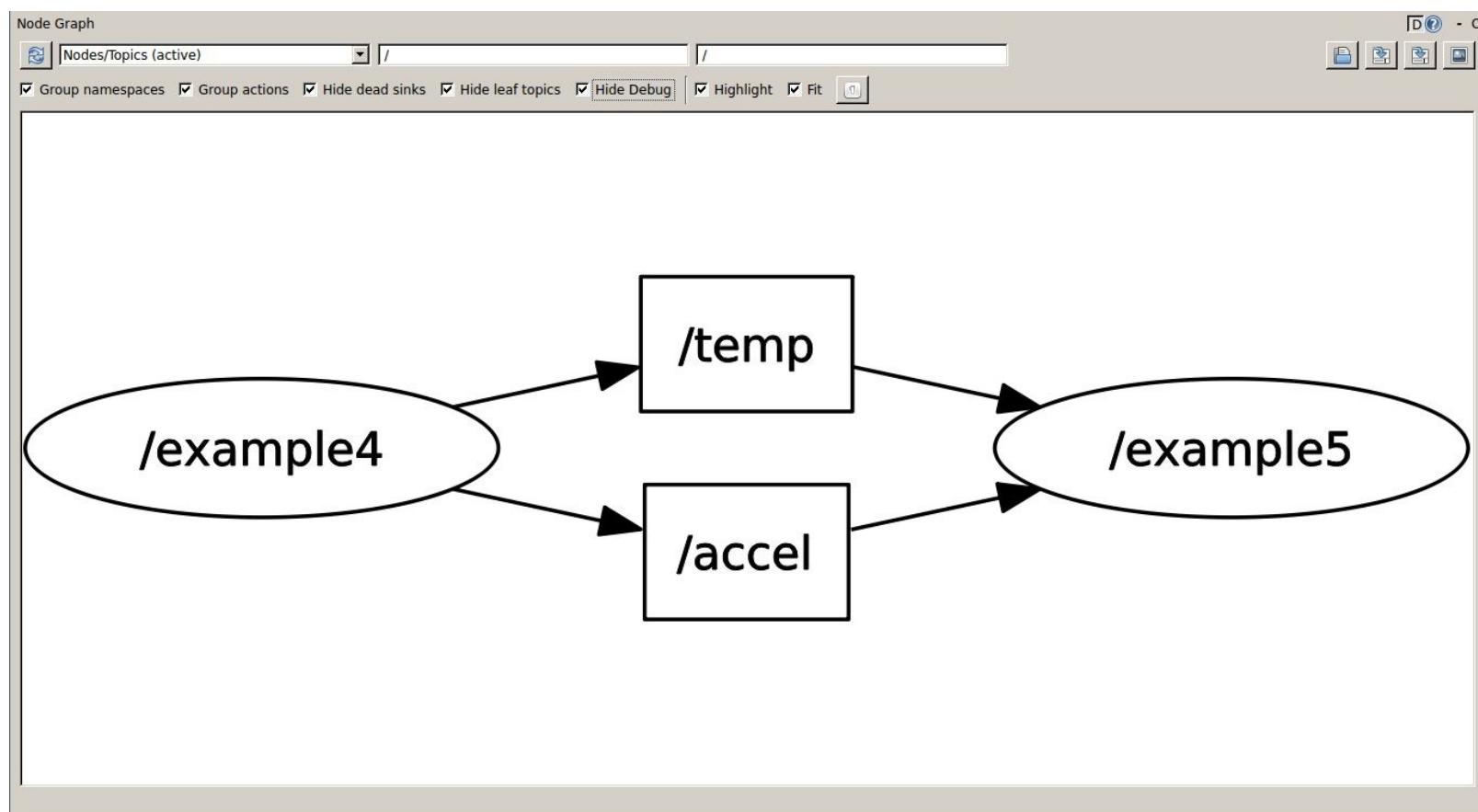
The current state of an ROS session can be shown as a directed graph where the nodes running are the graph nodes and the edges are the publisher-subscriber connections amongst these nodes through the topics. This graph is drawn dynamically by `rqt_graph`:

```
| $ rosrun rqt_graph rqt_graph
```

In order to illustrate how to inspect the nodes, topics, and services with `rqt_graph`, we are going to run the `example4` and `example5` nodes simultaneously with the following launch file:

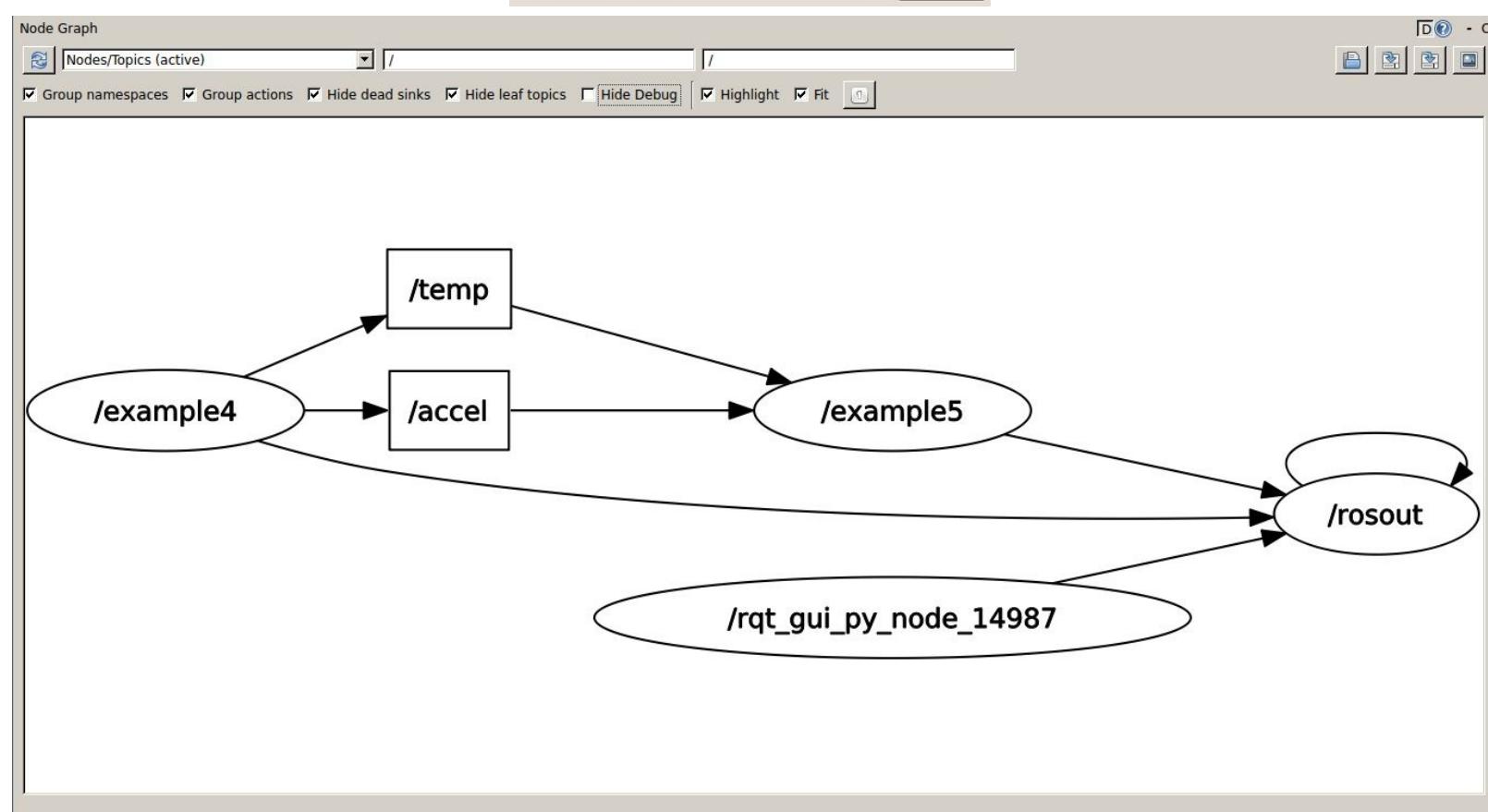
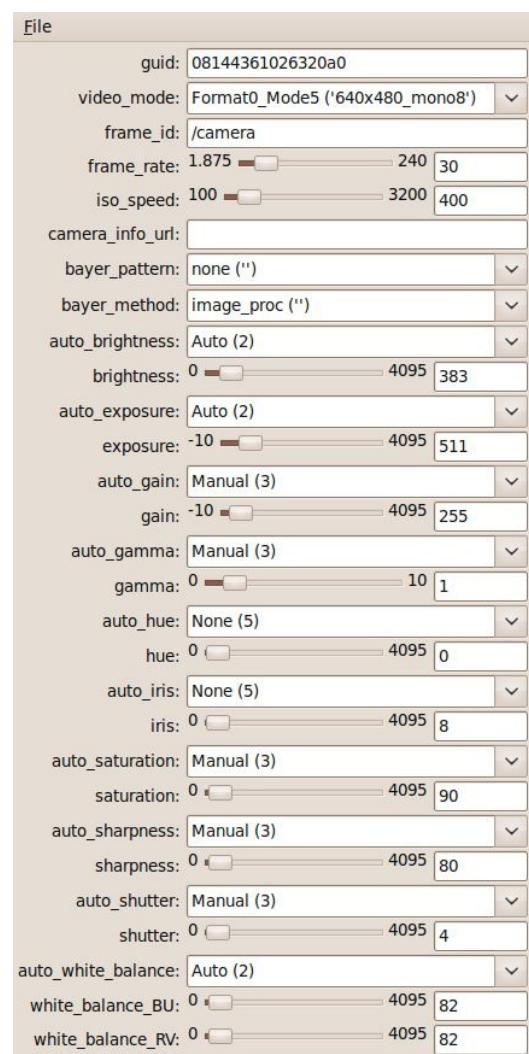
```
| $ roslaunch chapter3_tutorials example4_5.launch
```

The `example4` node publishes in two different topics and calls a service. Meanwhile, `example5` subscribes to those topics and also has the service server to attend the request queries and provide the response. Once the nodes are running, we have the node's topology, as shown in the following screenshot:



In this screenshot, we have the nodes `example4` and `example5` connected by the topics `temp` and `accel`. Since `Hide Debug` is selected, we do not see the ROS server node `rosout` as well as the `rosout` topic that publishes the logging messages for the diagnostic aggregator in the server as we did previously. We can deselect this option to show the debug nodes/topics so that the ROS server is shown, as well as the `rqt_graph` node itself.

It is useful to hide these nodes for larger systems because it simplifies the graph. Additionally, with ROS Kinetic, the nodes in the same namespace are grouped - for example, the image pipeline nodes:



When there is a problem in the system, the nodes appear in red all the time (not just when we move the mouse over them). In those cases, it is useful to select All Topics to also show unconnected topics. This usually shows misspelled topic names that break connections among nodes.

When running nodes on different machines, `rqt_graph` shows its great high-level debugging capabilities, as it shows whether the nodes see each other from one machine to the other, enumerating the connections.

Finally, we can enable statistics to see the message rate and bandwidth represented in the topic edge, with the rate written and the line width. We must set this parameter before running `rqt_graph` in order to have this information available:

```
| $ rosparam set enable_statistics true
```

But this is not the only thing we need to do to enable statistics. To be able to get the desired information, the `rqt_topic` utility needs to be launched and the appropriate topics need to be ticked. It should also be noted that the parameter might need to be enabled before the relevant nodes are launched.

Setting dynamic parameters

If a node implements a dynamic reconfigure parameter server, we can use `rqt_reconfigure` to modify it on the fly. Run the following example, which implements a dynamic reconfigure server with several parameters (see the `cfg` file in the `cfg` folder of the package):

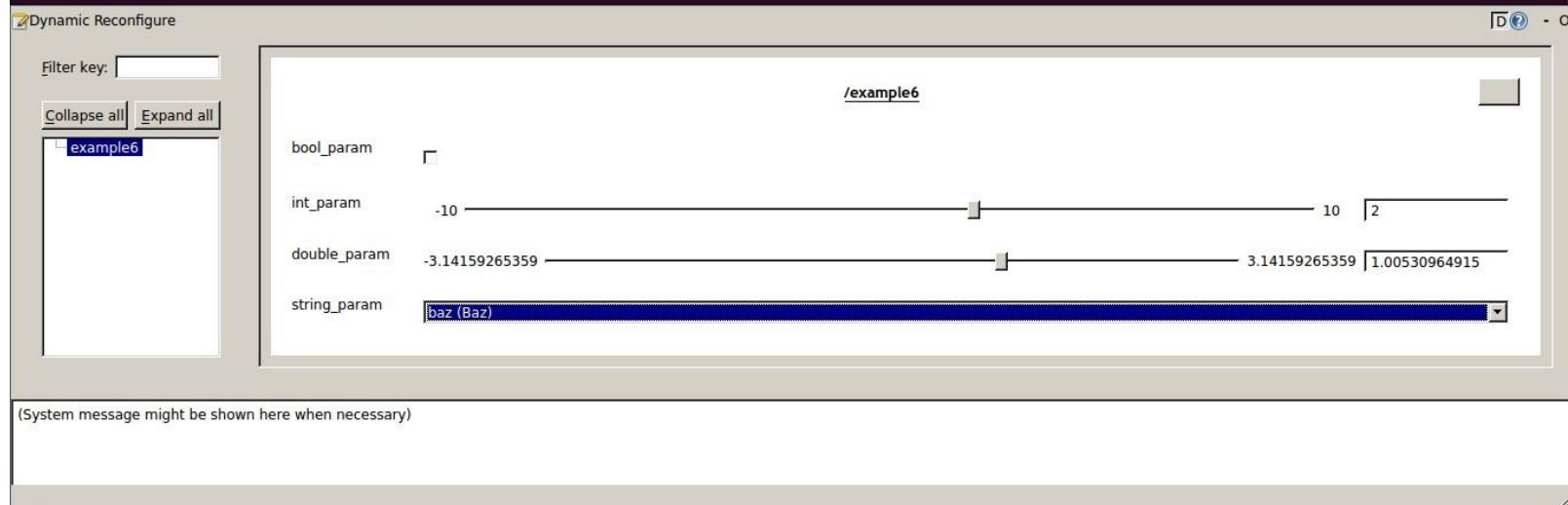
```
| $ roslaunch chapter3_tutorials example6.launch
```

With the dynamic reconfigure server running, open the GUI with the following command:

```
| $ rosrun rqt_reconfigure rqt_reconfigure
```

Select the `example6` server in the left-hand side panel, and you will see its parameters, which you can modify directly. The parameter changes take effect immediately, running the code inside a callback method in the example source code, which checks for the validity of the values. In this example, the parameters are printed every time they are changed, that is, when the callback method is executed. The following screenshot is an example of the expected behavior:

```
string = Foo
[ INFO] [1404155692.740704020]: New configuration received with level = 3:
bool   = 0
int    = 2
double = 1.00531
string = Foo
[ INFO] [1404155699.438546902]: New configuration received with level = 0:
bool   = 0
int    = 2
double = 1.00531
string = Bar
[ INFO] [1404155713.895218175]: New configuration received with level = 0:
bool   = 0
int    = 2
double = 1.00531
string = Baz
[]
```



Dynamic parameters were originally meant for drivers, in order to make it easier to modify them on the fly. For this reason, several drivers already implement them; nevertheless, they can be used for any other node. Examples of drivers that implement them are the `hokuyo_node` driver for the Hokuyo laser rangefinders or the Firewire `camera1394` driver. Indeed, in the case of Firewire cameras, it is common for drivers to support changing configuration parameters of the sensor, such as the frame rate, shutter speed, and brightness, among others. The ROS driver for FireWire (IEEE 1394, a and b) cameras can be run with the

following command:

```
$ rosrun camera1394 camera1394_node
```

Once the camera is running, we can configure its parameters with `rqt_reconfigure`, and we should see something similar to what's shown in the following screenshot:



Note that we will cover how to work with cameras in *Chapter 8, Computer Vision*, where we will also explain these parameters from a developer's point of view.

Dealing with the unexpected

ROS has several tools for detecting potential problems in all the elements of a given package. Just move with `roscd` to the package you want to analyze. Then, run `rosjwtf`. For `chapter3_tutorials`, we have the following output. Note that if you have something running, the ROS graph will be analyzed too. We have run the `roslaunch chapter3_tutorials example6.launch` command, which yields an output similar to the following screenshot:

```
$ rosjwtf
No package or stack in context
=====
Static checks summary:

Found 1 warning(s).
Warnings are things that may be just fine, but are sometimes at fault

WARNING You have pip installed packages on Ubuntu, remove and install using Debian packages: rospkg -- 

=====
Beginning tests of your ROS graph. These may take awhile...
analyzing graph...
... done analyzing graph
running graph rules...
... done running graph rules

Online checks summary:

No errors or warnings

WARNING: Package name "3dof_bringup" does not follow the naming conventions. It should start with a lower case letter and only contain lower case letters, digits and underscores.
WARNING: Package name "3dof_robot" does not follow the naming conventions. It should start with a lower case letter and only contain lower case letters, digits and underscores.
WARNING: Package name "3dof_description" does not follow the naming conventions. It should start with a lower case letter and only contain lower case letters, digits and underscores.
WARNING: Package name "3dof_controller_configuration" does not follow the naming conventions. It should start with a lower case letter and only contain lower case letters, digits and underscores.
process[rosout-1]: started with pid [12411]
started core service [/rosout]
process[example6-2]: started with pid [12423]
[ INFO] [1404156269.276210812]: New configuration received with level = 4294967295:
bool   = 1
int    = 0
double = 0
string = Foo
[]
```

In general, we should expect no error or warning, but even then some of them are innocuous. In the previous screenshot, we can see that `rosjwtf` does not detect any error; it only issues a warning about pip, which may sometimes generate problems with the Python code installed in the system. Note that the purpose of `rosjwtf` is to signal potential problems; we are responsible for checking whether they are real or meaningless ones, as in the previous case.

Another useful tool is `catkin_lint`, which helps to diagnose errors with catkin, usually in the `CMakeLists.txt` and `package.xml` files. For `chapter3_tutorials`, we have the following output:

```
$ catkin_lint -W2 --pkg chapter3_tutorials
```

With `-W2`, we see warnings that can be usually ignored, such as the ones shown in the following screenshot:

```
$ catkin_lint -W2 --pkg chapter3_tutorials
chapter3_tutorials: notice: target name 'example6' might not be sufficiently unique
chapter3_tutorials: notice: target name 'example7' might not be sufficiently unique
chapter3_tutorials: notice: target name 'example4' might not be sufficiently unique
chapter3_tutorials: notice: target name 'example5' might not be sufficiently unique
chapter3_tutorials: notice: target name 'example2' might not be sufficiently unique
chapter3_tutorials: notice: target name 'example3' might not be sufficiently unique
chapter3_tutorials: notice: target name 'example1' might not be sufficiently unique
chapter3_tutorials: notice: target name 'example8' might not be sufficiently unique
chapter3_tutorials: CMakeLists.txt(83): notice: extra arguments in endforeach()
catkin_lint: checked 1 packages and found 9 problems

WARNING: Package name "3dofBringup" does not follow the naming conventions. It should start with a lower case letter and only contain lower case letters, digits and underscores.
WARNING: Package name "3dofRobot" does not follow the naming conventions. It should start with a lower case letter and only contain lower case letters, digits and underscores.
WARNING: Package name "3dofDescription" does not follow the naming conventions. It should start with a lower case letter and only contain lower case letters, digits and underscores.
WARNING: Package name "3dofControllerConfiguration" does not follow the naming conventions. It should start with a lower case letter and only contain lower case letters, digits and underscores.
process[rosout-1]: started with pid [12411]
started core service [/rosout]
process[example6-2]: started with pid [12423]
[ INFO] [1404156269.276210812]: New configuration received with level = 4294967295:
bool   = 1
int    = 0
double = 0
string = Foo
```

Please be aware that you might need to install catkinlint separately; it is usually contained in the package `python-catkin-lint`.

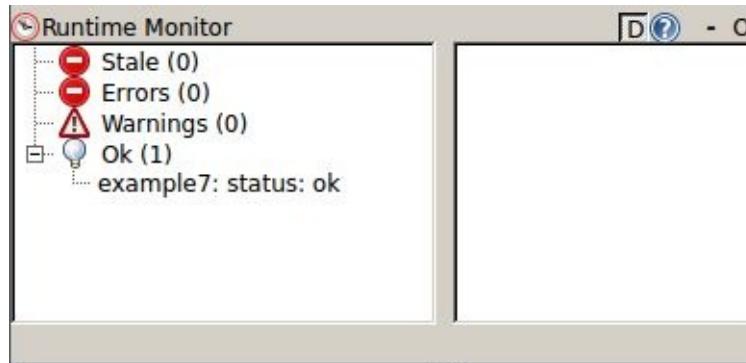
Visualizing nodes diagnostics

ROS nodes can provide diagnostic information using the diagnostics topic. For that, there is an API that helps to publish diagnostic information in a standard way. The information follows the diagnostic_msgs/DiagnosticStatus message type, which allows us to specify a level (OK, WARN, ERROR), name, message, and hardware ID as well as a list of diagnostic_msgs/KeyValue, which are pairs of key and value strings.

The interesting part comes with the tools that collect and visualize this diagnostic information. At the basic level, rqt_runtime_monitor allows us to visualize the information directly published through the diagnostics topic. Run the example7 node, which publishes information through the diagnostics topic, and this visualization tool, to see the diagnostic information:

```
$ rosrun chapter3_tutorials example7.launch  
$ rosrun rqt_runtime_monitor rqt_runtime_monitor
```

The previous commands display the following output:



When the system is large, we can aggregate diagnostic information using the `diagnostic_aggregator`. It processes and categorizes the diagnostics topic messages and republishes them on `diagnostics_agg`. These aggregated diagnostic messages can be visualized with `rqt_robot_monitor`. The diagnostic aggregator is configured with a configuration file, such as the following one (see `config/diagnostic_aggregator.yaml` in `chapter3_tutorials`), where we define different analyzers, in this case using an `AnalyzerGroup`:

```
type: AnalyzerGroup  
path: Sensors  
analyzers:  
status:  
type: GenericAnalyzer  
path: Status  
startswith: example7  
num_items: 1
```

The launch file used in the previous code already runs the `diagnostic_aggregator_node` with the previous configuration, so you can run the following command:

```
$ rosrun rqt_robot_monitor rqt_robot_monitor
```

Now, we can compare the visualization of `rqt_runtime_monitor` with the one of `rqt_robot_monitor`, as shown in

the following screenshot:

The screenshot displays the 'Robot Monitor' application interface. It consists of three main sections: 'Error Device', 'Warned Device', and 'All devices', each with a 'Message' column.

Error Device: Shows one entry: '/Sensors/Status/example7: status error' with a red error icon.

Warned Device: No entries are present.

All devices: Shows two entries under 'All devices': '(Err: 2, Wrn: 0) Sensors Error Error' and 'Status Error example7: status error'. The 'Status' entry has a red error icon.

At the bottom, there is a message bar with '<-- old' on the left, 'Last message received 0 seconds ago' in the center, and 'new -->' on the right. A horizontal bar at the bottom indicates message flow with green and red segments, and a 'Pause' button is located on the far right.

Plotting scalar data

Scalar data can be easily plotted with generic tools already available in ROS. Even non-scalar data can be plotted, but with each scalar field plotted separately. That is why we talk about scalar data, because most non-scalar structures are better represented with ad-hoc visualizers, some of which we will see later; for instance, images, poses, orientation/attitude, and so on.

Creating a time series plot with rqt_plot

Scalar data can be plotted as a time series over the time provided by the timestamps of the messages. Then, in the y axis, we can plot our scalar data. The tool for doing so is `rqt_plot`. It has a powerful argument syntax, which allows you to specify several fields of a structured message in a concise manner; we can also add or remove topics or fields manually from the GUI.

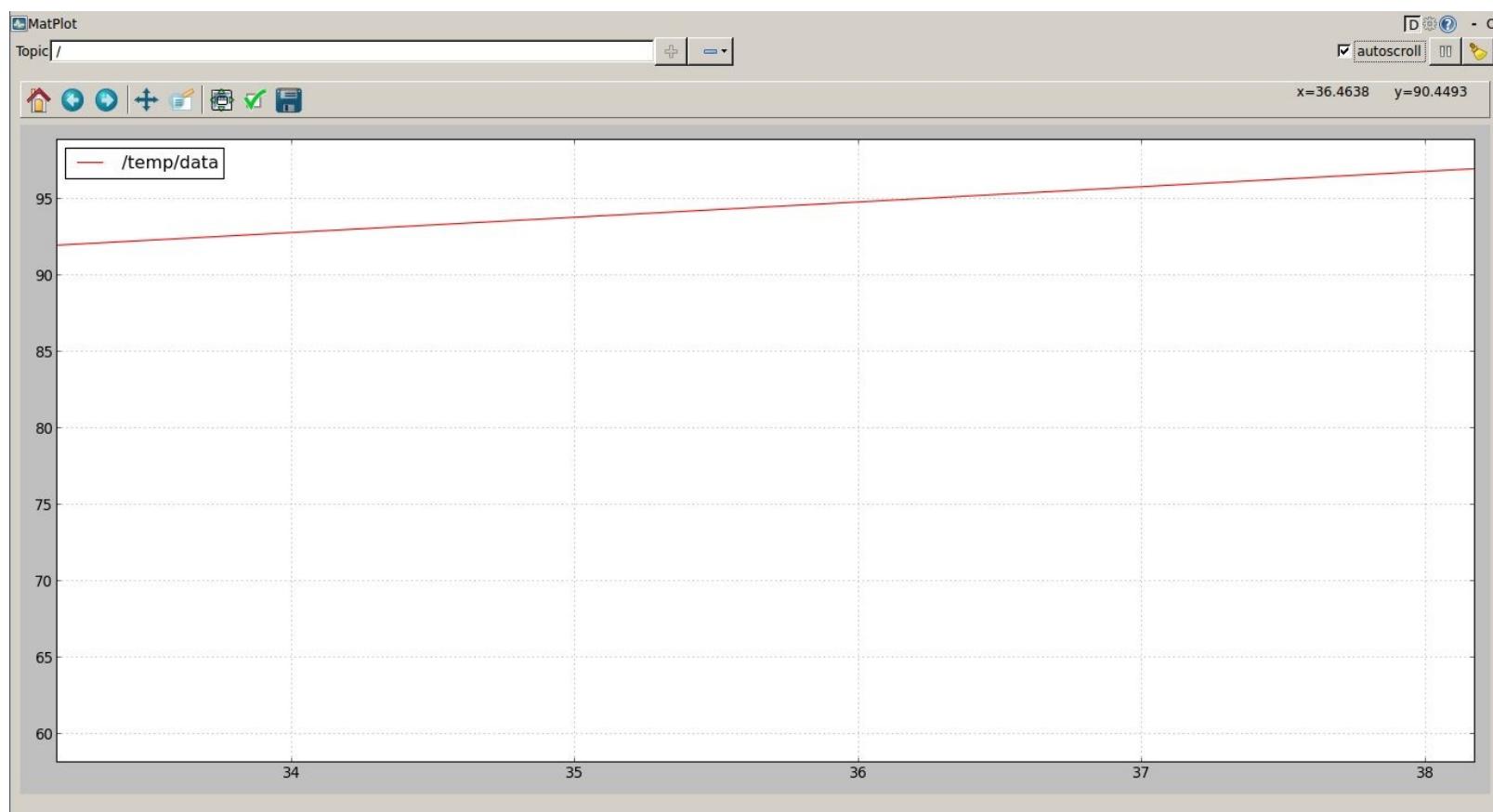
To show `rqt_plot` in action, we are going to use the `example4` node, as it publishes a scalar and a vector (non-scalar) in two different topics, which are `temp` and `accel`, respectively. The values put in these messages are synthetically generated, so they have no actual meaning, but they are useful for our plotting demonstration purposes. Start by running the node using the following command:

```
$ rosrun chapter3_tutorials example4
```

To plot a message, we must know its format; use `rosmsg show <msg type>` if you do not know it. In the case of scalar data, we always have a `data` field that has the actual value. Hence, for the `temp` topic, which is of the `Int32` type, we will run the following command:

```
$ rosrun rqt_plot rqt_plot /temp/data
```

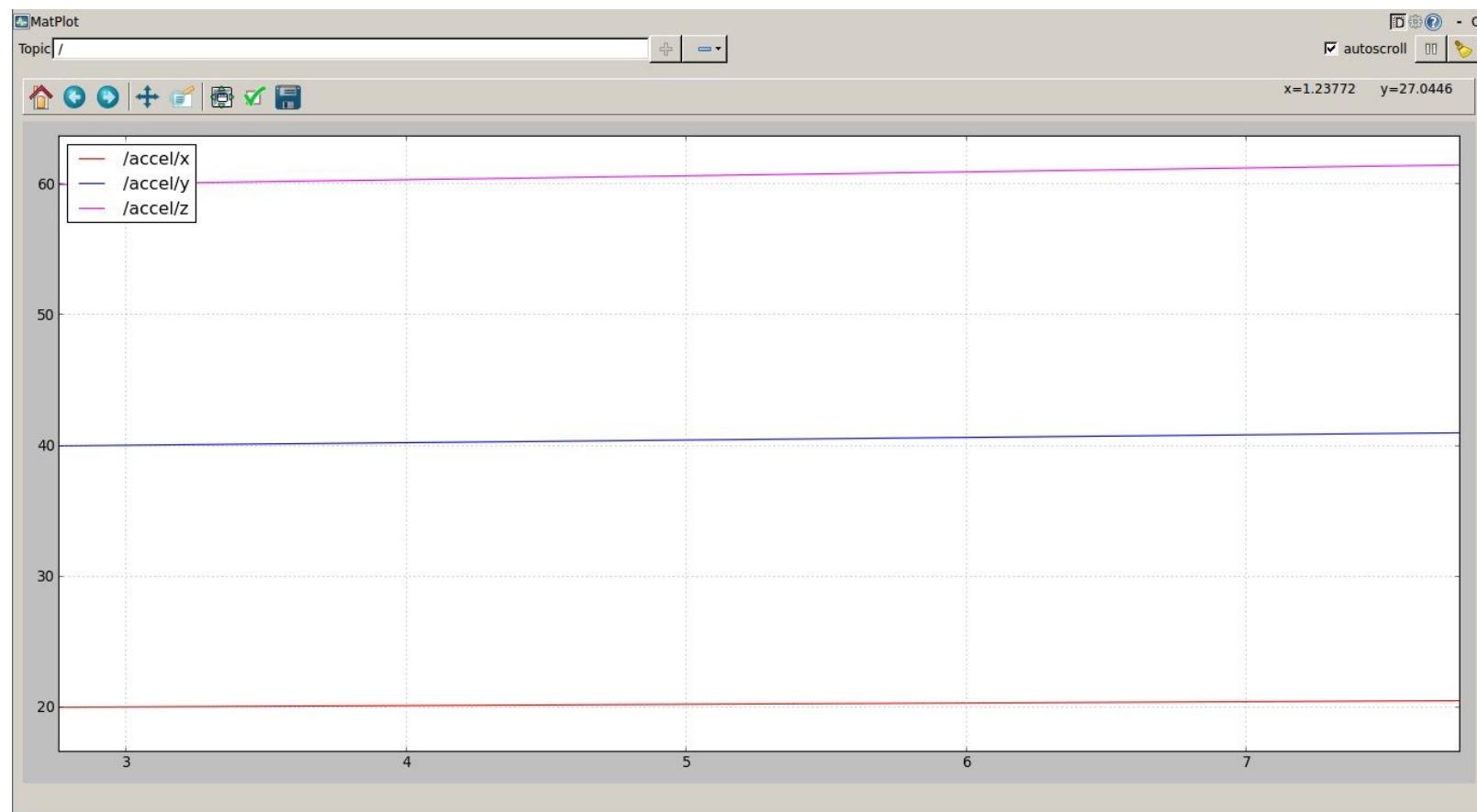
With the node running, we will see a plot that changes over time with the incoming messages, as shown in the following screenshot:



For `accel`, we have a `vector3` message (as you can check with `rostopic type /accel`), which contains three fields that we can visualize in a single plot. The `vector3` message has the x, y, and z fields. We can specify the fields separated by colons (:) or in the more concise manner, as shown in the following command:

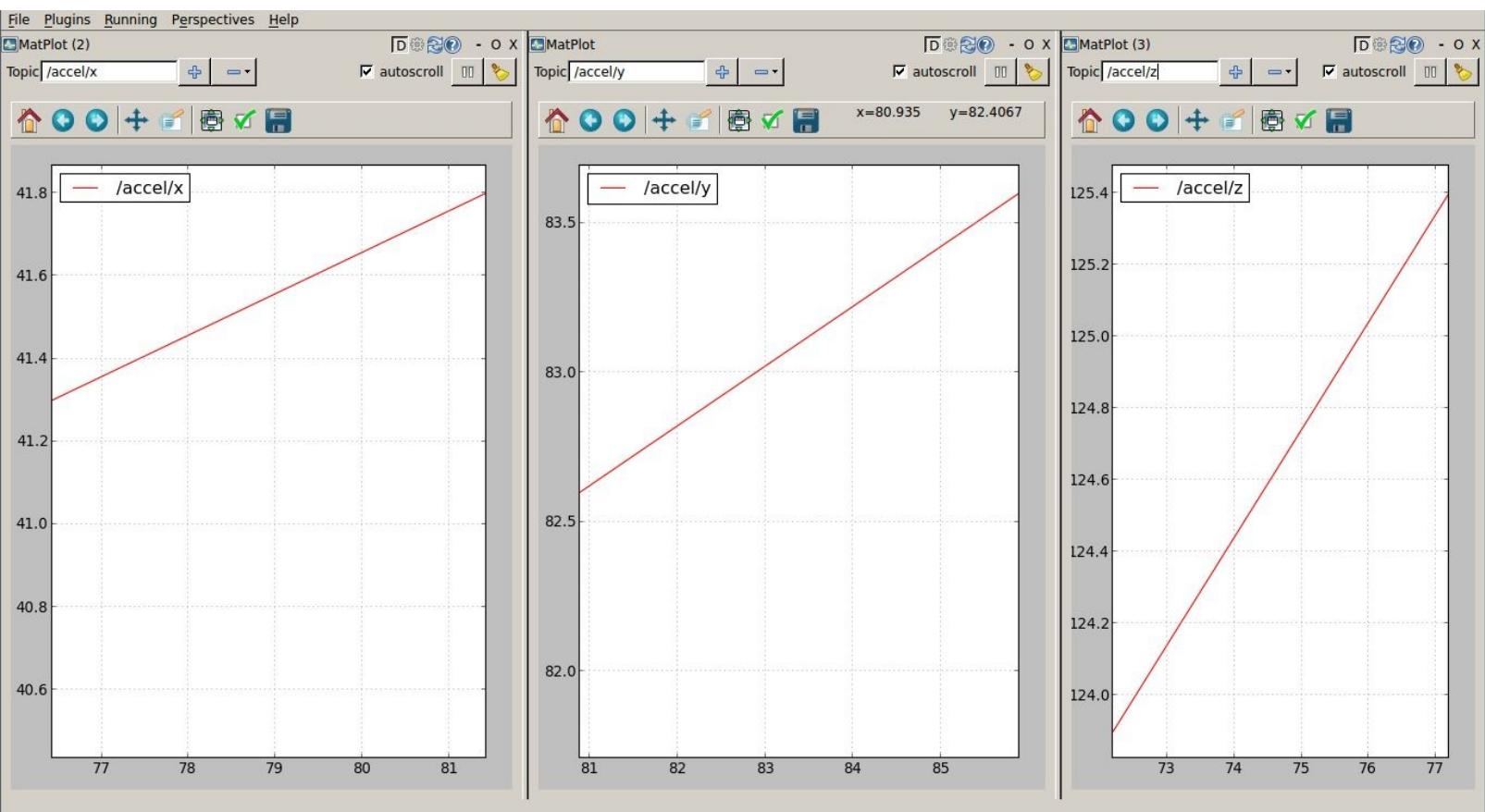
```
$ rosrun rqt_plot rqt_plot /accel/x:y:z
```

The plot should look similar to the one in the following screenshot:



We can also plot each field in a separate axis. However, `rqt_plot` does not support this directly. Instead, we must use `rqt_gui` and arrange three plots manually, as shown in the following command and the screenshot after that:

```
$ rosrun rqt_gui rqt_gui
```



The `rqt_plot` GUI supports three plotting frontends. We can use QT frontends, which are faster and support more time series simultaneously. You can access and select them from the configuration button:

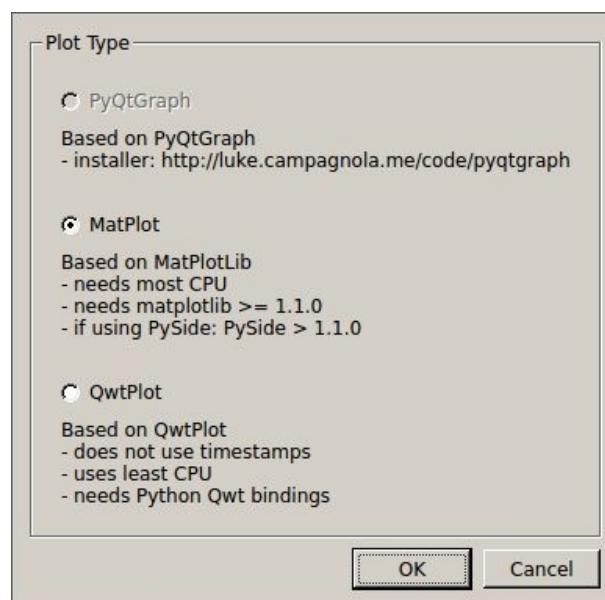


Image visualization

In ROS, we have a node that enables you to display images coming from a camera on-the-fly. This is an example of a topic with complex data, which is better visualized or analyzed with special tools. You only need a camera to do this, such as your laptop webcam. The `example8` node implements a basic camera capture program using OpenCV and ROS bindings to convert `cv::Mat` images into ROS Image messages that can be published in a topic. This node publishes the camera frames in the `/camera` topic.

We are only going to run the node with a launch file created to do so. The code inside the node is still new for the reader, but in the followings chapters we will cover how to work with cameras and images in ROS, so we will be able to come back to this node and understand it:

```
| $ roslaunch chapter3_tutorials example8.launch
```

Once the node is running, we can list the topics (`rostopic list`) and see that the `/camera` topic is there. A straightforward way to verify that we are actually capturing images is to see which frequency we are receiving images at in the topic with `rostopic hz/camera`. It should be something in the region of 30 Hz. This is shown in the following screenshot:

```
~$ rostopic hz /camera
subscribed to [/camera]
average rate: 10.728
    min: 0.084s max: 0.099s std dev: 0.00474s window: 10
average rate: 10.746
    min: 0.084s max: 0.099s std dev: 0.00441s window: 21
average rate: 10.725
    min: 0.084s max: 0.099s std dev: 0.00426s window: 31
average rate: 10.710
    min: 0.084s max: 0.100s std dev: 0.00409s window: 42
average rate: 10.702
    min: 0.084s max: 0.100s std dev: 0.00398s window: 53
[
```

Visualizing a single image

We cannot use `rostopic echo /camera` because, as it's an image, the amount of information in plain text would be huge and not human-readable. Hence, we are going to use the following command:

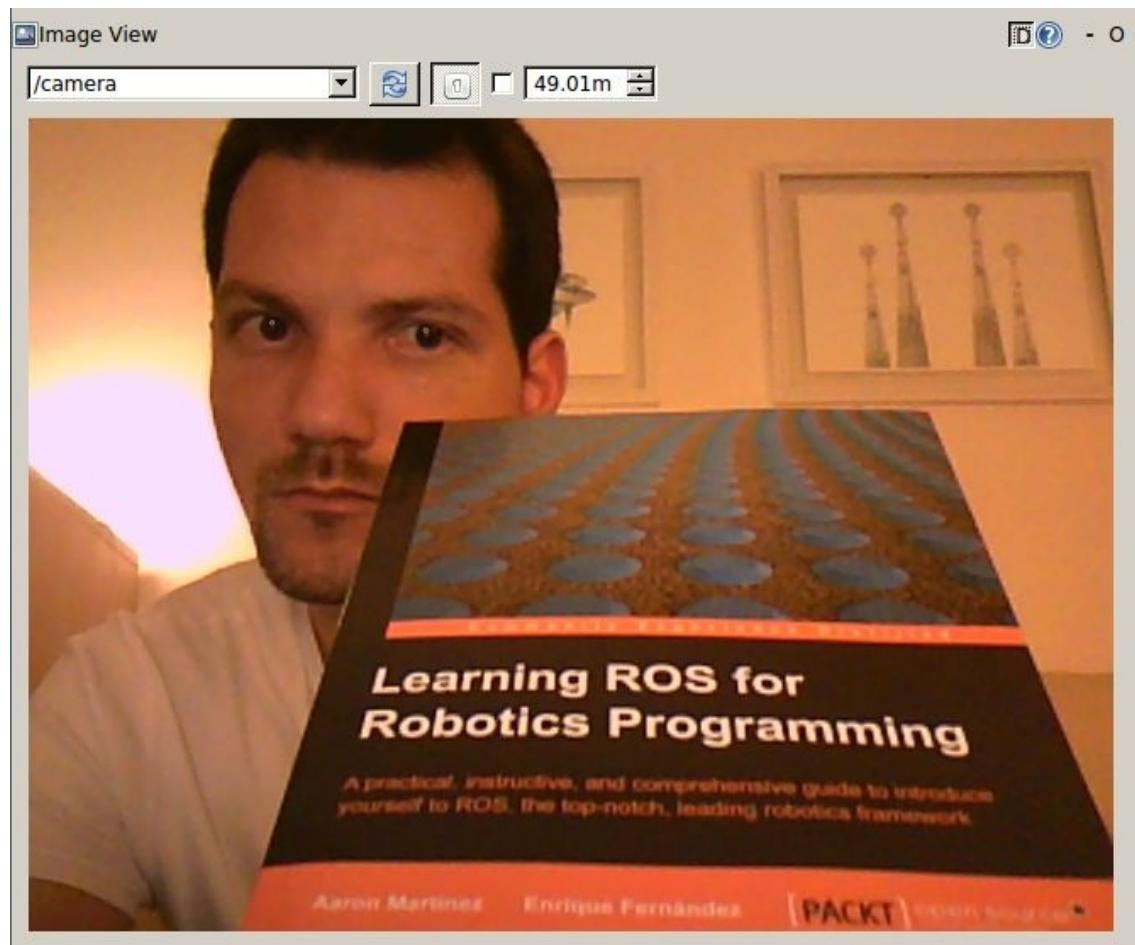
```
$ rosrun image_view image_view image:=/camera
```

This is the `image_view` node, which shows the images in the given topic (the `image` argument) in a window. This way, we can visualize every image or frame published in a topic in a very simple and flexible manner—even over a network. If you press the right button of your mouse in the window, you can save the current frame in the disk, usually in your home directory, or `~/.ros`.

ROS Kinetic also has `rqt_image_view`, which supports viewing multiple images in a single window but does not allow the saving of images by right-clicking. We can select the image topic manually on the GUI or as we do with `image_view`:

```
$ rosrun rqt_image_view rqt_image_view
```

The previous command yields an output shown in the following screenshot:



ROS provides a camera calibration interface built on top of the `opencv` calibration API. We will cover this in *Chapter 8, Computer Vision*, when we will see how to work with cameras. There, we will see

monocular and stereo cameras as well as the ROS image pipeline (`image_proc` and `stereo_image_proc`), which allows the rectification of the camera image distortion and computes the depth image disparity for stereo pairs so that we obtain a point cloud.

3D visualization

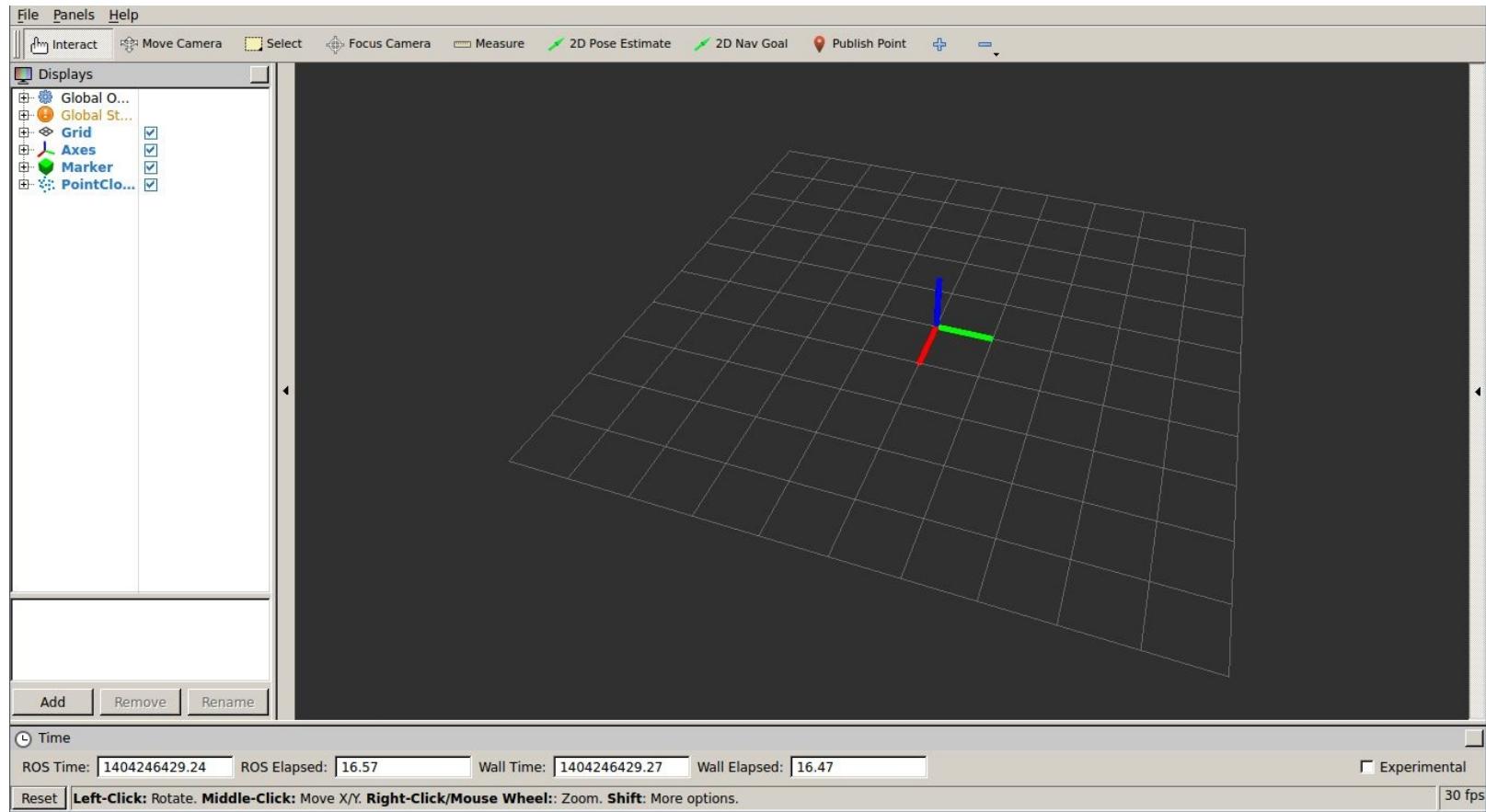
There are certain devices (such as stereo cameras, 3D lasers, the Kinect sensor, and so on) that provide 3D data-usually in the form of point clouds (organized/ordered or not). For this reason, it is extremely useful to have tools that visualize this type of data. In ROS, we have `rviz` or `rqt_rviz`, which integrates an OpenGL interface with a 3D world that represents sensor data in a world representation, using the frame of the sensor that reads the measurements in order to draw such readings in the correct position in respect to each other.

Visualizing data in a 3D world using rqt_rviz

With `roscore` running, start `rqt_rviz` (note that `rviz` is still valid in ROS Kinetic) with:

```
$ rosrun rqt_rviz rqt_rviz
```

We will see the graphical interface of the following screenshot, which has a simple layout:



On the left-hand side, we have the Displays panel, in which we have a tree list of the different elements in the world, which appears in the middle. In this case, we have certain elements already loaded. This layout is saved in the `config/example9.rviz` file, which can be loaded in the File | Open Config menu.

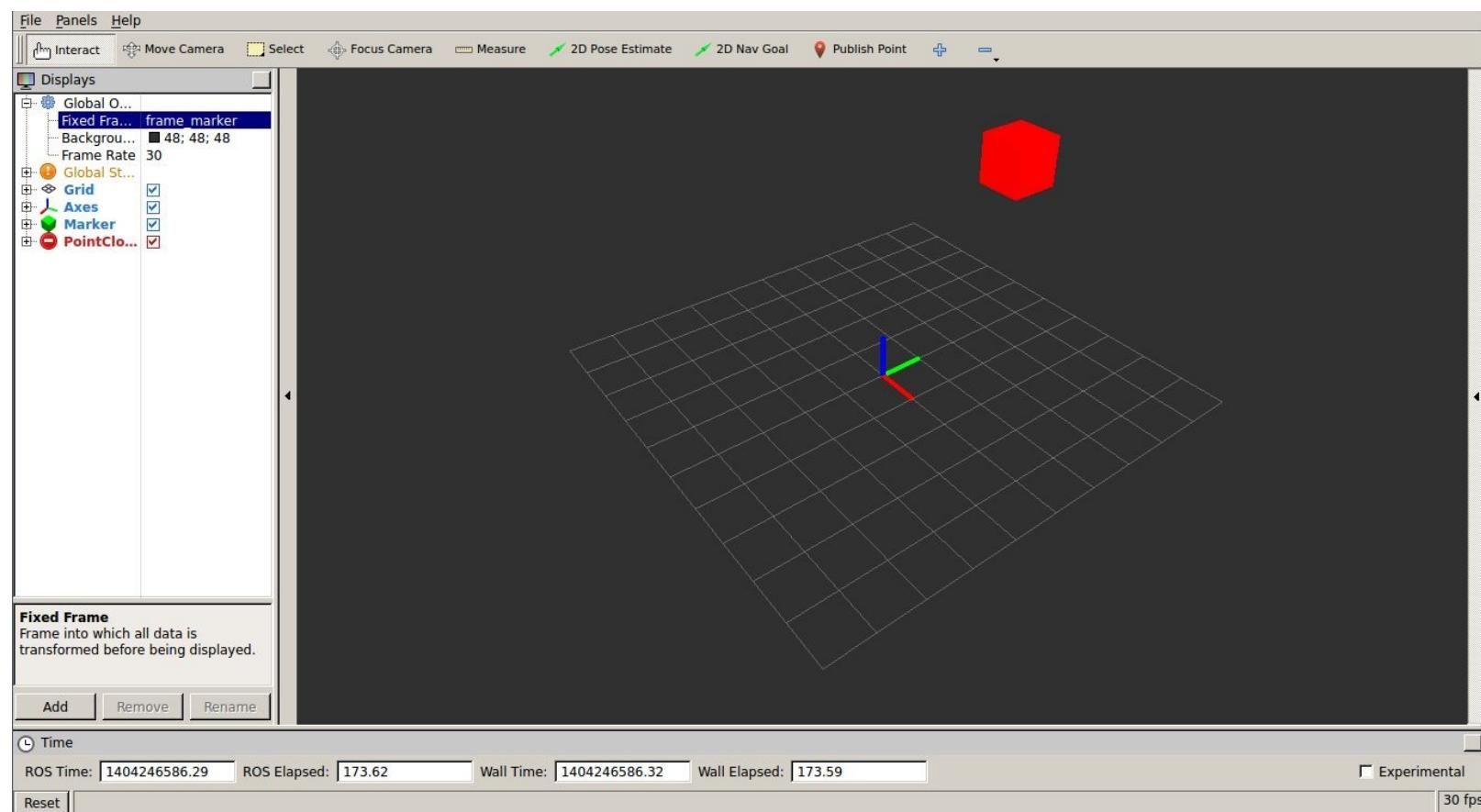
Below the Displays area is the Add button, which allows you to add more elements by topic or type. Also note that there are global options, which are basically tools to set the fixed frame in the world in respect to others. Then, we have Axes and a Grid as a reference for the rest of the elements. In this case, for the `example9` node, we are going to see Marker and PointCloud2.

Finally, on the status bar, we have information regarding the time, and on the right-hand side, there are menus. The Tools properties allows us to configure certain plugin parameters, such as the 2D Nav Goal and 2D Pose Estimate topic names. The Views menu gives different view types, where Orbit and TopDownOrtho are generally enough; one for a 3D view and the other for a 2D top-view. Another menu shows elements selected in the environment. At the top, we also have a menu bar with the current operation mode (Interact, Move Camera, Measure, and so on) and certain plugins.

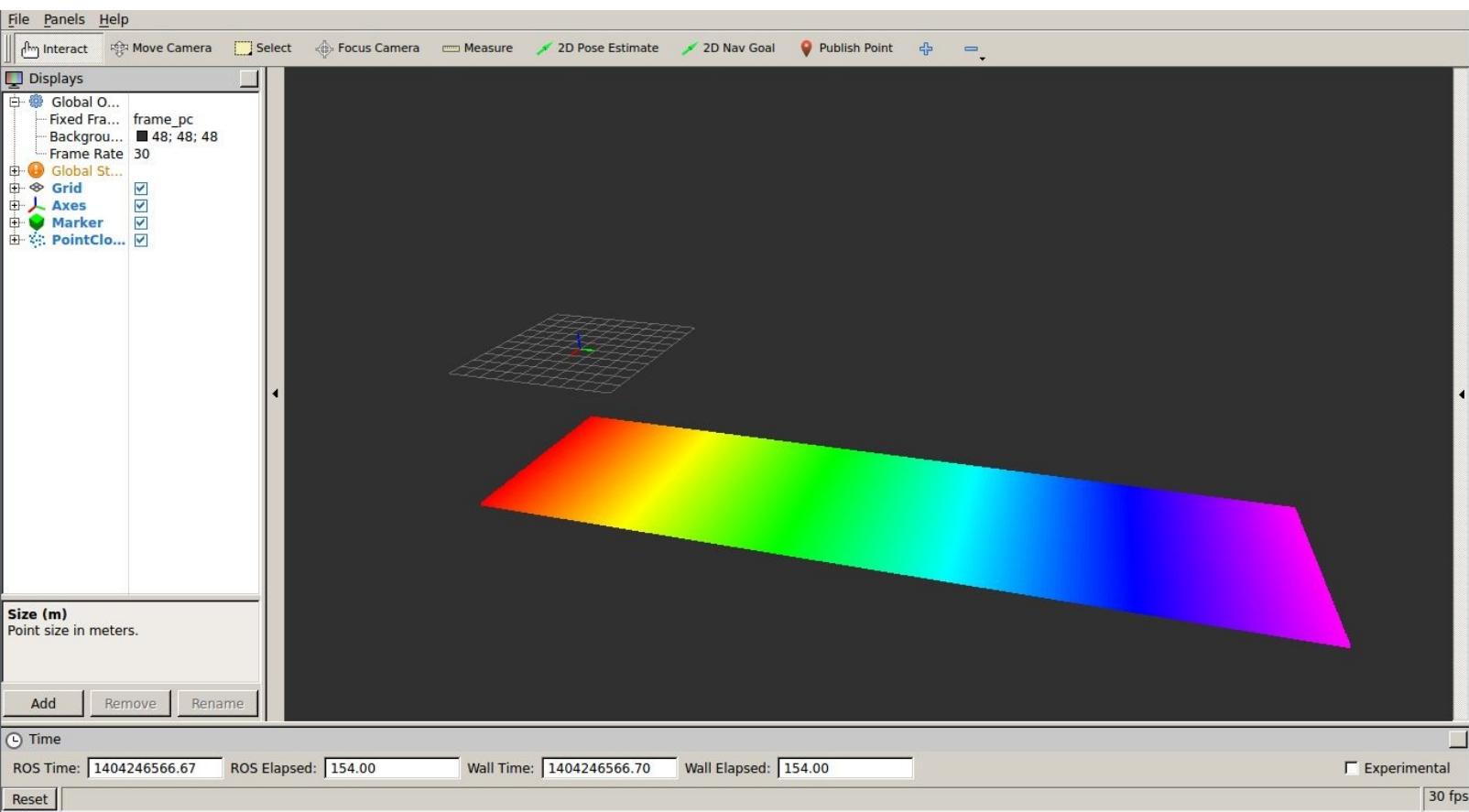
Now we are going to run the `example9` node using the following command:

```
$ roslaunch chapter3_tutorials example9.launch
```

In `rqt_rviz`, we are going to set `frame_id` of the marker, which is `frame_marker`, in the fixed frame. We will see a red cubic marker moving, as shown in the following screenshot:



Similarly, if we set Fixed Frame to `frame_pc`, we will see a point cloud that represents a plane of 200 x 100 points, as shown in the following screenshot:



The list of supported built-in types in `rqt_viz` includes Camera and Image, which are shown in a window similar to `image_view`. In the case of Camera, its calibration is used, and in the case of stereo images, they allow us to overlay the point cloud. We can also see the `LaserScan` data from range lasers, Range cone values from IR/sonar sensors, or `PointCloud2` from 3D sensors, such as the Kinect sensor.

For the navigation stack, which we will cover in following chapters, we have several data types, such as `Odometry` (which plots the robot odometry poses), `Path` (which draws the path plan followed by the robot), `Pose` objects, `PoseArray` for particle clouds with the robot pose estimate, the **Occupancy Grid Map (OGM)** as a Map, and `costmaps` (which are of the Map type in ROS Kinetic and were `GridCell` before ROS Kinetic).

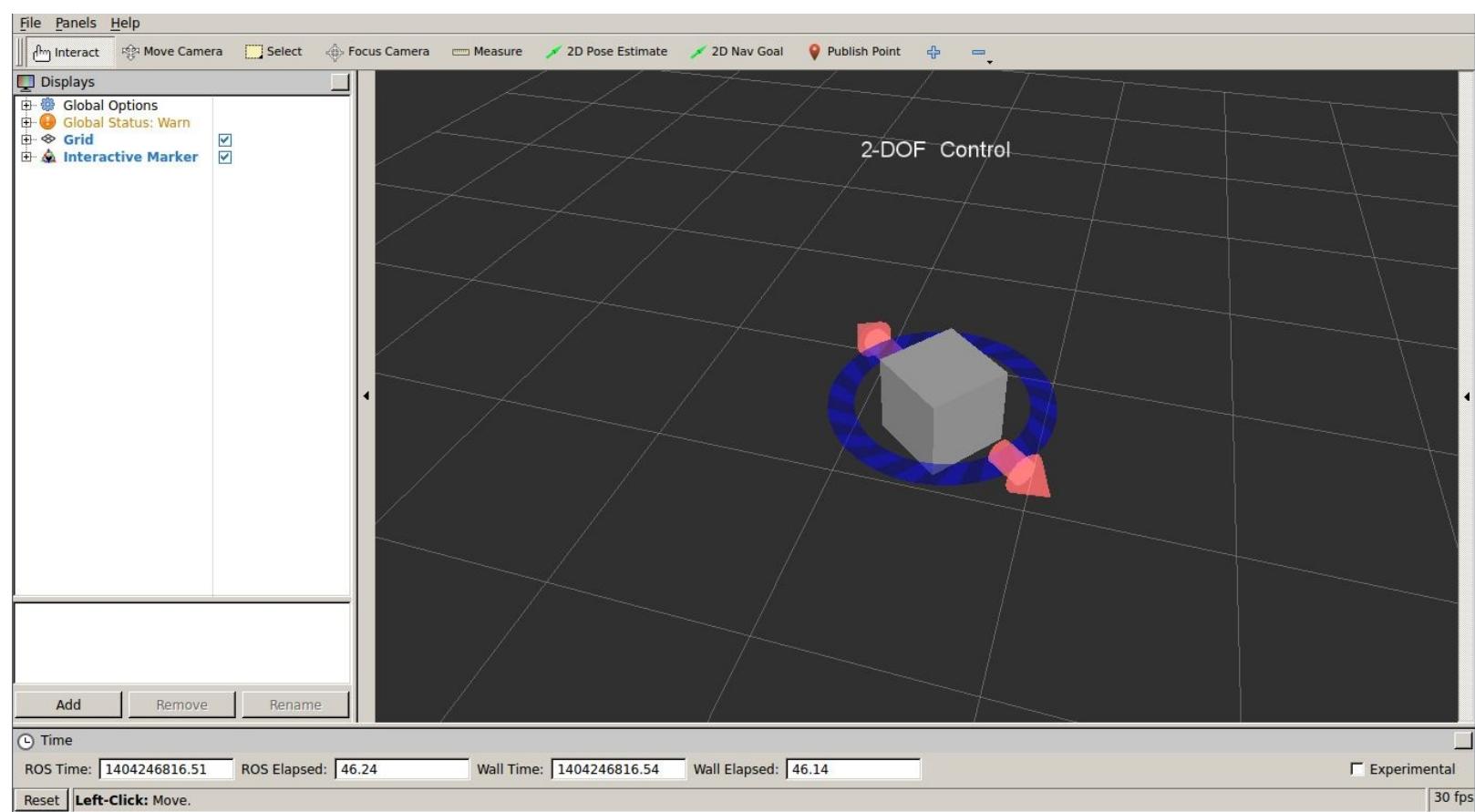
Among other types, it is also worth mentioning the `RobotModel`, which shows the CAD model of all the robot parts, taking the transformation among the frames of each element into account. Indeed, `tf` elements can also be drawn, which is very useful for debugging the frames in the system; we will see an example of this in the following section. In `RobotModel`, we also have the links that belong to the robot **Unified Robot Description Format (URDF)** description with the option to draw a trail showing how they move over time.

Basic elements can also be represented, such as a **Polygon** for the robot footprint; several kind of **Markers**, which support basic geometric elements, such as cubes, spheres, lines, and so on; and even **Interactive Marker** objects, which allow the user to set a pose (position and orientation) in the 3D world. Run the `example8` node to see an example of a simple interactive marker:

```
$ roslaunch chapter3_tutorials example10.launch
```

You will see a marker that you can move in the interactive mode of `rqt_rviz`. Its pose can be used to modify

the pose of another element in the system, such as the joint of a robot:



The relationship between topics and frames

All topics must have a frame if they are publishing data from a particular sensor that has a physical location in the real world. For example, a laser is located in a position with respect to the base link of the robot (usually in the middle of the traction wheels in wheeled robots). If we use the laser scans to detect obstacles in the environment or to build a map, we must use the transformation between the laser and the base link. In ROS, stamped messages have `frame_id`, apart from the timestamp (which is also extremely important when synchronizing different messages). A `frame_id` gives a name to the frame it belongs to.

However, the frames themselves are meaningless; we need the transformation among them. We already have the `tf` frame, which usually has the `base_link` frame as its root (or map if the navigation stack is running). Then, in `rqt_rviz`, we can see how this and other frames move in respect to each other.

Visualizing frame transformations

To illustrate how to visualize frame transformations, we are going to use the `turtlesim` example. Run the following command to start the demonstration:

```
$ roslaunch turtle_tf turtle_tf_demo.launch
```

This is a very basic example with the purpose of illustrating the `tf` visualization in `rqt_rviz`; note that, for the full extent of the capabilities offered by the `tf` API, you should wait until you have reached later chapters of this section. For now, it is enough to know that it allows us to make computations in one frame and then transform them to another, including time delays. It is also important to know that `tf` is published at a certain frequency in the system, so it is like a subsystem, where we can traverse the `tf` tree to obtain the transformation between any frames; we can do this in any node of our system just by consulting `tf`.

If you receive an error, it is probably because the listener died on the launch startup due to another required node that was not ready. If so, run the following command on another terminal to start it again:

```
$ rosrun turtle_tf turtle_tf_listener
```

Now you should see a window with two turtles, one following the other. You can control one of the turtles with the arrow keys as long as your focus is the terminal where you run the launch file. The following screenshot shows how one turtle has been following the other for some time:

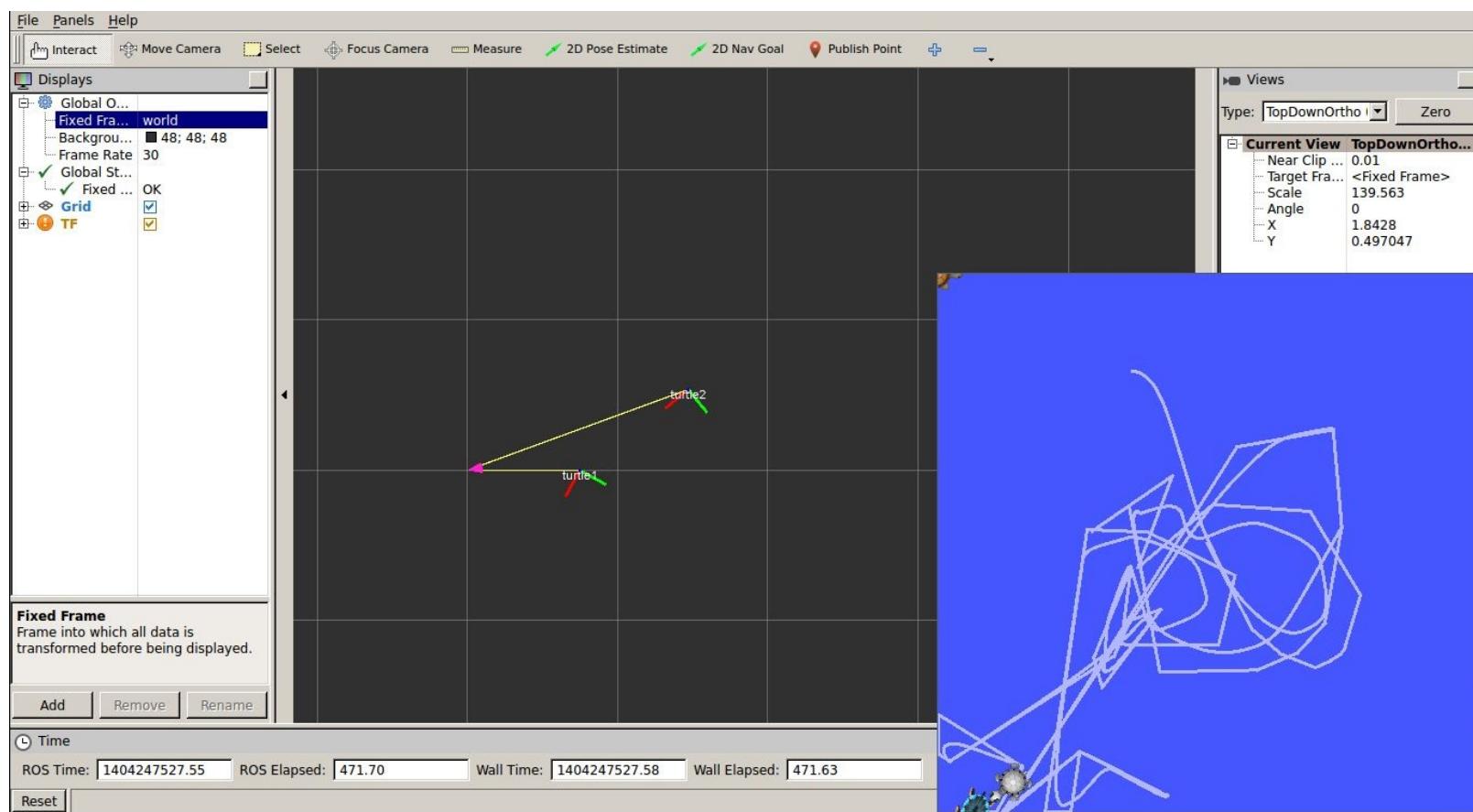


Each turtle has its own frame, so we can see it in `rqt_rviz`:

```
$ rosrun rqt_rviz rqt_rviz
```

Now, instead of viewing the `turtlesim` window, we are going to see how the turtles' frames move in `rqt_rviz` as we move our turtle with the arrow keys. We have to set the fixed frame to `/world` and then add the `tf` tree to the left-hand side area. We will see that we have the `/turtle1` and `/turtle2` frames both as children of the root `/world` frame. In the world representation, the frames are shown as axes and the parent-child links are shown with a yellow arrow that has a pink end. Set the view type to `TopDownOrtho` so it is easier to see how the frames move, as they only move on the 2D ground plane. You might also find it useful to translate the world center, which is done with the mouse with the `Shift` key pressed.

In the following screenshot, you can see how the two turtle frames are shown in respect to the `/world` frame. You can change the fixed frame to experiment with this example and `tf`. Note that `config/example_tf.rviz` is provided to give the basic layout for this example:



Saving and playing back data

Usually, when we work with robotic systems, we have to deal with our resources either being shared or not always available, or with the fact that experiments cannot be done regularly because of the cost or time required for preparing and performing them. For this reason, it is good practice to record the data of the experiment session for later analysis and to work, develop, and test our algorithms. However, the process of saving good data so that we can reproduce the experiment offline is not trivial. Fortunately, in ROS we have powerful tools that have already solved this problem.

ROS can save all the messages published on any topic. It has the ability to create a bag file containing the messages as they are, with all their fields and timestamps. This allows for the reproduction of the experiment offline, with its real conditions on the robot as the latency of message transmissions. What's more, ROS tools do all this efficiently, with a high bandwidth, and in an adequate manner, to organize the saved data.

In the following section, we will explain the tools provided by ROS to save and play back data stored in bag files, which use a binary format designed for and by ROS developers. We will also see how to manage these files, that is, inspect the contents (number of messages, topics, and so on), compress them, and split or merge several of them.

What is a bag file?

A bag file is a container of messages sent by topics that were recorded during a session using a robot or nodes. In brief, they are the log files for the messages transferred during the execution of our system, and they allow us to play back everything, even with time delays, since all messages are recorded with a timestamp-not only the timestamp in the header, but also for the packets contained within the bag file. The difference between the timestamp used to record and the one in the header is that the first one is set once the message is recorded, while the other is set by the producer/publisher of the message.

The data stored in a bag file is in the binary format. The particular structure of this container allows for extremely fast recording throughput, which is the most important concern when saving data. Additionally, the size of the bag file is relevant, but usually at the expense of speed. Nonetheless, we have the option to compress the file on the fly with the `bz2` algorithm; just use the `-j` parameter when you record with `rosbag record`.

Every message is recorded along with the topic that published it. Therefore, we can specify which topics to record or we can record them all (with `-a`). Later, when we play the bag file back, we can also select a particular subset of the topics contained in the bag file by indicating the name of the topics we want to publish.

Recording data in a bag file with rosbag

The first thing we have to do is simply record some data. We are going to use a very simple system as an example-our `example4` node. Hence, we will first run the node:

```
$ rosrun chapter3_tutorials example4
```

Now, we have two options. First, we can record all the topics with the following command:

```
$ rosbag record -a
```

Otherwise, we can record only specific topics. In this case, it makes sense to record only the `example4` topics, so we will run the following command:

```
$ rosbag record /temp /accel
```

By default, when we run the previous command, the `rosbag` program subscribes to the node and starts recording the message in a bag file in the current directory with the date as the name. Once the experiment has been completed, you only need to hit Ctrl + C in the running terminal. The following is an example of a recording session and the resulting bag file:

```
[ INFO] [1404248014.668263731]: Subscribing to /temp
[ INFO] [1404248014.671339658]: Subscribing to /accel
[ INFO] [1404248014.674950564]: Recording to 2014-07-01-22-54-34.bag.
```

You can find more options with `rosbag help record`, which include the bag file size, the duration of the recording, options to split the files into several smaller files of a given size, and so on. As we mentioned before, the file can be compressed on the fly (the `-j` option). This is only useful for small data rate recording, as it also consumes CPU time and might end up dropping messages. If messages are dropped, we can increase the buffer (`-b`) size for the recorder in MB, which defaults to 256 MB, although it can be increased to a couple of GB if the data rate is very high (especially with images).

It is also possible to include the call to `rosbag record` into a launch file, which is useful when we want to set up a recorder for certain topics. To do so, we must add the following node:

```
<node pkg="rosbag" type="record" name="bag_record"
args="/temp /accel"/>
```

Note that topics and other arguments are passed to the command using the `args` argument. It is also important to note that when `rosbag` is run from the launch file, the bag file is created by default in `~/.ros`, unless we give the name of the file with `-o` (prefix) or `-O` (full name).

Playing back a bag file

Now that we have a bag file recorded, we can use it to play back all the messages for the topics contained in it; note that we need `roscore` running and nothing else. We will move to the folder in which the bag file is located (there are two examples in the `bag` folder of this chapter's tutorials) and run this command:

```
$ rosbag play 2014-07-01-22-54-34.bag
```

Hopefully, `rosbag` will produce the following output:

```
[ INFO] [1404248314.594700096]: Opening 2014-07-01-22-54-34.bag
Waiting 0.2 seconds after advertising topics... done.
Hit space to toggle paused, or 's' to step.
[RUNNING] Bag Time: 1404248078.757944 Duration: 2.801764 / 39.999515
```

In the terminal in which the bag file is being played, we can pause (hit *spacebar*) or move step by step (hit *S*). As usual, press *Ctrl + C* to finish the execution. Once we have reached the end of the file, it will close; however, there is an option to loop (*-l*), which may sometimes be useful.

We will see the topics with `rostopic list` automatically, as follows:

```
/accel
/clock
/rosout
/rosout_agg
/temp
```

The `/clock` topic appears because we can simulate the system clock to produce a faster playback. This can be configured using the `-r` option. The `/clock` topic publishes the time for the simulation at a configurable frequency with the `--hz` argument (it defaults to 100 Hz).

We can also specify a subset of the topics in the file to be published. This is done with the `--topics` option.

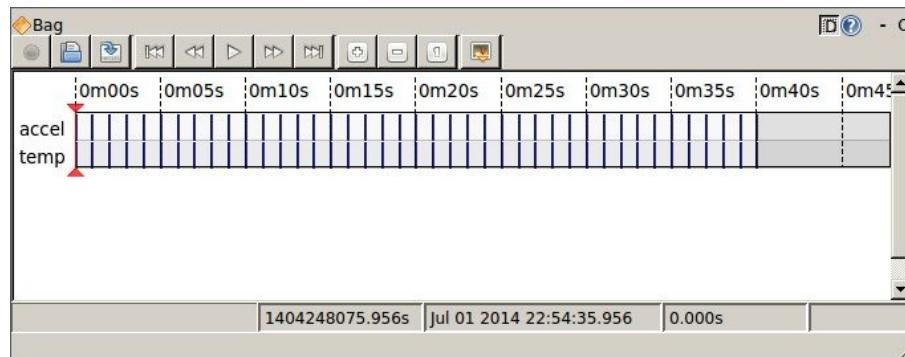
Inspecting all the topics and messages in a bag file

There are two main ways to inspect the contents of a bag file. The first one is very simple: we just type `rosbag info <bag_file>`, and the result is something similar to the one shown in the following screenshot:

```
$ rosbag info 2014-07-01-22-54-34.bag
path:      2014-07-01-22-54-34.bag
version:   2.0
duration:  40.0s
start:    Jul 01 2014 22:54:35.96 (1404248075.96)
end:     Jul 01 2014 22:55:15.96 (1404248115.96)
size:    10.9 KB
messages: 82
compression: none [1/1 chunks]
types:    geometry_msgs/Vector3 [4a842b65f413084dc2b10fb484ea7f17]
          std_msgs/Int32      [da5909fbe378aeaf85e547e830cc1bb7]
topics:   /accel  41 msgs   : geometry_msgs/Vector3
          /temp   41 msgs   : std_msgs/Int32
```

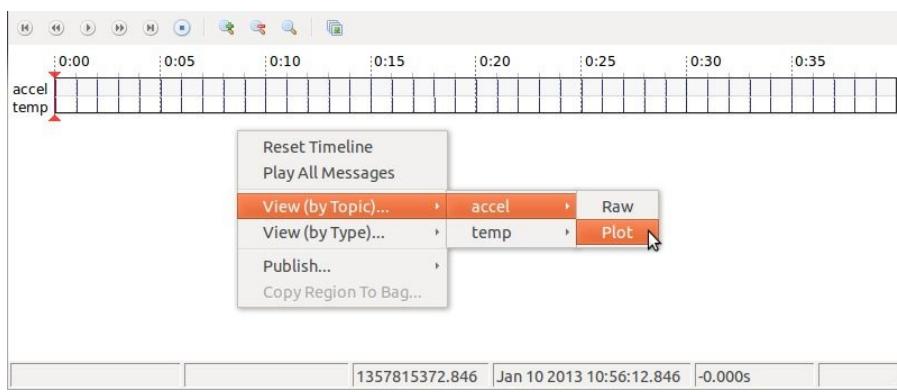
We have information about the bag file itself, such as the creation date, duration, size, the number of messages, and the compression (if any). Then, we have the list of data types inside the file, and finally the list of topics with their corresponding name, number of messages, and type.

The second way to inspect a bag file is extremely powerful. It is a GUI called `rqt_bag` which also allows you to play back the files, view the images (if any), plot scalar data, and also view the Raw structure of the messages; it is `rxbag` behavior's replacement. We only have to pass the name of the bag file to see something similar to the following screenshot (for the previous bag file):

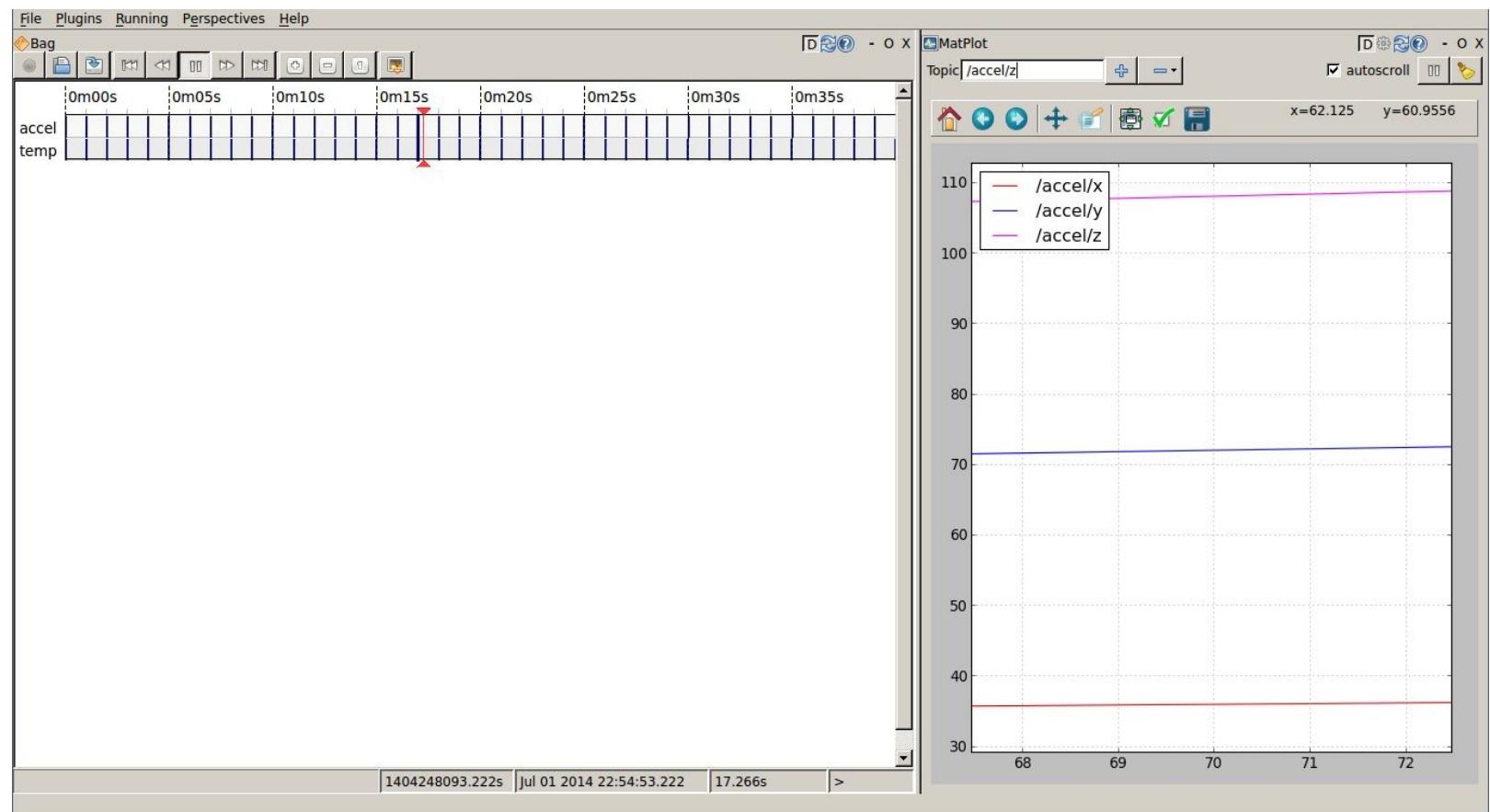


We have a timeline for all the topics where each message appears with a mark. If the bag file we're inspecting contains images, we can enable thumbnails to see them in the timeline.

In the following screenshot, we can see how to access the Raw, Plot, and Image (if the topic is of the Image type) views for the topics in the file. This pop-up menu appears with a right-click over the timeline:



As an alternative, we can use `rqt_gui` and put the `rqt_bag` and `rqt_plot` plugins in the same window; the layout of the following screenshot can be imported from the perspective given in the `config/bag_plot.perspective` folder. However, we have to use Publish All and play to actually see the plot, which differs from the `rxbag` behavior. For `/accel`, we can plot all the fields in a single axis. To do so, once we have the plot view, we add each field by pressing the + button/icon. Note that we can remove them later or create different axes. As mentioned previously, the plot is not generated for all the values in the file, rather it simply shows the data that is played back and published:



Remember that with the `rxbag` behavior, we must press the play button at least once to be able to plot the data. Then we can play, pause, stop, and move to the beginning or the end of the file. The images are straightforward, and a simple window appears with the current frame with options to save them as image files.

Using the rqt_gui and rqt plugins

Since **ROS Fuerte**, the `rx` applications or tools have been deprecated, so we should use the `rqt` nodes instead. They are basically the same, and only a few of them incorporate small updates, bug fixes, and new features. The following table shows the equivalences for the tools shown in this chapter (the ROS Kinetic `rqt` tool and the one it replaces from previous ROS distributions):

| ROS Kinetic rqt tool | Replaces (ROS Fuerte or before) |
|---|---|
| <code>rqt_console</code> and <code>rqt_logger_level</code> | <code>rxconsole</code> |
| <code>rqt_graph</code> | <code>rxgraph</code> |
| <code>rqt_reconfigure</code> and <code>rqt_reconfigure</code> | <code>dynamic_reconfigurerereconfigure_gui</code> |
| <code>rqt_plot</code> | <code>rxplot</code> |
| <code>rqt_image_view</code> | <code>image_view</code> |
| <code>rqt_bag</code> | <code>rxbag</code> |

In ROS Kinetic, there are even more standalone plugins, such as a shell (`rqt_shell`), a topic publisher (`rqt_publisher`), a message type viewer (`rqt_msg`), and many more (the most important ones have been covered in this chapter). Even `rqt_viz` is a plugin, which replaces `rviz`, which can also be integrated into the new `rqt_gui` interface. We can run this GUI and add and arrange several plugins manually on the window, as it has been seen in several examples in this chapter:

```
$ rosrun rqt_gui rqt_gui
```

Summary

After reading and running the code of this chapter, you will have learned how to use many tools that will help you to develop robotic systems faster, debug errors, and visualize your results, so as to evaluate their quality or validate them. Some of the specific concepts and tools you will require in your life as a robotics developer have been summarized here.

Now you know how to include logging messages in your code with different levels of verbosity, which will help you to debug errors in your nodes. For this purpose, you can also use the powerful tools included in ROS, such as the `rqt_console` interface. Additionally, you can inspect or list the nodes running, topics published, and services provided in the whole system. This includes the inspection of the node graph using `rqt_graph`.

Regarding the visualization tools, you should now be able to plot scalar data using `rqt_plot` for a more intuitive analysis of certain variables published by your nodes. Similarly, you can view more complex types (non-scalar ones). This includes images and 3D data using `rqt_image_view` and `rqt_rviz`, respectively. Similarly, you can use tools to calibrate and rectify camera images.

Finally, you are now able to record and play back messages in a session thanks to `rosbag`. You have also learned how to view the contents of a bag file with `rqt_bag`. This allows you to record the data from your experiments and process them later with your AI or robotics algorithms.

The Navigation Stack - Robot Setups

In this chapter, you will learn what is probably one of the most powerful features in ROS, something that will let you move our robot autonomously.

Thanks to the community and the shared code, ROS has many algorithms that can be used for navigation.

First of all, in this chapter, you will learn all the necessary ways to configure the navigation stack with your robot. In the next chapter, you will learn how to configure and launch the navigation stack on the simulated robot, giving goals and configuring some parameters to get the best results. In particular, we will cover the following topics in this chapter:

- Introduction to the navigation stacks and their powerful capabilities-clearly one of the greatest pieces of software that comes with ROS.
- The `tf` library-showing the transformation of one physical element to the other from the frame; for example, the data received using a sensor or the command for the desired position of an actuator. `tf` is a library for keeping track of the coordinate frames.
- Creating a laser driver or simulating it.
- Computing and publishing the odometry and how this is provided by Gazebo.
- Base controllers and creating one for your robot.
- Executing **Simultaneous Localization and Mapping (SLAM)** with ROS-building a map from the environment with your robot as it moves through it. Localizing your robot in the map using the **Adaptive Monte Carlo Localization (AMCL)** algorithm of the navigation stack. AMCL is a probabilistic localization system for a robot moving in 2D. It implements the AMCL approach, which uses a particle filter to track the pose of a robot against a known map.

The navigation stack in ROS

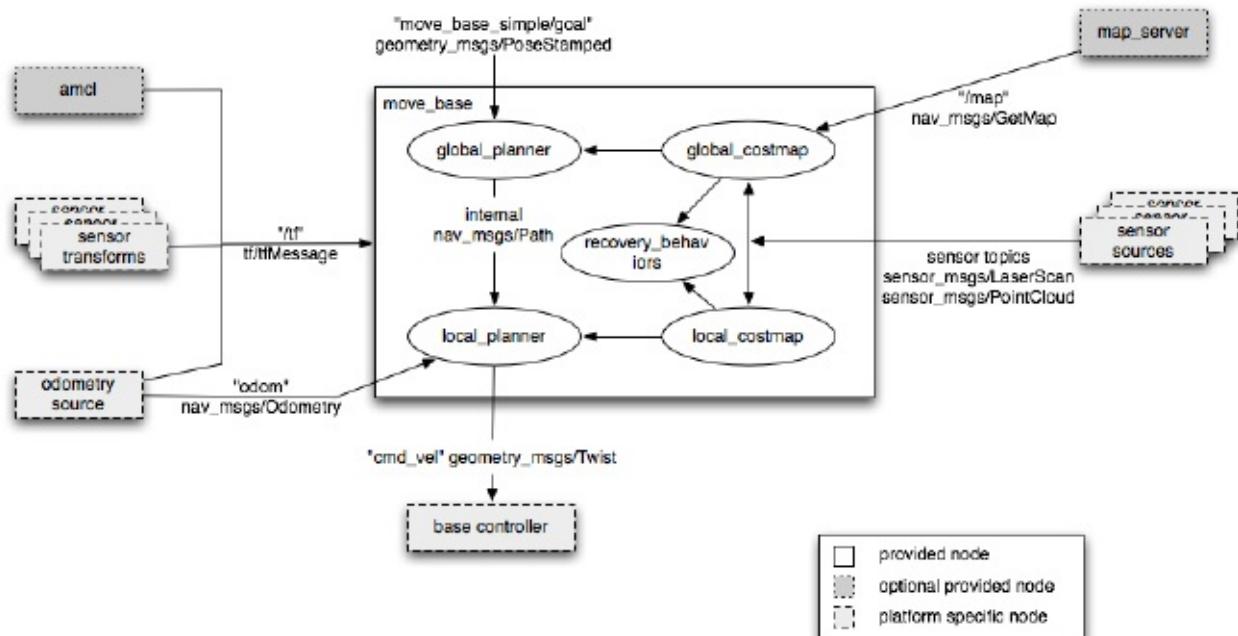
In order to understand the navigation stack, you should think of it as a set of algorithms that use the sensors of the robot and the odometry so that you can control the robot using a standard message. It can move your robot without any problems, such as crashing, getting stuck in a location, or getting lost to another position.

You would assume that this stack can be easily used with any robot. This is almost true, but it is necessary to tune some configuration files and write some nodes to use the stack.

The robot must satisfy some requirements before it uses the navigation stack:

- The navigation stack can only handle a differential drive and holonomic-wheeled robots. The shape requisites of the robot must either be a square or a rectangle. However, it can also do certain things with biped robots, such as robot localization, as long as the robot does not move sideways.
- It requires that the robot publishes information about the relationships between the positions of all the joints and sensors.
- The robot must send messages with linear and angular velocities.
- A planar laser must be on the robot to create the map and localization. Alternatively, you can generate something equivalent to several lasers or a sonar, or you can project the values to the ground if they are mounted at another place on the robot.

The following diagram shows you how the navigation stacks are organized. You can see three groups of boxes with colors (gray and white) and dotted lines. The plain white boxes indicate the stacks that are provided by ROS, and they have all the nodes to make your robot really autonomous:



In the following sections, we will see how to create the parts marked in gray in the diagram. These parts depend on the platform used; this means that it is necessary to write code to adapt the platform to be used

in ROS and to be used by the navigation stack.

Creating transforms

The navigation stack needs to know the position of the sensors, wheels, and joints.

To do that, we use the **Transform Frames (tf)** software library. It manages a transform tree. You could do this with mathematics, but if you have a lot of frames to calculate, it will be a bit complicated and messy.

Thanks to `tf`, we can add more sensors and parts to the robot, and `tf` will handle all the relations for us.

If we put the laser 10 cm backwards and 20 cm above with reference to the origin of the `base_link` coordinates, we would need to add a new frame to the transformation tree with these offsets.

Once inserted and created, we could easily know the position of the laser with reference to the `base_link` value or the wheels. The only thing we need to do is call the `tf` library and get the transformation.

Creating a broadcaster

Let's test this with a simple code. Create a new file in `chapter5_tutorials/src` with the name `tf_broadcaster.cpp`, and put the following code inside it:

```
#include <ros/ros.h>
#include <tf/transform_broadcaster.h>

int main(int argc, char** argv){
    ros::init(argc, argv, "robot_tf_publisher");
    ros::NodeHandle n;

    ros::Rate r(100);

    tf::TransformBroadcaster broadcaster;

    while(n.ok()){
        broadcaster.sendTransform(
            tf::StampedTransform(
                tf::Transform(tf::Quaternion(0, 0, 0, 1), tf::Vector3(0.1,
                    0.0, 0.2)),
                ros::Time::now(), "base_link", "base_laser"));
        r.sleep();
    }
}
```

Remember to add the following line in your `CMakeList.txt` file to create the new executable:

```
add_executable(tf_broadcaster src/tf_broadcaster.cpp)
target_link_libraries(tf_broadcaster ${catkin_LIBRARIES})
```

We also create another node that will use the transform, and which will give us the position of a point on the sensor with reference to the center of `base_link` (our robot).

Creating a listener

Create a new file in `chapter5_tutorials/src` with the name `tf_listener.cpp` and input the following code:

```
#include <ros/ros.h>
#include <geometry_msgs/PointStamped.h>
#include <tf/transform_listener.h>

void transformPoint(const tf::TransformListener& listener){
    //we'll create a point in the base_laser frame that we'd like to
    transform to the base_link frame
    geometry_msgs::PointStamped laser_point;
    laser_point.header.frame_id = "base_laser";

    //we'll just use the most recent transform available for our
    simple example
    laser_point.header.stamp = ros::Time();

    //just an arbitrary point in space
    laser_point.point.x = 1.0;
    laser_point.point.y = 2.0;
    laser_point.point.z = 0.0;

    geometry_msgs::PointStamped base_point;
    listener.transformPoint("base_link", laser_point, base_point);

    ROS_INFO("base_laser: (%.2f, %.2f, %.2f) -----> base_link: (%.2f,
    %.2f, %.2f) at time %.2f",
    laser_point.point.x, laser_point.point.y, laser_point.point.z,
    base_point.point.x, base_point.point.y, base_point.point.z,
    base_point.header.stamp.toSec());
    ROS_ERROR("Received an exception trying to transform a point from
    \"base_laser\" to \"base_link\": %s", ex.what());
}

int main(int argc, char** argv){
    ros::init(argc, argv, "robot_tf_listener");
    ros::NodeHandle n;

    tf::TransformListener listener(ros::Duration(10));

    //we'll transform a point once every second
    ros::Timer timer = n.createTimer(ros::Duration(1.0),
    boost::bind(&transformPoint, boost::ref(listener)));

    ros::spin();
}
```

Remember to add the line in the `CMakeList.txt` file to create the executable. Compile the package and run both the nodes using the following commands in each terminal:

```
$ catkin_make
$ rosrun chapter5_tutorials tf_broadcaster
$ rosrun chapter5_tutorials tf_listener
```

Remember, always run `roscore` before starting with the examples. You will see the following message:

```
[ INFO] [1368521854.336910465]: base_laser: (1.00, 2.00, 0.00) -----> base_link: (1.10, 2.00, 0.20) at time
1368521854.33
[ INFO] [1368521855.336347545]: base_laser: (1.00, 2.00, 0.00) -----> base_link: (1.10, 2.00, 0.20) at time
1368521855.33
```

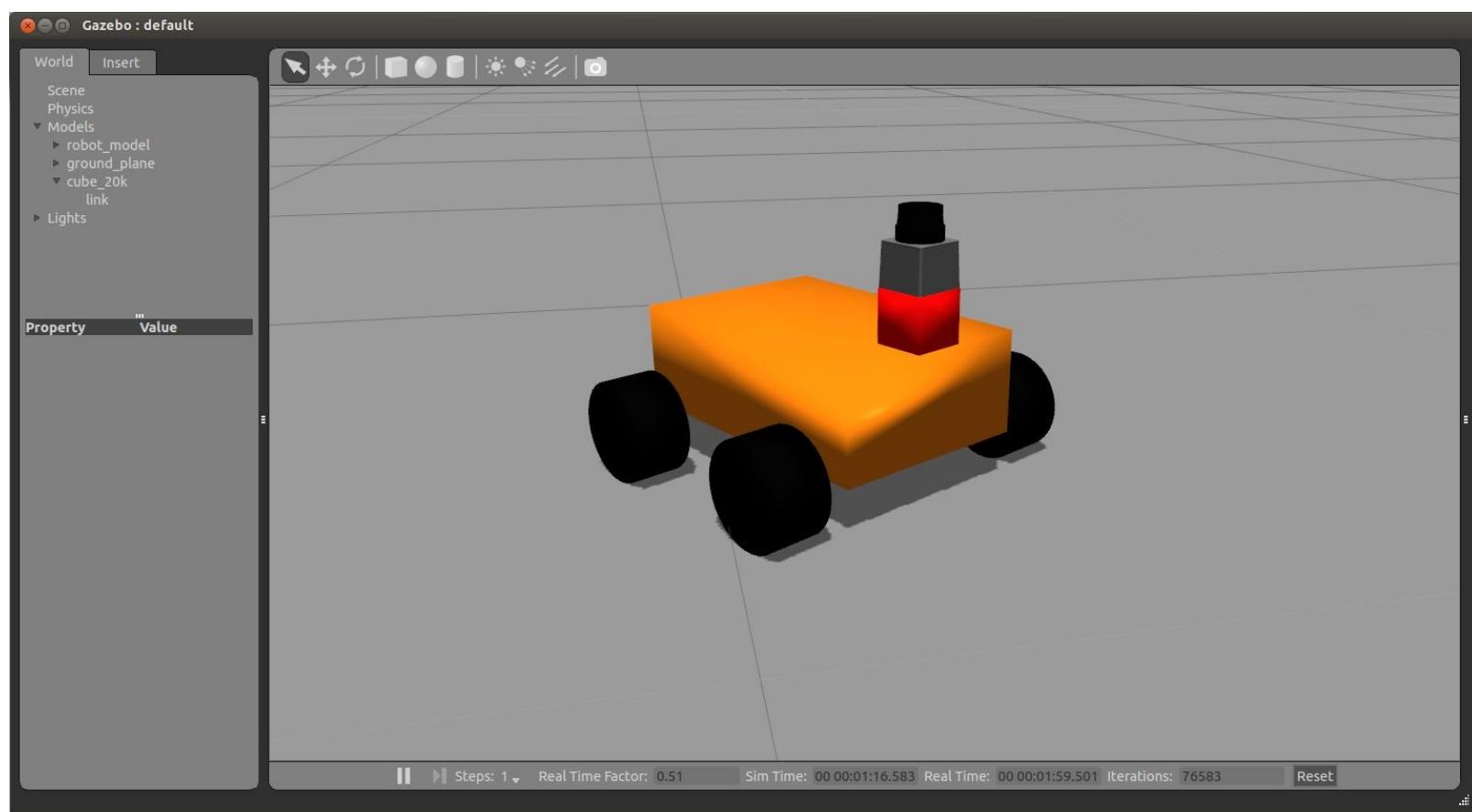
This means that the point that you published on the node, with the position `(1.00, 2.00, 0.00)` relative to

`base_laser`, has the position (1.10, 2.00, 0.20) relative to `base_link`.

As you can see, the `tf` library performs all the mathematics for you to get the coordinates of a point or the position of a joint relative to another point.

A transform tree defines offsets in terms of both translation and rotation between different coordinate frames. Let us see an example to help you understand this.

In our robot model, we are going to add another laser, say, on the back of the robot (`base_link`):



The system in our robot had to know the position of the new laser to detect collisions, such as the one between the wheels and walls. With the `tf` tree, this is very simple to do and maintain, apart from being scalable. Thanks to `tf`, we can add more sensors and parts, and the `tf` library will handle all the relations for us. All the sensors and joints must be correctly configured on `tf` to permit the navigation stack to move the robot without problems, and to know exactly where each one of their components is.

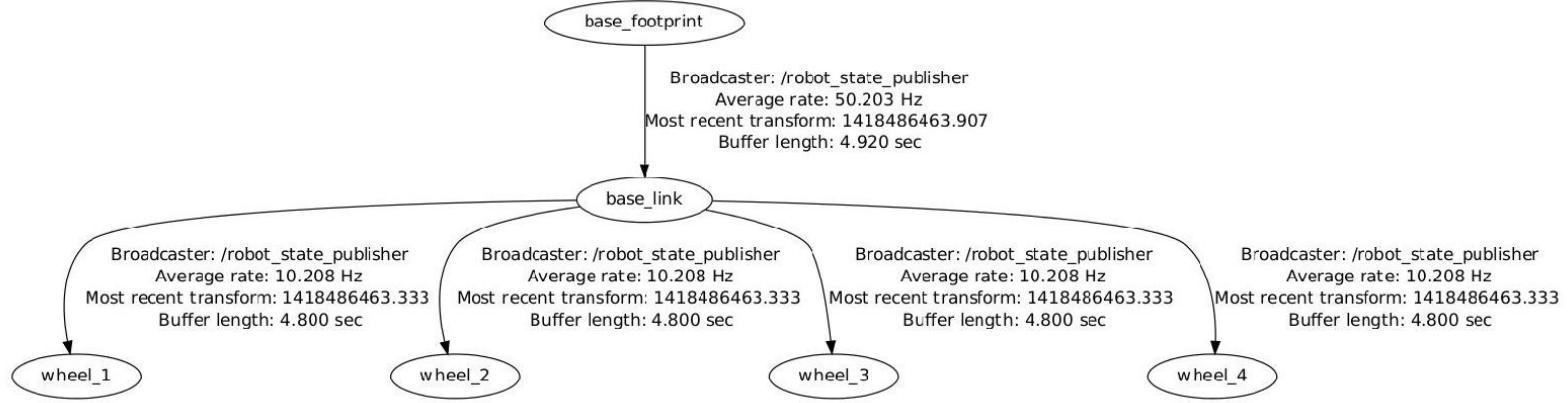
Before starting to write the code to configure each component, keep in mind that you have the geometry of the robot specified in the URDF file. So, for this reason, it is not necessary to configure the robot again. Perhaps you do not know it, but you have been using the `robot_state_publisher` package to publish the transform tree of your robot.

Watching the transformation tree

If you want to see the transformation tree of your robot, use the following command:

```
$ roslaunch chapter5_tutorials gazebo_map_robot.launch model:='`rospack find chapter5_tutorials`/urdf/robot1_base_01.xacro'
$ rosrun tf view_frames
```

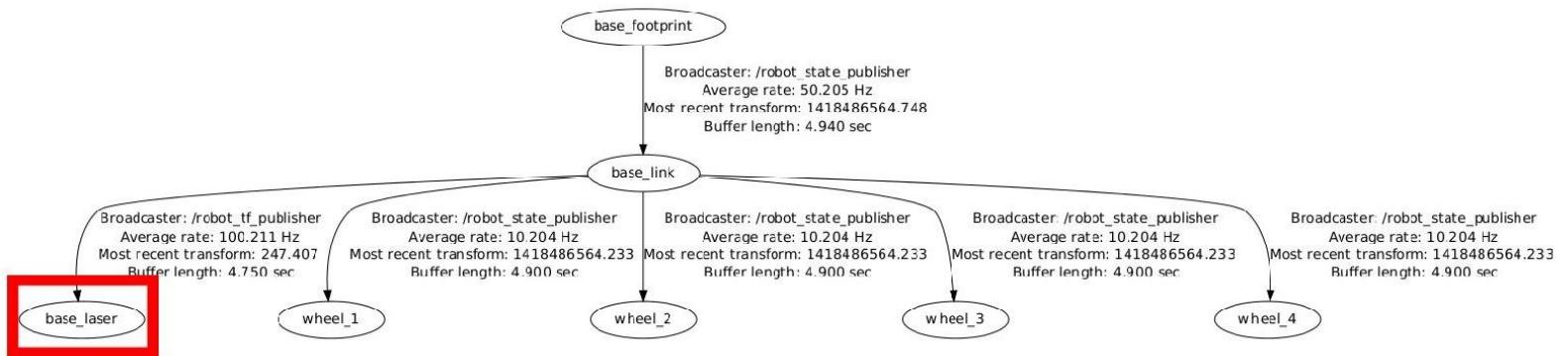
The resultant frame is depicted as follows:



And now, if you run `tf_broadcaster` and run the `rosrun tf view_frames` command again, you will see the frame that you have created using code:

```
$ rosrun chapter5_tutorials tf_broadcaster
$ rosrun tf view_frames
```

The resultant frame is depicted as follows:



Publishing sensor information

Your robot can have a lot of sensors to see the world; you can program a lot of nodes to take this data and do something, but the navigation stack is prepared only to use the planar laser's sensor. So, your sensor must publish the data with one of these types: `sensor_msgs/LaserScan` OR `sensor_msgs/PointCloud2`.

We are going to use the laser located in front of the robot to navigate in Gazebo. Remember that this laser is simulated on Gazebo, and it publishes data on the `hokuyo_link` frame with the topic name `/robot/laser/scan`.

In our case, we do not need to configure anything in our laser to use it on the navigation stack. This is because we have `tf` configured in the `.urdf` file, and the laser is publishing data with the correct type.

If you use a real laser, ROS might have a driver for it. Indeed, in Chapter 7, *Using Sensors and Actuators with ROS*, we will show you how to connect the Hokuyo laser to ROS. Anyway, if you are using a laser that has no driver on ROS and want to write a node to publish the data with the `sensor_msgs/LaserScan` sensor, you have an example template to do it, which is shown in the following section.

But first, remember the structure of the message `sensor_msgs/LaserScan`. Use the following command:

```
$ rosmsg show sensor_msgs/LaserScan
```

The preceding command will generate the following output:

```
std_msgs/Header header
uint32 seq
time stamp
string frame_id
float32 angle_min
float32 angle_max
float32 angle_increment
float32 time_increment
float32 scan_time
float32 range_min
float32 range_max
float32[] ranges
float32[] intensities
```

Creating the laser node

Now we will create a new file in `chapter5_tutorials/src` with the name `laser.cpp` and put the following code in it:

```
#include <ros/ros.h>
#include <sensor_msgs/LaserScan.h>

int main(int argc, char** argv){
    ros::init(argc, argv, "laser_scan_publisher");

    ros::NodeHandle n;
    ros::Publisher scan_pub = n.advertise<sensor_
        msgs::LaserScan>("scan", 50);

    unsigned int num_readings = 100;
    double laser_frequency = 40;
    double ranges[num_readings];
    double intensities[num_readings];

    int count = 0;
    ros::Rate r(1.0);
    while(n.ok()){
        //generate some fake data for our laser scan
        for(unsigned int i = 0; i < num_readings; ++i){
            ranges[i] = count;
            intensities[i] = 100 + count;
        }
        ros::Time scan_time = ros::Time::now();
        //populate the LaserScan message
        sensor_msgs::LaserScan scan;
        scan.header.stamp = scan_time;
        scan.header.frame_id = "base_link";
        scan.angle_min = -1.57;
        scan.angle_max = 1.57;
        scan.angle_increment = 3.14 / num_readings;
        scan.time_increment = (1 / laser_frequency) / (num_readings);
        scan.range_min = 0.0;
        scan.range_max = 100.0;

        scan.ranges.resize(num_readings);
        scan.intensities.resize(num_readings);
        for(unsigned int i = 0; i < num_readings; ++i){
            scan.ranges[i] = ranges[i];
            scan.intensities[i] = intensities[i];
        }

        scan_pub.publish(scan);
        ++count;
        r.sleep();
    }
}
```

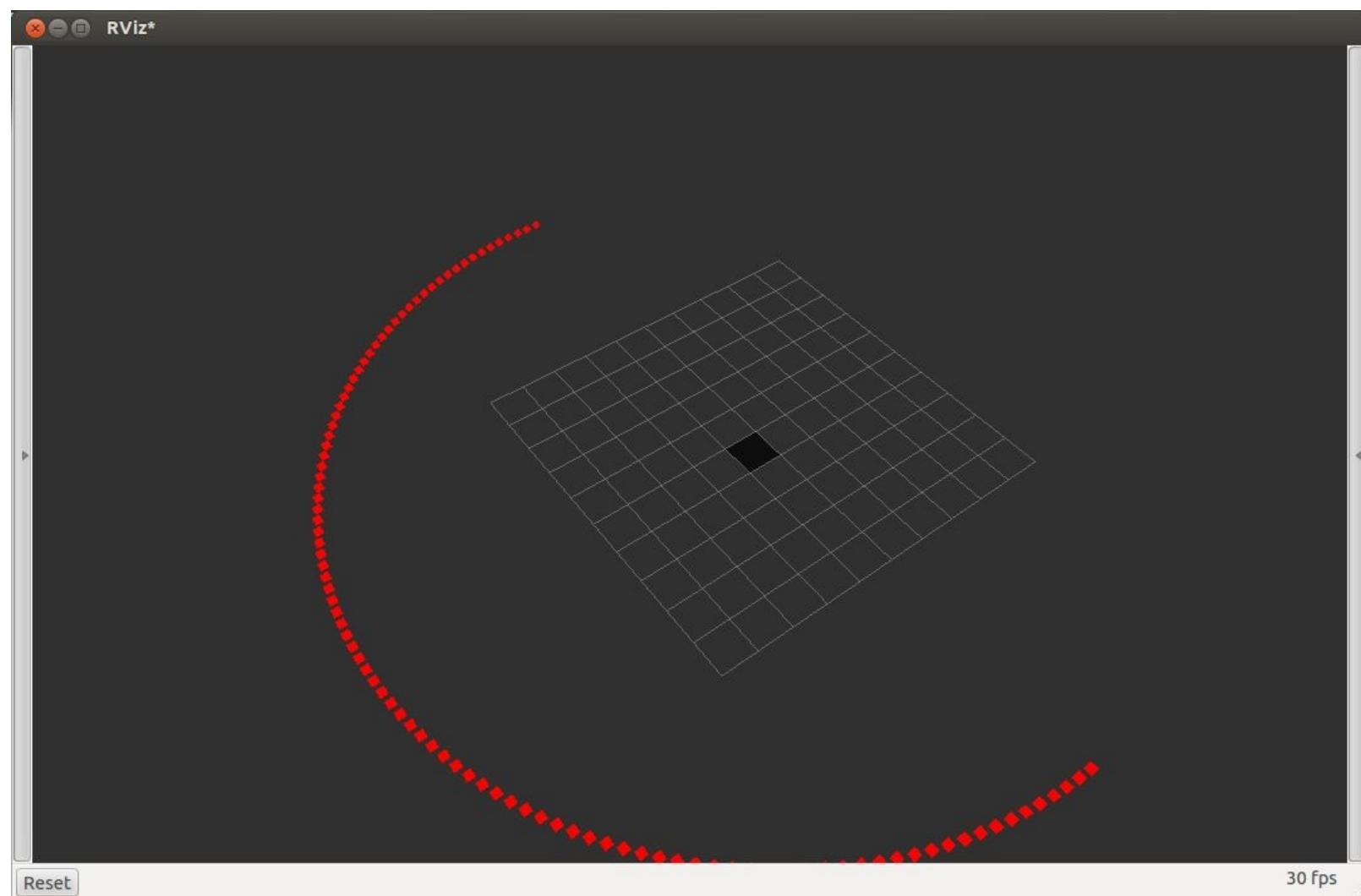
As you can see, we are going to create a new topic with the name `scan` and the message type `sensor_msgs/LaserScan`. You will become familiar with this message type when you reach Chapter 7, *Using Sensors and Actuators with ROS*. The name of the topic must be unique. When you configure the navigation stack, you will select this topic to be used for navigation. The following command line shows how to create the topic with the correct name:

```
ros::Publisher scan_pub =
n.advertise<sensor_msgs::LaserScan>("scan", 50);
```

It is important to publish data with `header`, `stamp`, `frame_id`, and many more elements because, if not, the navigation stack could fail with such data:

```
| scan.header.stamp = scan_time;  
| scan.header.frame_id = "base_link";
```

Other important data on `header` is `frame_id`. It must be one of the frames created in the `.urdf` file and must have a frame published on the `tf` frame transforms. The navigation stack will use this information to know the real position of the sensor and make transforms, such as the one between the data sensor and obstacles.



With this template, you can use any laser, even if it has no driver for ROS. You only have to change the fake data with the right data from your laser.

This template can also be used to create something that looks like a laser but is not. For example, you could simulate a laser using stereoscopy or using a sensor such as a sonar.

Publishing odometry information

The navigation stack also needs to receive data from the robot odometry. The odometry is the distance of something relative to a point. In our case, it is the distance between `base_link` and a fixed point in the frame `odom`.

The type of message used by the navigation stack is `nav_msgs/Odometry`. We can see its structure using the following command:

```
$ rosmsg show nav_msgs/Odometry
```

```
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
string child_frame_id
geometry_msgs/PoseWithCovariance pose
  geometry_msgs/Pose pose
    geometry_msgs/Point position
      float64 x
      float64 y
      float64 z
    geometry_msgs/Quaternion orientation
      float64 x
      float64 y
      float64 z
      float64 w
    float64[36] covariance
geometry_msgs/TwistWithCovariance twist
  geometry_msgs/Twist twist
    geometry_msgs/Vector3 linear
      float64 x
      float64 y
      float64 z
    geometry_msgs/Vector3 angular
      float64 x
      float64 y
      float64 z
    float64[36] covariance
```

As you can see in the message structure, `nav_msgs/Odometry` gives the position of the robot between `frame_id` and `child_frame_id`. It also gives us the pose of the robot using the `geometry_msgs/Pose` message, and the velocity with the `geometry_msgs/Twist` message.

The pose has two structures that show the position in Euler coordinates and the orientation of the robot using a quaternion. The orientation is the angular displacement of the robot.

The velocity has two structures that show the linear velocity and the angular velocity. For our robot, we will use only the linear `x` velocity and the angular `z` velocity. We will use the linear `x` velocity to know whether the robot is moving forward or backward. The angular `z` velocity is used to check whether the robot is rotating towards the left or right.

As the odometry is the displacement between two frames, it is necessary to publish its transform. We did it in the last section, but later on in this section, we will show you an example for publishing the odometry and the transform of our robot.

Now, let us show you how Gazebo works with the odometry.

How Gazebo creates the odometry

As you have seen in other examples with Gazebo, our robot moves in the simulated world just like a robot in the real world. We use a driver for our robot, `diffdrive_plugin`.

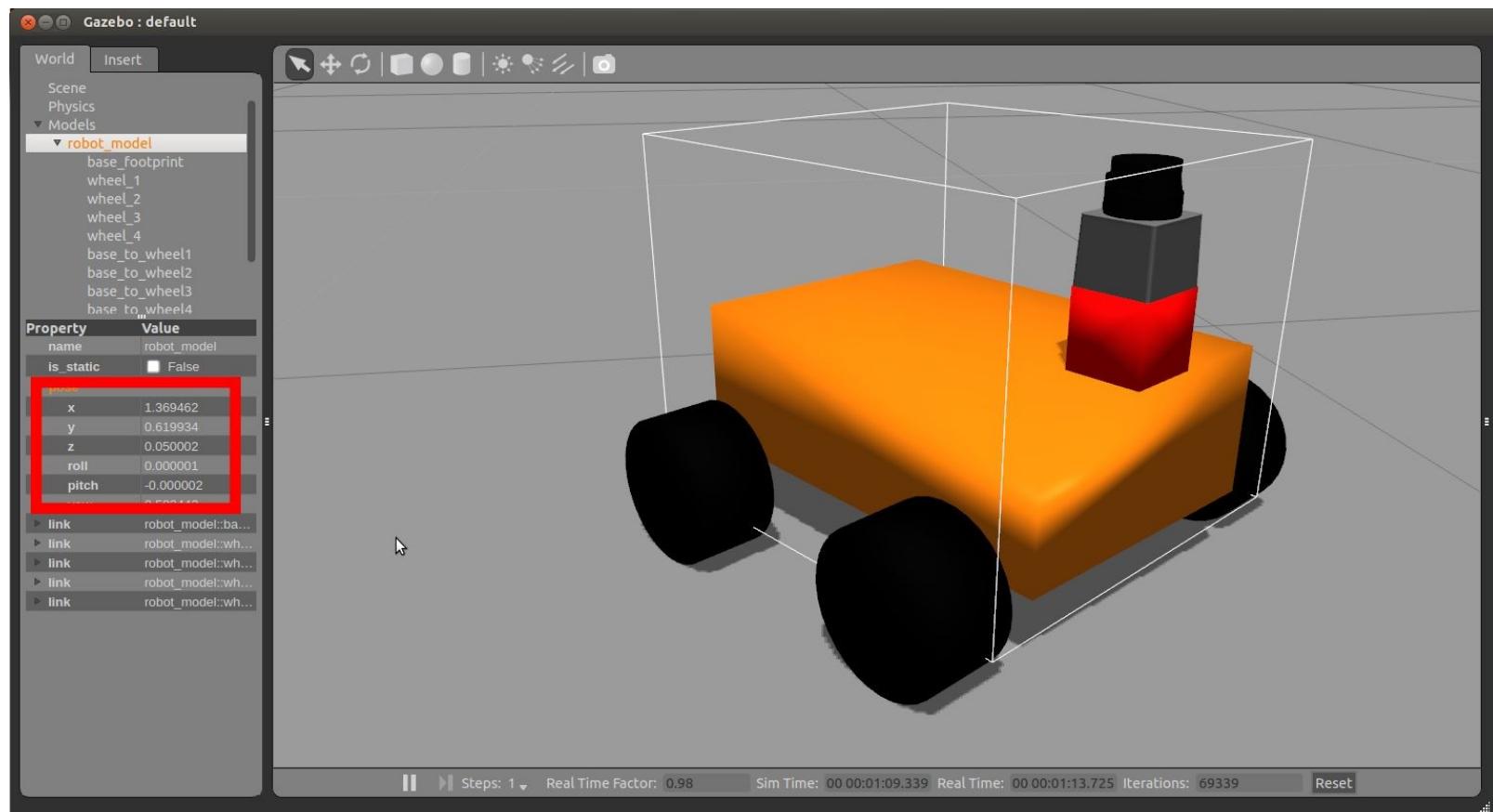
This driver publishes the odometry generated in the simulated world, so we do not need to write anything for Gazebo.

Execute the robot sample in Gazebo to see the odometry working. Type the following commands in the shell:

```
$ roslaunch chapter5_tutorials gazebo_xacro.launch model:='rospack find  
robot1_description'/urdf/robot1_base_04.xacro"  
$ rosrun teleop_twist_keyboard teleop_twist_keyboard.py
```

Then, with the `teleop` node, move the robot for a few seconds to generate new data on the odometry topic.

On the screen of the Gazebo simulator, if you click on `robot_model1`, you will see some properties of the object. One of these properties is the pose of the robot. Click on the pose, and you will see some fields with data. What you are watching is the position of the robot in the virtual world. If you move the robot, the data changes:



Gazebo continuously publishes the odometry data. Check the topic and see what data it is sending. Type the following command in a shell:

```
$ rostopic echo /odom/pose/pose
```

The following is the output that you will receive:

As you can observe, Gazebo is creating the odometry as the robot moves. We are going to see how Gazebo creates it by looking inside the plugin's source code.

The plugin file is located in the `gazebo_plugins` package, and the file is `gazebo_ros_skid_steer_drive.cpp`. You can find the code at https://github.com/ros-simulation/gazebo_ros_pkgs/blob/kinetic-devel/gazebo_plugins/src/gazebo_ros_skid_steer_drive.cpp.

The file has a lot of code, but the important part for us now is the following function, `publishOdometry()`:

```

void GazeboRosSkidSteerDrive::publishOdometry(double step_time)
{
    ros::Time current_time = ros::Time::now();
    std::string odom_frame =
    tf::resolve(tf_prefix_, odometry_frame_);
    std::string base_footprint_frame =
    tf::resolve(tf_prefix_, robot_base_frame_);
    // TODO create some non-perfect odometry!
    // getting data for base_footprint to odom transform
    math::Pose pose = this->parent->GetWorldPose();
    tf::Quaternion qt(pose.rot.x, pose.rot.y, pose.rot.z, pose.rot.w);
    tf::Vector3 vt(pose.pos.x, pose.pos.y, pose.pos.z);
    tf::Transform base_footprint_to_odom(qt, vt);
    if (this->broadcast_tf_)
    {
        transform_broadcaster_->sendTransform(
            tf::StampedTransform(base_footprint_to_odom, current_time,
            odom_frame, base_footprint_frame));
    }
    // publish odom topic
    odom_.pose.pose.position.x = pose.pos.x;
    odom_.pose.pose.position.y = pose.pos.y;
    odom_.pose.pose.orientation.x = pose.rot.x;
    odom_.pose.pose.orientation.y = pose.rot.y;
    odom_.pose.pose.orientation.z = pose.rot.z;
    odom_.pose.pose.orientation.w = pose.rot.w;
    odom_.pose.covariance[0] = 0.00001;
    odom_.pose.covariance[7] = 0.00001;
    odom_.pose.covariance[14] = 1000000000000.0;
    odom_.pose.covariance[21] = 1000000000000.0;
    odom_.pose.covariance[28] = 1000000000000.0;
    odom_.pose.covariance[35] = 0.01;
    // get velocity in /odom frame

```

```

math::Vector3 linear;
linear = this->parent->GetWorldLinearVel();
odom_.twist.twist.angular.z = this->parent->
    GetWorldAngularVel().z;
// convert velocity to child_frame_id (aka base_footprint)
float yaw = pose.rot.GetYaw();
odom_.twist.twist.linear.x = cosf(yaw) * linear.x + sinf(yaw) *
    linear.y; odom_.twist.twist.linear.y = cosf(yaw) * linear.y - sinf(yaw) *
    linear.x;
odom_.header.stamp = current_time;
odom_.header.frame_id = odom_frame;
odom_.child_frame_id = base_footprint_frame;
odometry_publisher_.publish(odom_);
}

```

The `publishOdometry()` function is where the odometry is published. You can see how the fields of the structure are filled and the name of the topic for the odometry is set (in this case, it is `odom`). The pose is generated in the other part of the code that we will see in the following section.

Once you have learned how and where Gazebo creates the odometry, you will be ready to learn how to publish the odometry and the transform for a real robot. The following code will show a robot doing circles continuously. The final outcome does not really matter; the important thing to know here is how to publish the correct data for our robot.

Using Gazebo to create the odometry

To obtain some insight of how Gazebo does that, we are going to take a sneak peek inside the `diffdrive_plugin.cpp` file. You can find it at https://github.com/ros-simulation/gazebo_ros_pkgs/blob/kinetic-devel/gazebo_plugins/src/gazebo_ros_skid_steer_drive.cpp.

The `Load` function performs the function of registering the subscriber of the topic, and when a `cmd_vel` topic is received, the `cmdVelCallback()` function is executed to handle the message:

```
void GazeboRosSkidSteerDrive::Load(physics::ModelPtr _parent,
sdf::ElementPtr _sdf)
{
    ...
    ...
    // ROS: Subscribe to the velocity command topic (usually
    "cmd_vel")
    ros::SubscribeOptions so =
    ros::SubscribeOptions::create<geometry_msgs::Twist>(command_topic_
    , 1,
    boost::bind(&GazeboRosSkidSteerDrive::cmdVelCallback, this, _1),
    ros::VoidPtr(), &queue_);
    ...
    ...
}
```

When a message arrives, the linear and angular velocities are stored in the internal variables to run operations later:

```
void GazeboRosSkidSteerDrive::cmdVelCallback(
const geometry_msgs::Twist::ConstPtr& cmd_msg)
{
    boost::mutex::scoped_lock scoped_lock(lock);
    x_ = cmd_msg->linear.x;
    rot_ = cmd_msg->angular.z;
}
```

The plugin estimates the velocity for each motor, using the formulas from the kinematic model of the robot, in the following manner:

```
void GazeboRosSkidSteerDrive::getWheelVelocities() {
    boost::mutex::scoped_lock scoped_lock(lock);
    double vr = x_;
    double va = rot_;
    wheel_speed_[RIGHT_FRONT] = vr + va * wheel_separation_ / 2.0;
    wheel_speed_[RIGHT_REAR] = vr + va * wheel_separation_ / 2.0;
    wheel_speed_[LEFT_FRONT] = vr - va * wheel_separation_ / 2.0;
    wheel_speed_[LEFT_REAR] = vr - va * wheel_separation_ / 2.0;
}
```

And finally, it estimates the distance traversed by the robot using more formulas from the kinematic motion model of the robot. As you can see in the code, you must know the wheel diameter and the wheel separation of your robot:

```
// Update the controller
void GazeboRosSkidSteerDrive::UpdateChild()
{
    common::Time current_time = this->world->GetSimTime();
    double seconds_since_last_update =
    (current_time - last_update_time_).Double();
    if (seconds_since_last_update > update_period_)
```

```
{  
    publishOdometry(seconds_since_last_update);  
    // Update robot in case new velocities have been requested  
    getWheelVelocities();  
    joints[LEFT_FRONT]->SetVelocity(0, wheel_speed_[LEFT_FRONT] /  
        wheel_diameter_);    joints[RIGHT_FRONT]->SetVelocity(0, wheel_speed_[RIGHT_FRONT] /  
        wheel_diameter_);    joints[LEFT_REAR]->SetVelocity(0, wheel_speed_[LEFT_REAR] /  
        wheel_diameter_);    joints[RIGHT_REAR]->SetVelocity(0, wheel_speed_[RIGHT_REAR] /  
        wheel_diameter_);  
    last_update_time_+= common::Time(update_period_);  
}  
}
```

This is the way `gazebo_ros_skid_steer_drive` controls our simulated robot in Gazebo.

Creating our own odometry

Create a new file in `chapter5_tutorials/src` with the name `odometry.cpp`, and write a code to publish odometry, you can use the following code snippets or check the code repository to have the entire code:

```
#include <string>
#include <ros/ros.h>
#include <sensor_msgs/JointState.h>
#include <tf/transform_broadcaster.h>
#include <nav_msgs/Odometry.h>

int main(int argc, char** argv) {

    ros::init(argc, argv, "state_publisher");
    ros::NodeHandle n;
    ros::Publisher odom_pub = n.advertise<nav_msgs::Odometry>("odom",
10);
    ...

    while (ros::ok()) {
        current_time = ros::Time::now();

        double dt = (current_time - last_time).toSec();
        double delta_x = (vx * cos(th) - vy * sin(th)) * dt;
        double delta_y = (vx * sin(th) + vy * cos(th)) * dt;
        double delta_th = vth * dt;

        x += delta_x;
        y += delta_y;
        th += delta_th;

        geometry_msgs::Quaternion odom_quat;
        odom_quat = tf::createQuaternionMsgFromRollPitchYaw(0,0,th);

        // update transform
        odom_trans.header.stamp = current_time;
        odom_trans.transform.translation.x = x;
        odom_trans.transform.translation.y = y;
        odom_trans.transform.translation.z = 0.0;
        odom_trans.transform.rotation =
            tf::createQuaternionMsgFromYaw(th);

        // filling the odometry
        nav_msgs::Odometry odom;
        odom.header.stamp = current_time;
        odom.header.frame_id = "odom";
        odom.child_frame_id = "base_footprint";
        // position
        odom.pose.pose.position.x = x;
        odom.pose.pose.position.y = y;
        odom.pose.pose.position.z = 0.0;
        odom.pose.pose.orientation = odom_quat;

        // velocity
        odom.twist.twist.linear.x = vx;
        odom.twist.twist.linear.y = vy;

        ...
        odom.twist.twist.angular.z = vth;
        last_time = current_time;

        // publishing the odometry and the new tf
        broadcaster.sendTransform(odom_trans);
        odom_pub.publish(odom);

        loop_rate.sleep();
    }
    return 0;
}
```

| }

First, create the transformation variable and fill it with `frame_id` and `child_frame_id` values in order to know when the frames have to move. In our case, the `base_footprint` will move relatively towards the frame `odom`:

```
geometry_msgs::TransformStamped odom_trans;
odom_trans.header.frame_id = "odom";
odom_trans.child_frame_id = "base_footprint";
```

In this part, we generate the pose of the robot. With the linear velocity and the angular velocity, we can calculate the theoretical position of the robot after a while:

```
double dt = (current_time - last_time).toSec();
double delta_x = (vx * cos(th) - vy * sin(th)) * dt;
double delta_y = (vx * sin(th) + vy * cos(th)) * dt;
double delta_th = vth * dt;

x += delta_x;
y += delta_y;
th += delta_th;

geometry_msgs::Quaternion odom_quat;
odom_quat = tf::createQuaternionMsgFromRollPitchYaw(0, 0, th);
```

In the transformation, we will only fill in the `x` and `rotation` fields, as our robot can only move forward and backward and can turn:

```
odom_trans.header.stamp = current_time;
odom_trans.transform.translation.x = x;
odom_trans.transform.translation.y = 0.0;
odom_trans.transform.translation.z = 0.0;
odom_trans.transform.rotation = tf::createQuaternionMsgFromYaw(th);
```

With the odometry, we will do the same. Fill the `frame_id` and `child_frame_id` fields with `odom` and `base_footprint`.

As the odometry has two structures, we will fill in the `x`, `y`, and `orientation` of the pose. In the `twist` structure, we will fill in the linear velocity `x` and the angular velocity `z`:

```
// position
odom.pose.pose.position.x = x;
odom.pose.pose.position.y = y;
odom.pose.pose.orientation = odom_quat;

// velocity
odom.twist.twist.linear.x = vx;
odom.twist.twist.angular.z = vth;
```

Once all the necessary fields are filled in, publish the data:

```
// publishing the odometry and the new tf
broadcaster.sendTransform(odom_trans);
odom_pub.publish(odom);
```

Remember to create the following line in the `CMakeLists.txt` file before compiling it:

```
add_executable(odometry src/odometry.cpp)
target_link_libraries(odometry ${catkin_LIBRARIES})
```

Compile the package and launch the robot without using Gazebo, using only `rviz` to visualize the model

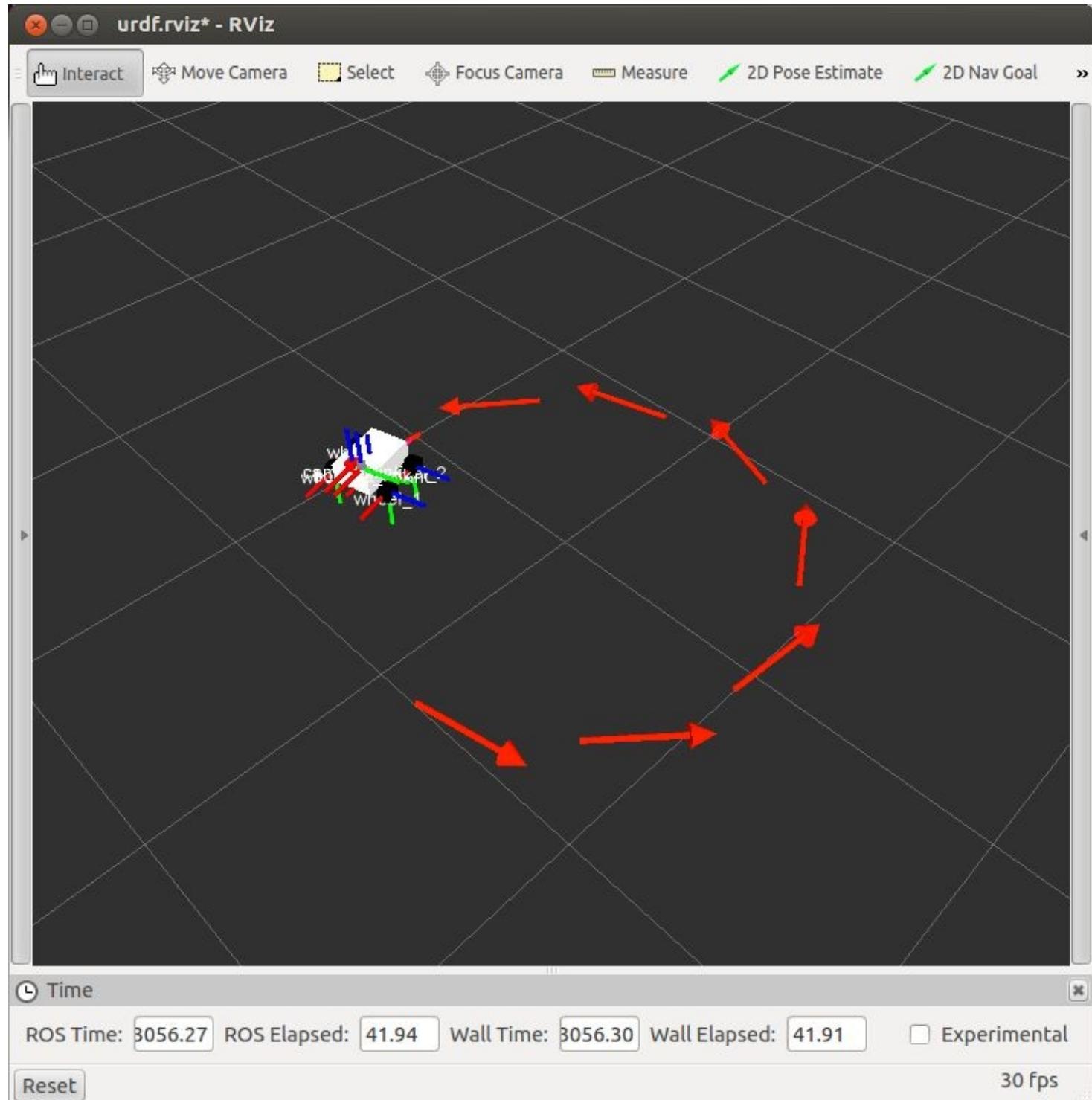
and the movement of the robot. Use the following command to do this:

```
$ roslaunch chapter5_tutorials display_xacro.launch model:='rospack find chapter5_tutorials'/urdf/robot1_base_04.xacro'
```

Now, run the `odometry` node with the following command:

```
$ rosrun chapter5_tutorials odometry
```

The following output is what you will get:



On the `rviz` screen, you can see the robot moving over some red arrows (grid). The robot moves over the grid because you published a new `tf` frame transform for the robot. The red arrows are the graphical

representation for the `odometry` message. You will see the robot moving in circles continuously as we programmed in the code.

Creating a base controller

A base controller is an important element in the navigation stack because it is the only way to effectively control your robot. It communicates directly with the electronics of your robot.

ROS does not provide a standard base controller, so you must write a base controller for your mobile platform.

Your robot has to be controlled with the message type `geometry_msgs/Twist`. This message was used on the `Odometry` message that we saw before.

So, your base controller must subscribe to a topic with the name `cmd_vel`, and must generate the correct commands to move the platform with the correct linear and angular velocities.

We are now going to recall the structure of this message. Type the following command in a shell to see the structure:

```
$ rosmsg show geometry_msgs/Twist
```

The output of this command is as follows:

```
geometry_msgs/Vector3 linear
  float64 x
  float64 y
  float64 z
geometry_msgs/Vector3 angular
  float64 x
  float64 y
  float64 z
```

The vector with the name `linear` indicates the linear velocity for the axes `x`, `y`, and `z`. The vector with the name `angular` is for the angular velocity on the axes.

For our robot, we will only use the linear velocity `x` and the angular velocity `z`. This is because our robot is on a differential-wheeled platform; it has two motors to move the robot forward and backward and to turn.

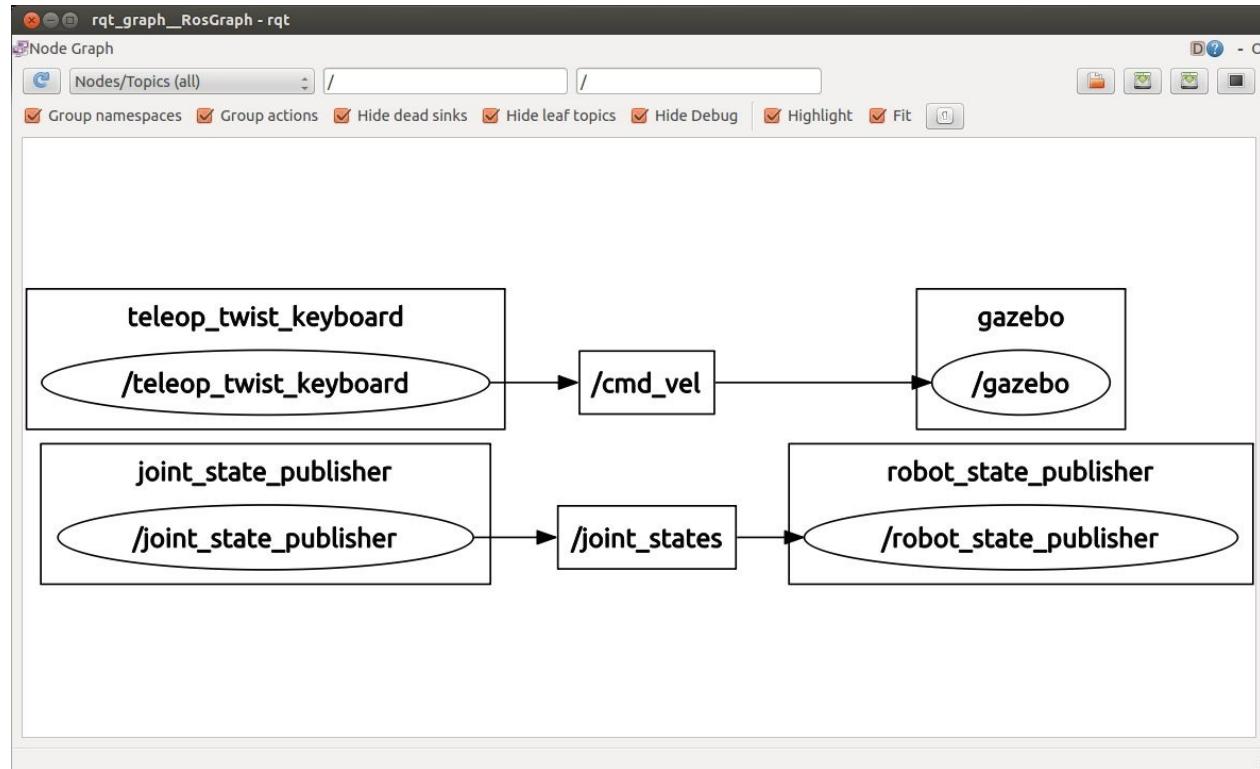
We are working with a simulated robot on Gazebo, and the base controller is implemented on the driver used to move/simulate the platform. This means that we will not have to create the base controller for this robot.

Anyway, in this chapter, you will see an example to implement the base controller on your physical robot. Before that, let's execute our robot on Gazebo to see how the base controller works. Run the following commands on different shells:

```
$ roslaunch chapter5_tutorials gazebo_xacro.launch model:='`rospack find
robot1_description`/urdf/robot1_base_04.xacro'
$ rosrun teleop_twist_keyboard teleop_twist_keyboard.py
```

When all the nodes are launched and working, open rxgraph to see the relation between all the nodes:

```
$ rqt_graph
```



You can see that Gazebo subscribes automatically to the `cmd_vel` topic that is generated by the `teleop` node.

Inside the Gazebo simulator, the plugin of our robot is running and is getting the data from the `cmd_vel` topic. Also, this plugin moves the robot in the virtual world and generates the odometry.

Creating our base controller

Now, we are going to do something similar, that is, prepare a code to be used with a real robot with two wheels and encoders.

Create a new file in `chapter5_tutorials/src` with the name `base_controller.cpp` and put in the following code:

```
#include <ros/ros.h>
#include <sensor_msgs/JointState.h>
#include <tf/transform_broadcaster.h>
#include <nav_msgs/Odometry.h>
#include <iostream>

using namespace std;

double width_robot = 0.1;
double vl = 0.0;
double vr = 0.0;
ros::Time last_time;
double right_enc = 0.0;
double left_enc = 0.0;
double right_enc_old = 0.0;
double left_enc_old = 0.0;
double distance_left = 0.0;
double distance_right = 0.0;
double ticks_per_meter = 100;
double x = 0.0;
double y = 0.0;
double th = 0.0;
geometry_msgs::Quaternion odom_quat;
```

In this part of the code, we have declared global variables and including the libraries to calculate the odometry and making the spatial transformation for positioning our robot. Now, you should create a call back function to receive velocity commands. Applying some kinematics equations, you could relate velocity commands with the speed of each wheels of your robot.

```
void cmd_velCallback(const geometry_msgs::Twist &twist_aux)
{
    geometry_msgs::Twist twist = twist_aux;
    double vel_x = twist_aux.linear.x;
    double vel_th = twist_aux.angular.z;
    double right_vel = 0.0;
    double left_vel = 0.0;

    if(vel_x == 0){
        // turning
        right_vel = vel_th * width_robot / 2.0;
        left_vel = (-1) * right_vel;
    }else if(vel_th == 0){
        // forward / backward
        left_vel = right_vel = vel_x;
    }else{
        // moving doing arcs
        left_vel = vel_x - vel_th * width_robot / 2.0;
        right_vel = vel_x + vel_th * width_robot / 2.0;
    }
    vl = left_vel;
    vr = right_vel;
}
```

In the `main` function, you are going to code a loop that will update the real velocity of your robot using data from encoders and calculate the odometry.

```

int main(int argc, char** argv){
ros::init(argc, argv, "base_controller");
ros::NodeHandle n;
ros::Subscriber cmd_vel_sub = n.subscribe("cmd_vel", 10,
cmd_velCallback);
ros::Rate loop_rate(10);

while(ros::ok())
{
    double dxy = 0.0;
    double dth = 0.0;
    ros::Time current_time = ros::Time::now();
    double dt;
    double velxy = dxy / dt;
    double velth = dth / dt;

    ros::spinOnce();
    dt = (current_time - last_time).toSec();
    last_time = current_time;

    // calculate odometry
    if(right_enc == 0.0){
        distance_left = 0.0;
        distance_right = 0.0;
    }else{
        distance_left = (left_enc - left_enc_old) / ticks_per_meter;
        distance_right = (right_enc - right_enc_old) / ticks_per_meter;
    }
}

```

Once, the distance traversed by each wheel is known, you will be able to update the robot position calculating the increment of distance `dxy` and the angle increment `dth`.

```

left_enc_old = left_enc;
right_enc_old = right_enc;

dxy = (distance_left + distance_right) / 2.0;
dth = (distance_right - distance_left) / width_robot;

if(dxy != 0){
    x += dxy * cosf(dth);
    y += dxy * sinf(dth);
}

if(dth != 0){
    th += dth;
}

```

Position of the robot is calculated in `x` and `y`, for a base controller of a 2D robot platform you will suppose `z` is 0 and constant. For the robot orientation, you can assume that pitch and roll are also equal to zero and only the yaw angle should be update.

```

odom_quat = tf::createQuaternionMsgFromRollPitchYaw(0,0,th);
loop_rate.sleep();
}

```

Do not forget to insert the following in your `CMakeLists.txt` file to create the executable of this file:

```

add_executable(base_controller src/base_controller.cpp)
target_link_libraries(base_controller ${catkin_LIBRARIES})

```

This code is only a common example and must be extended with more code to make it work with a specific robot. It depends on the controller used, the encoders, and so on. We assume that you have the right background to add the necessary code in order to make the example work fine. In Chapter 7, *Using Sensors and Actuators with ROS* a fully functional example will be provided with a real robot platform

with wheels and encoders.

Creating a map with ROS

Getting a map can sometimes be a complicated task if you do not have the correct tools. ROS has a tool that will help you build a map using the odometry and a laser sensor. This tool is the `map_server` (http://wiki.ros.org/map_server). In this example, you will learn how to use the robot that we created in Gazebo, as we did in the previous chapters, to create a map, save it, and load it again.

We are going to use a `.launch` file to make it easy. Create a new file in `chapter5_tutorials/launch` with the name `gazebo_mapping_robot.launch` and put in the following code:

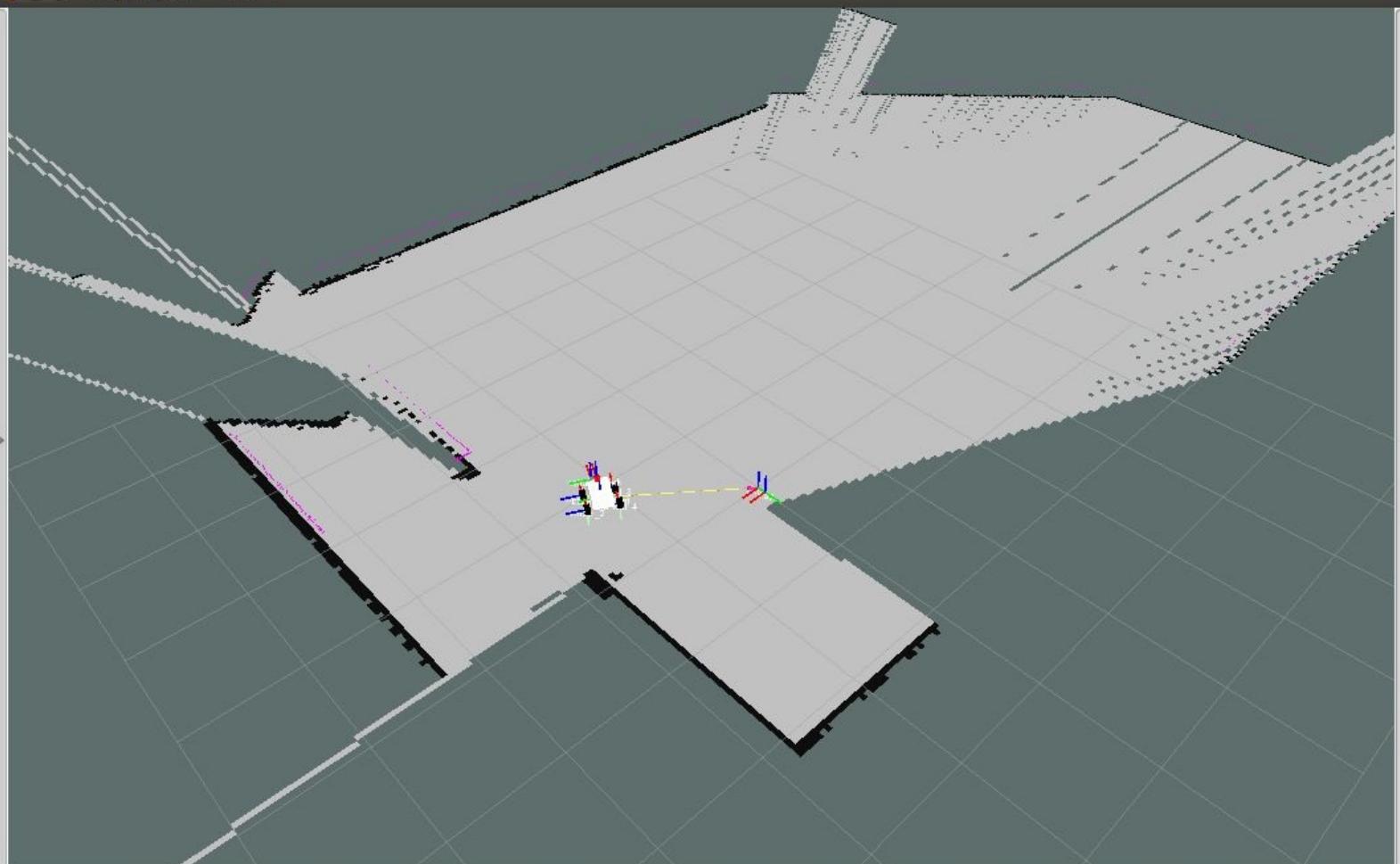
```
<?xml version="1.0"?>
<launch>
  <param name="/use_sim_time" value="true" />
  <include file="$(find
    gazebo_ros)/launch/willowgarage_world.launch"/>
  <arg name="model" />
  <param name="robot_description" command="$(find xacro)/xacro.py
    $(arg model)" /> <node name="joint_state_publisher" pkg="joint_state_publisher"
    type="joint_state_publisher" /></node>
  <!-- start robot state publisher -->
  <node pkg="robot_state_publisher" type="robot_state_publisher"
    name="robot_state_publisher" output="screen" />
  <param name="publish_frequency" type="double" value="50.0" />
  </node>
  <node name="spawn_robot" pkg="gazebo_ros" type="spawn_model"
    args="-urdf -param robot_description -z 0.1 -model robot_model"
    respawn="false" output="screen" /> <node name="rviz" pkg="rviz" type="rviz" args="-d $(find
    chapter5_tutorials)/launch/mapping.rviz"/>
  <node name="slam_gmapping" pkg="gmapping" type="slam_gmapping">
    <remap from="scan" to="/robot/laser/scan"/>
    <param name="base_link" value="base_footprint"/>
  </node>
</launch>
```

With this `.launch` file, you can launch the Gazebo simulator with the 3D model, the `rviz` program with the correct configuration file, and `slam_mapping` to build a map in real time. Launch the file in a shell, and in the other shell, run the `teleop` node to move the robot:

```
$ roslaunch chapter5_tutorials gazebo_mapping_robot.launch model:='`rospack find
robot1_description`/urdf/robot1_base_04.xacro'
$ rosrun teleop_twist_keyboard teleop_twist_keyboard.py
```

When you start to move the robot with the keyboard, you will see the free and unknown space on the `rviz` screen, as well as the map with the occupied space; this is known as an **Occupancy Grid Map (OGM)**. The `slam_mapping` node updates the map state when the robot moves, or more specifically, when (after some motion) it has a good estimate of the robot's location and how the map is. It takes the laser scans and the odometry to build the OGM for you.

mapping.rviz* - RViz



Reset

30 fps

Saving the map using map_server

Once you have a complete map or something acceptable, you can save it to use it later in the navigation stack. To save it, use the following command:

```
$ rosrun map_server map_saver -f map
```

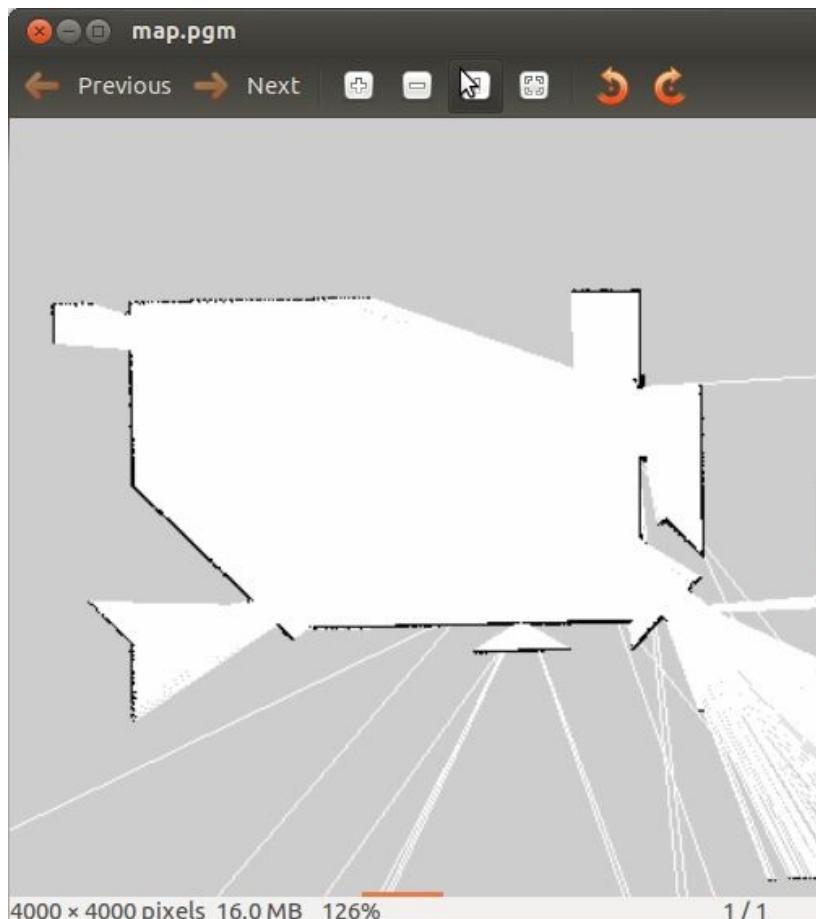
This command would give the following result:

```
[ INFO] [1418594807.613374681]: Waiting for the map
[ INFO] [1418594807.958979924, 126.530000000]: Received a 4000 X 4000 map @ 0.050 m/pix
[ INFO] [1418594807.959452501, 126.530000000]: Writing map occupancy data to map.pgm
[ INFO] [1418594808.997886519, 127.085000000]: Writing map occupancy data to map.yaml
[ INFO] [1418594808.998301431, 127.085000000]: Done
```

This command will create two files, `map.pgm` and `map.yaml`. The first one is the map in the `.pgm` format (the portable gray map format). The other is the configuration file for the map. If you open it, you will see the following output:

```
image: map.pgm
resolution: 0.050000
origin: [-100.000000, -100.000000, 0.000000]
negate: 0
occupied_thresh: 0.65
free_thresh: 0.196
```

Now, open the `.pgm` image file with your favorite viewer, and you will see the map being built before you:



Loading the map using map_server

When you want to use the map built with your robot, it is necessary to load it with the `map_server` package. The following command will load the map:

```
$ rosrun map_server map_server map.yaml
```

But to make it easy, create another `.launch` file in `chapter5_tutorials/launch` with the name `gazebo_map_robot.launch`, and put in the following code:

```
<?xml version="1.0"?>
<launch>
  <param name="/use_sim_time" value="true" />
  <!-- start up wg world -->
  <include file="$(find
    gazebo_ros)/launch/willowgarage_world.launch"/>
  <arg name="model" />
  <param name="robot_description" command="$(find xacro)/xacro.py
    $(arg model)" /> <node name="joint_state_publisher" pkg="joint_state_publisher"
    type="joint_state_publisher" ></node>
  <!-- start robot state publisher -->
  <node pkg="robot_state_publisher" type="robot_state_publisher"
    name="robot_state_publisher" output="screen" >
    <param name="publish_frequency" type="double" value="50.0" />
  </node>
  <node name="spawn_robot" pkg="gazebo_ros" type="spawn_model"
    args="-urdf -param robot_description -z 0.1 -model robot_model"
    respawn="false" output="screen" /> <node name="map_server" pkg="map_server" type="map_server"
    args=" $(find chapter5_tutorials)/maps/map.yaml" /> <node name="rviz" pkg="rviz" type="rviz" args="-d $(find
    chapter5_tutorials)/launch/mapping.rviz" />
</launch>
```

Now, launch the file using the following command and remember to put in the model of the robot that will be used:

```
$ roslaunch chapter5_tutorials gazebo_map_robot.launch model:='`rospack find
chapter5_tutorials`/urdf/robot1_base_04.xacro'
```

Then, you will see `rviz` with the robot and the map. The navigation stack, in order to know the localization of the robot, will use the map published by the map server and the laser readings. This will help it perform a scan matching algorithm that helps in estimating the robot's location using a particle filter implemented in the AMCL node.

We will see more about maps, as well as more useful tools, in the next chapter.

Summary

In this chapter, you worked on the steps required to configure your robot in order to use it with the navigation stack. Now you know that the robot must have a planar laser, must be a differential-wheeled robot, and it should satisfy some requirements for the base control and the geometry.

Keep in mind that we are working with Gazebo to demonstrate the examples and to explain how the navigation stack works with different configurations. It is more complex to explain all of this directly on a real, robotic platform because we do not know whether you have one or have access to one. In any case, depending on the platform, the instructions may vary and the hardware may fail, so it is safer and useful to run these algorithms in simulations; later, we can test them on a real robot, as long as it satisfies the requirements described thus far.

In the next chapter, you will learn how to configure the navigation stack, create the `.launch` files, and navigate autonomously in Gazebo with the robot that you created in the previous chapters.

In brief, what you will learn after this chapter will be extremely useful because it shows you how to configure everything correctly so you know how to use the navigation stack with other robots, either simulated or real.

The Navigation Stack - Beyond Setups

We have created packages, nodes, 3D models of robots, and more. In Chapter 4, *The Navigation Stack - Robot Setups*, you configured your robot in order to be used with the navigation stack, and in this chapter, we will finish the configuration for the navigation stack so that you learn how to use it with your robot.

All the work done in the previous chapters has been a preamble for this precise moment. This is when the fun begins and when the robots come alive.

In this chapter, we are going to learn how to do the following:

- Apply the knowledge of Chapter 4, *The Navigation Stack - Robot Setups* and the programs developed therein
- Understand the navigation stack and how it works
- Configure all the necessary files
- Create launch files to start the navigation stack

Let's begin!

Creating a package

The correct way to create a package is by adding the dependencies with the other packages created for your robot. For example, you could use the next command to create the package:

```
$ roscreate-pkg my_robot_name_2dnav move_base my_tf_configuration_dep my_odom_configuration_dep  
my_sensor_configuration_dep
```

But in our case, as we have everything in the same package, it is only necessary to execute the following:

```
$ catkin_create_pkg chapter6_tutorials roscpp tf
```

Remember that in the repository, you may find all the necessary files for the chapter.

Creating a robot configuration

To launch the entire robot, we are going to create a launch file with all the necessary files to activate all the systems. Anyway, here you have a launch file for a real robot that you can use as a template. The following script is present in `configuration_template.launch`:

```
&lt;launch>
  &lt;node pkg="sensor_node_pkg" type="sensor_node_type"
    name="sensor_node_name" output="screen">
    &lt;param name="sensor_param" value="param_value" />
  &lt;/node>

  &lt;node pkg="odom_node_pkg" type="odom_node_type" name="odom_node"
    output="screen">
    &lt;param name="odom_param" value="param_value" />
  &lt;/node>

  &lt;node pkg="transform_configuration_pkg"
    type="transform_configuration_type"
    name="transform_configuration_name" output="screen">      &lt;param name="transform_configuration_param"
      value="param_value" />
  &lt;/node>
&lt;/launch>
```

This launch file will launch three nodes that will start up the robot.

The first one is the node responsible for activating the sensors, for example, the **Laser Imaging, Detection, and Ranging (LIDAR)** system. The `sensor_param` parameter can be used to configure the sensor's port, for example, if the sensor uses a USB connection. If your sensor needs more parameters, you need to duplicate the line and add the necessary parameters. Some robots have more than one sensor to help in the navigation. In this case, you can add more nodes or create a launch file for the sensors, and include it in this launch file. This could be a good option for easily managing all the nodes in the same file.

The second node is to start the odometry, the base control, and all the necessary files to move the base and calculate the robot's position. Remember that in Chapter 4, *The Navigation Stack - Robot Setups*, we looked at these nodes in some detail. As in the other section, you can use the parameters to configure something in the odometry, or replicate the line to add more nodes.

The third part is meant to launch the node responsible for publishing and calculating the geometry of the robot, and the transform between the arms, sensors, and so on.

The previous file is for your real robot, but for our example, the next launch file is all we need.

Create a new file in `chapter6_tutorials/launch` with the name `chapter6_configuration_gazebo.launch`, and add the following code:

```
&lt;?xml version="1.0"?>
&lt;launch>
  &lt;param name="/use_sim_time" value="true" />
  &lt;remap from="robot/laser/scan" to="/scan" />
  &lt;!-- start up wg world -->
  &lt;include file="$(find
    gazebo_ros)/launch/willowgarage_world.launch"/>  &lt;arg name="model" default="$(find
```

```

robot1_description)/urdf/robot1_base_04.xacro"/> &lt;param name="robot_description" command="$(find
xacro)/xacro.py
$(arg model)" /> &lt;node name="joint_state_publisher" pkg="joint_state_publisher"
type="joint_state_publisher" />&lt;/node>
&lt;!-- start robot state publisher -->
&lt;node pkg="robot_state_publisher" type="robot_state_publisher"
name="robot_state_publisher" output="screen" /> &lt;node name="spawn_robot" pkg="gazebo_ros" type="spawn_model"
args="-urdf -param robot_description -z 0.1 -model robot_model"
respawn="false" output="screen" /> &lt;node name="rviz" pkg="rviz" type="rviz" args="-d $(find
chapter6_tutorials)/launch/navigation.rviz" />
&lt;/launch>

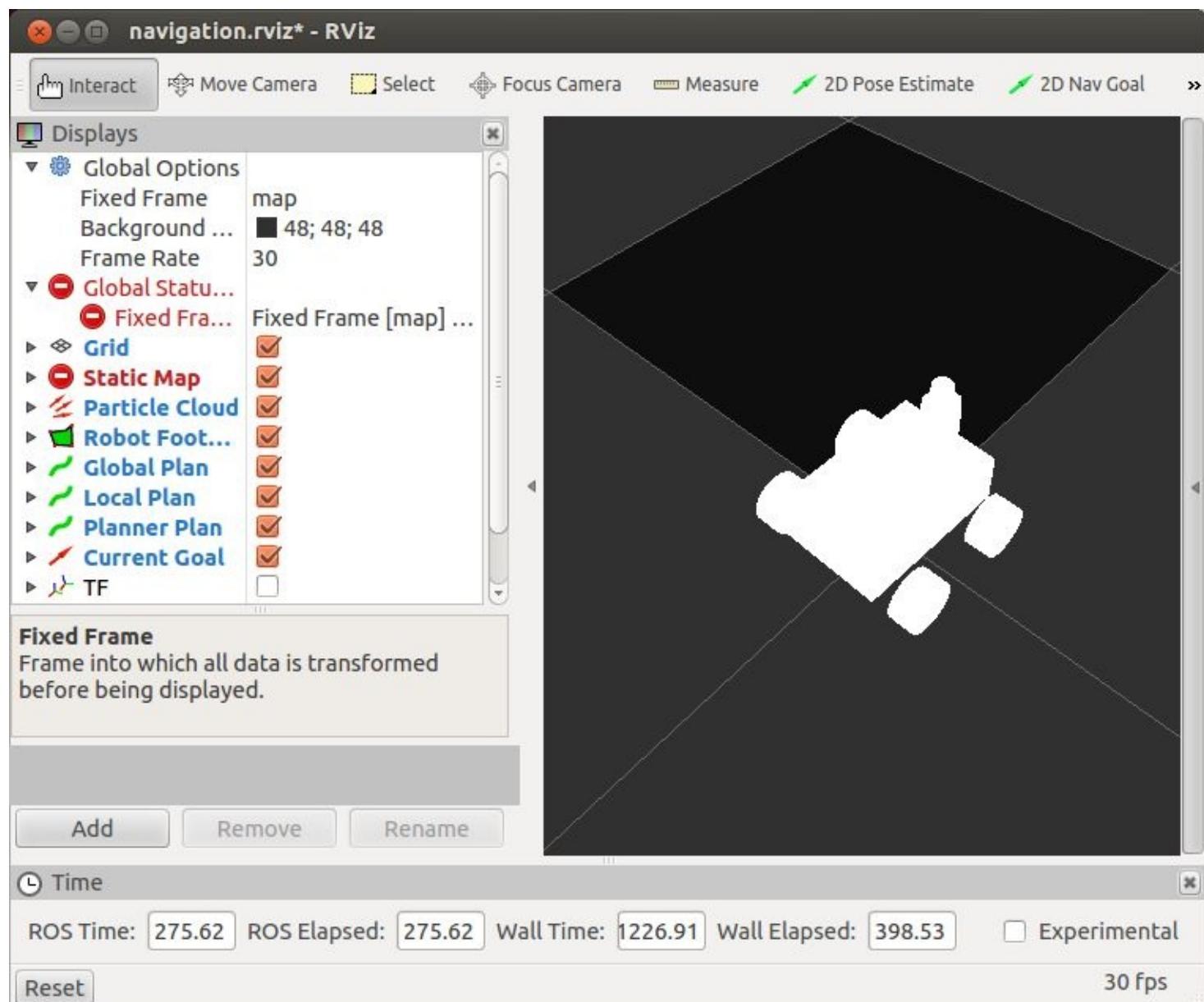
```

This launch file is the same that we used in the previous chapters, so it does not need any additional explanation.

Now to launch this file, use the following command:

```
$ ros launch chapter6_tutorials chapter6_configuration_gazebo.launch
```

You will see the following window:



Notice that in the previous screenshot, there are some fields in red, blue, and yellow, without you having configured anything before. This is because in the launch file, a configuration file for the `rviz` layout is

loaded along with `rviz`, and this file was configured in the previous chapter of this section.

In the upcoming sections, you will learn how to configure `rviz` to use it with the navigation stack and view all the topics.

Configuring the costmaps - global_costmap and local_costmap

Okay, now we are going to start configuring the navigation stack and all the necessary files to start it. To start with the configuration, first we will learn what **costmaps** are and what they are used for. Our robot will move through the map using two types of navigation: `global` and `local`:

- The `global` navigation is used to create paths for a goal in the map or at a far-off distance
- The `local` navigation is used to create paths in the nearby distances and avoid obstacles, for example, a square window of 4 x 4 meters around the robot

These modules use costmaps to keep all the information of our map. The `global` costmap is used for `global` navigation and the `local` costmap for `local` navigation.

The costmaps have parameters to configure the behaviors, and they have common parameters as well, which are configured in a shared file.

The configuration basically consists of three files where we can set up different parameters. The files are as follows:

- `costmap_common_params.yaml`
- `global_costmap_params.yaml`
- `local_costmap_params.yaml`

Just by reading the names of these configuration files, you can instantly guess what they are used for. Now that you have a basic idea about the usage of costmaps, we are going to create the configuration files and explain the parameters that are configured in them.

Configuring the common parameters

Let's start with the common parameters. Create a new file in `chapter6_tutorials/launch` with the name `costmap_common_params.yaml`, and add the following code.

The following script is present in `costmap_common_params.yaml`:

```
obstacle_range: 2.5
raytrace_range: 3.0
footprint: [[-0.2, -0.2], [-0.2, 0.2], [0.2, 0.2], [0.2, -0.2]]
inflation_radius: 0.5
cost_scaling_factor: 10.0
observation_sources: scan
scan: {sensor_frame: base_link, observation_persistence: 0.0,
       max_obstacle_height: 0.4, min_obstacle_height: 0.0, data_type:
       LaserScan, topic: /scan, marking: true, clearing: true}
```

This file is used to configure common parameters. The parameters are used in `local_costmap` and `global_costmap`. Let's break the code and understand it.

The `obstacle_range` and `raytrace_range` attributes are used to indicate the maximum distance that the sensor will read and introduce new information in the costmaps. The first one is used for the obstacles. If the robot detects an obstacle closer than 2.5 meters in our case, it will put the obstacle in the costmap. The other one is used to clean/clear the costmap and update the free space in it when the robot moves. Note that we can only detect the echo of the laser or sonar with the obstacle; we cannot perceive the whole obstacle or object itself, but this simple approach will be enough to deal with these kinds of measurements, and we will be able to build a map and localize within it.

The `footprint` attribute is used to indicate the geometry of the robot to the navigation stack. It is used to keep the right distance between the obstacles and the robot, or to find out if the robot can go through a door. The `inflation_radius` attribute is the value given to keep a minimal distance between the geometry of the robot and the obstacles.

The `cost_scaling_factor` attribute modifies the behavior of the robot around the obstacles. You can make a behavior aggressive or conservative by changing the parameter.

With the `observation_sources` attribute, you can set the sensors used by the navigation stack to get the data from the real world and calculate the path.

In our case, we are using a simulated LIDAR in Gazebo, but we can use a point cloud to do the same.

The following line will configure the sensor's frame and the uses of the data:

```
scan: {sensor_frame: base_link, data_type: LaserScan, topic:
       /scan, marking: true, clearing: true}
```

The laser configured in the previous line is used to add and clear obstacles in the costmap. For example, you could add a sensor with a wide range to find obstacles and another sensor to navigate and clear the obstacles. The topic's name is configured in this line. It is important to configure it well, because the

navigation stack could wait for another topic and all this while, the robot is moving and could crash into a wall or an obstacle.

Configuring the global costmap

The next file to be configured is the `global costmap` configuration file. Create a new file in `chapter6_tutorials/launch` with the name `global_costmap_params.yaml`, and add the following code:

```
global_costmap:  
  global_frame: /map  
  robot_base_frame: /base_footprint  
  update_frequency: 1.0  
  static_map: true
```

The `global_frame` and the `robot_base_frame` attributes define the transformation between the map and the robot. This transformation is for the `global costmap`.

You can configure the frequency of updates for the costmap. In this case, it is 1 Hz. The `static_map` attribute is used for the `global costmap` to see whether a map or the map server is used to initialize the costmap. If you aren't using a static map, set this parameter to `false`.

Configuring the local costmap

The next file is for configuring the `local` costmap. Create a new file in `chapter6_tutorials/launch` with the name `local_costmap_params.yaml`, and add the following code:

```
local_costmap:  
  global_frame: /map  
  robot_base_frame: /base_footprint  
  update_frequency: 5.0  
  publish_frequency: 2.0  
  static_map: false  
  rolling_window: true  
  width: 5.0  
  height: 5.0  
  resolution: 0.02  
  transform_tolerance: 0.5  
  planner_frequency: 1.0  
  planner_patiente: 5.0
```

The `global_frame`, `robot_base_frame`, `update_frequency` and `static_map` parameters are the same as described in the previous section, configuring the `global` costmap. The `publish_frequency` parameter determines the frequency for publishing information. The `rolling_window` parameter is used to keep the costmap centered on the robot when it is moving around the world.

The `transform_tolerance` parameter configures the maximum latency for the transforms, in our case 0.5 seconds. With the `planner_frequency` parameter, we can configure the rate in Hz at which to run the planning loop. And the `planner_patiente` parameter configures how long the planner will wait in seconds in an attempt to find a valid plan, before space-clearing operations are performed.

You can configure the dimensions and the resolution of the costmap with the `width`, `height`, and `resolution` parameters. The values are given in meters.

Base local planner configuration

Once we have the costmaps configured, it is necessary to configure the base planner. The base planner is used to generate the velocity commands to move our robot. Create a new file in `chapter6_tutorials/launch` with the name `base_local_planner_params.yaml`, and add the following code:

```
TrajectoryPlannerROS:  
  max_vel_x: 0.2  
  min_vel_x: 0.05  
  max_rotational_vel: 0.15  
  min_in_place_rotational_vel: 0.01  
  min_in_place_vel_theta: 0.01  
  max_vel_theta: 0.15  
  min_vel_theta: -0.15  
  acc_lim_th: 3.2  
  acc_lim_x: 2.5  
  acc_lim_y: 2.5  
  holonomic_robot: false
```

The `config` file will set the maximum and minimum velocities for your robot. The acceleration is also set.

The `holonomic_robot` parameter is `true` if you are using a holonomic platform. In our case, our robot is based on a non-holonomic platform and the parameter is set to `false`. A **holonomic vehicle** is one that can move in all the configured space from any position. In other words, if the places where the robot can go are defined by any `x` and `y` values in the environment, this means that the robot can move there from any position. For example, if the robot can move forward, backward, and laterally, it is holonomic. A typical case of a non-holonomic vehicle is a car, as it cannot move laterally, and from a given position, there are many other positions (or poses) that are not reachable. Also, a differential platform is non-holonomic.

Creating a launch file for the navigation stack

Now we have all the files created and the navigation stack is configured. To run everything, we are going to create a launch file. Create a new file in the `chapter6_tutorials/launch` folder, and put the following code in a file with the name `move_base.launch`:

```
&lt;launch>
  &lt;!-- Run the map server -->
  &lt;node name="map_server" pkg="map_server" type="map_server"
    args="$(find chapter6_tutorials)/maps/map.yaml" output="screen"
  />
  &lt;include file="$(find amcl)/examples/amcl_diff.launch" />
  &lt;node pkg="move_base" type="move_base" respawn="false"
    name="move_base" output="screen">    &lt;rosparam file="$(find
    chapter6_tutorials)/launch/costmap_common_params.yaml"
    command="load" ns="global_costmap" />    &lt;rosparam file="$(find
    chapter6_tutorials)/launch/costmap_common_params.yaml"
    command="load" ns="local_costmap" />    &lt;rosparam file="$(find
    chapter6_tutorials)/launch/local_costmap_params.yaml"
    command="load" />    &lt;rosparam file="$(find
    chapter6_tutorials)/launch/global_costmap_params.yaml"
    command="load" />    &lt;rosparam file="$(find
    chapter6_tutorials)/launch/base_local_planner_params.yaml"
    command="load" />
  &lt;/node>
&lt;/launch>
```

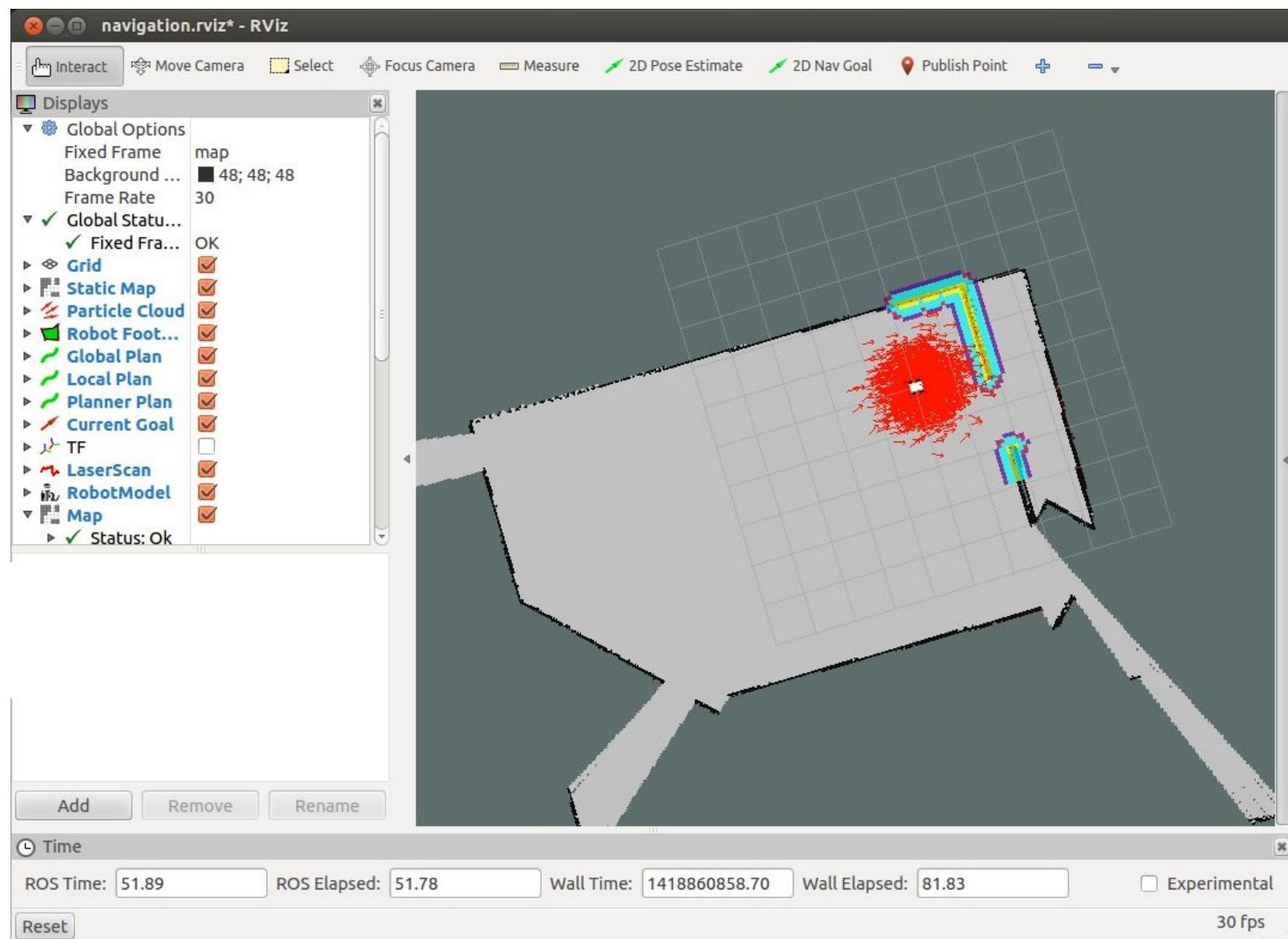
Notice that in this file, we are launching all the files created earlier. We will launch a map server as well with a map that we created in Chapter 4, *The Navigation Stack - Robot Setups* and the `amcl` node.

The `amcl` node that we are going to use is for differential robots because our robot is also a differential robot. If you want to use `amcl` with holonomic robots, you will need to use the `amcl_omni.launch` file. If you want to use another map, go to Chapter 4, *The Navigation Stack - Robot Setups*, and create a new one.

Now launch the file and type the next command in a new shell. Recall that before you launch this file, you must launch the `chapter6_configuration_gazebo.launch` file:

```
$ rosrun chapter6_tutorials move_base.launch
```

You will see the following window:



If you compare this image with the image that you saw when you launched the `chapter6_configuration_gazebo.launch` file, you will see that all the options are in blue; this is a good signal and it means that everything is okay.

As we said before, in the next section you will learn which options are necessary to visualize all the topics used in a navigation stack.

Setting up rviz for the navigation stack

It is good practice to visualize all possible data that the navigation stack does. In this section, we will show you the visualization topic that you must add to `rviz` to see the correct data sent by the navigation stack. Discussions on each visualization topic that the navigation stack publishes are given next.

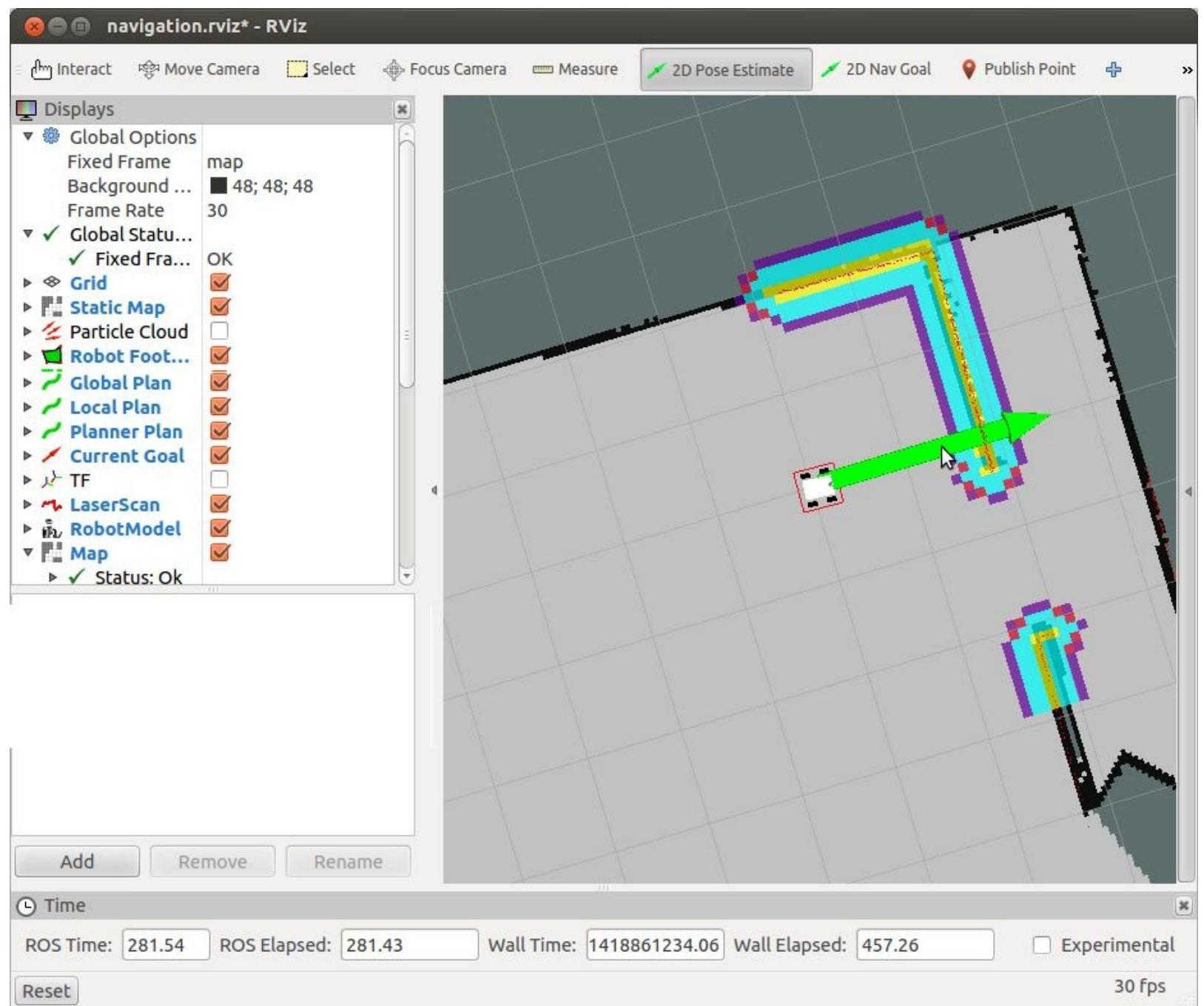
The 2D pose estimate

The **2D pose estimate** (P shortcut) allows the user to initialize the localization system used by the navigation stack by setting the pose of the robot in the world.

The navigation stack waits for the new pose of a new topic with the name `initialpose`. This topic is sent using the `rviz` windows where we previously changed the name of the topic.

You can see in the following screenshot how you can use `initialpose`. Click on the 2D Pose Estimate button, and click on the map to indicate the initial position of your robot. If you don't do this at the beginning, the robot will start the auto-localization process and try to set an initial pose:

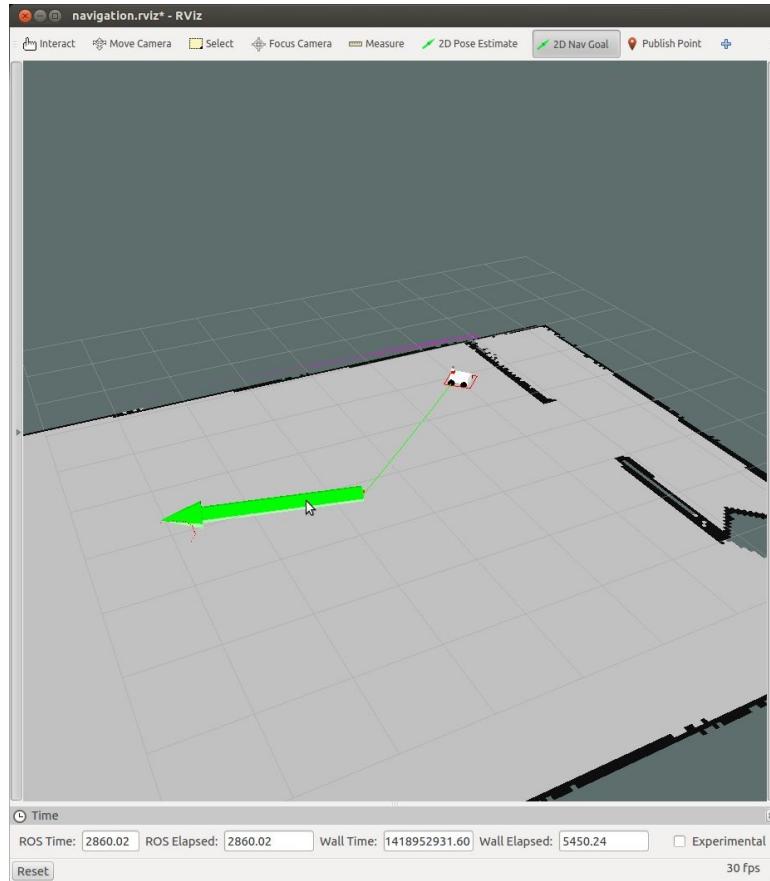
- **Topic:** `initialpose`
- **Type:** `geometry_msgs/PoseWithCovarianceStamped`



The 2D nav goal

The **2D nav goal** (G shortcut) allows the user to send a goal to the navigation by setting a desired pose for the robot to achieve. The navigation stack waits for a new goal with `/move_base_simple/goal` as the topic name; for this reason, you must change the topic's name in the `rviz` windows in Tool Properties in the 2D Nav Goal menu. The new name that you must put in this textbox is `/move_base_simple/goal`. In the next window, you can see how to use it. Click on the 2D Nav Goal button, and select the map and the goal for your robot. You can select the x and y position and the end orientation for the robot:

- **Topic:** `move_base_simple/goal`
- **Type:** `geometry_msgs/PoseStamped`

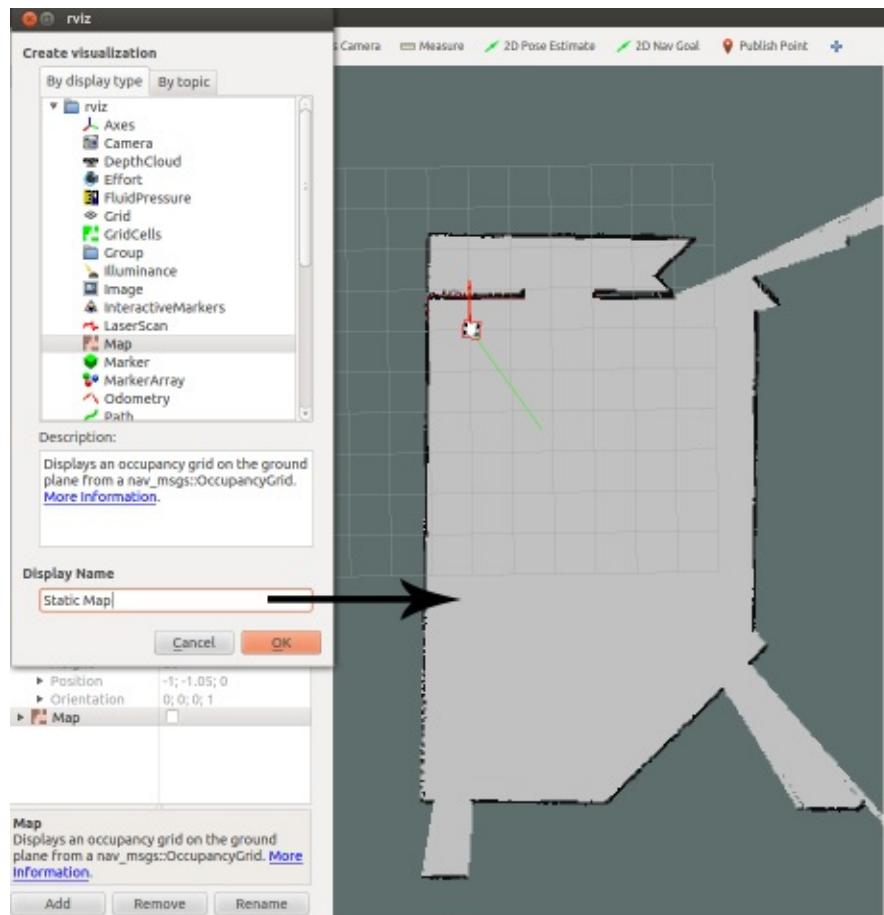


The static map

This displays the static map that is being served by `map_server`, if one exists. When you add this visualization, you will see the map we captured in Chapter 4, *The Navigation Stack - Robot Setups*, in the *Creating a map with ROS* section.

In the next window, you can see the display type that you need to select and the name that you must put in the display name:

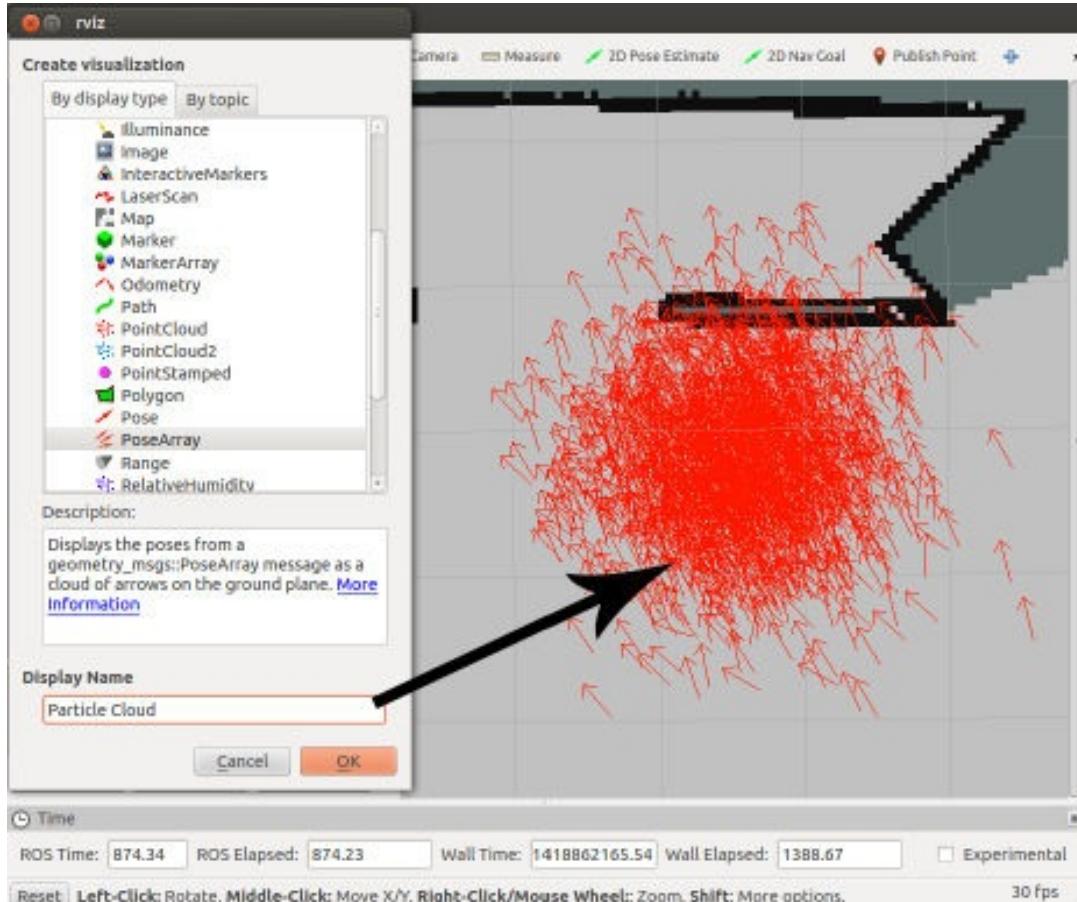
- **Topic:** `map`
- **Type:** `nav_msgs/GetMap`



The particle cloud

This displays the particle cloud used by the robot's localization system. The spread of the cloud represents the localization system's uncertainty about the robot's pose. A cloud that spreads out a lot reflects high uncertainty, while a condensed cloud represents low uncertainty. In our case, you will obtain the following cloud for the robot:

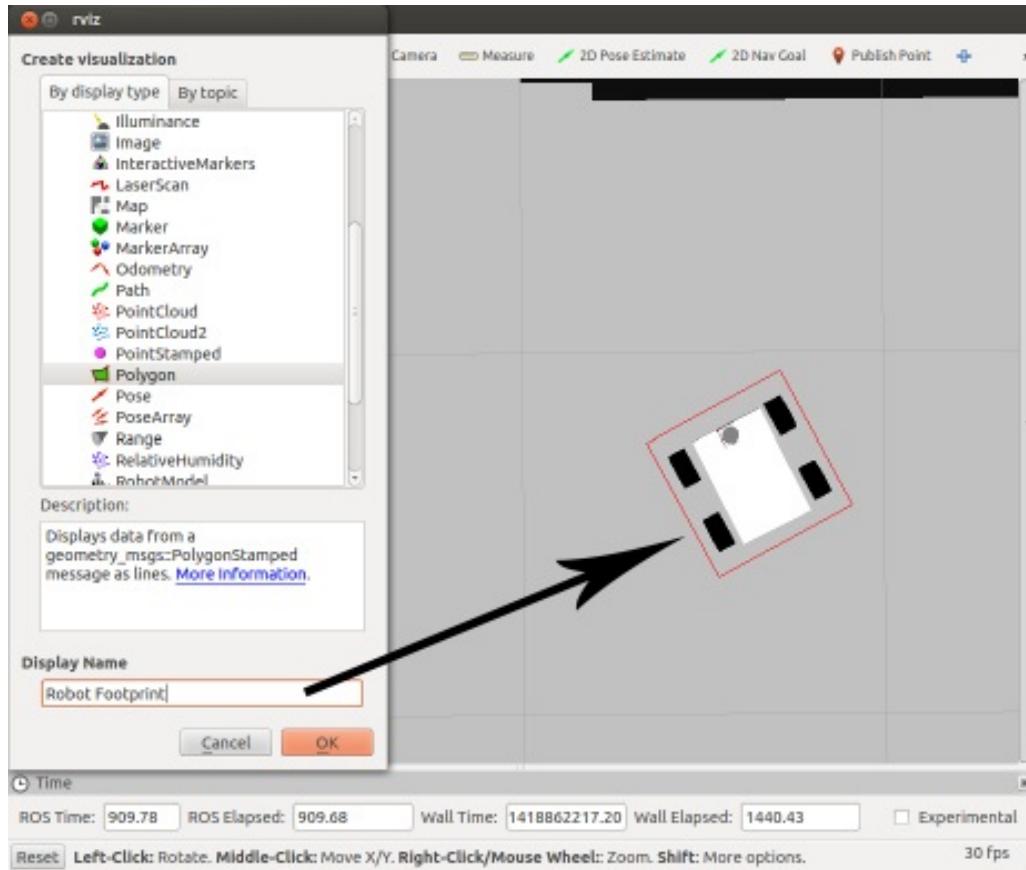
- **Topic:** particlecloud
- **Type:** geometry_msgs/PoseArray



The robot's footprint

This shows the footprint of the robot; in our case, the robot has a footprint, which has a width of 0.4 meters and a height of 0.4 meters. Remember that this parameter is configured in the `costmap_common_params` file. This dimension is important because the navigation stack will move the robot in a safe mode using the values configured before:

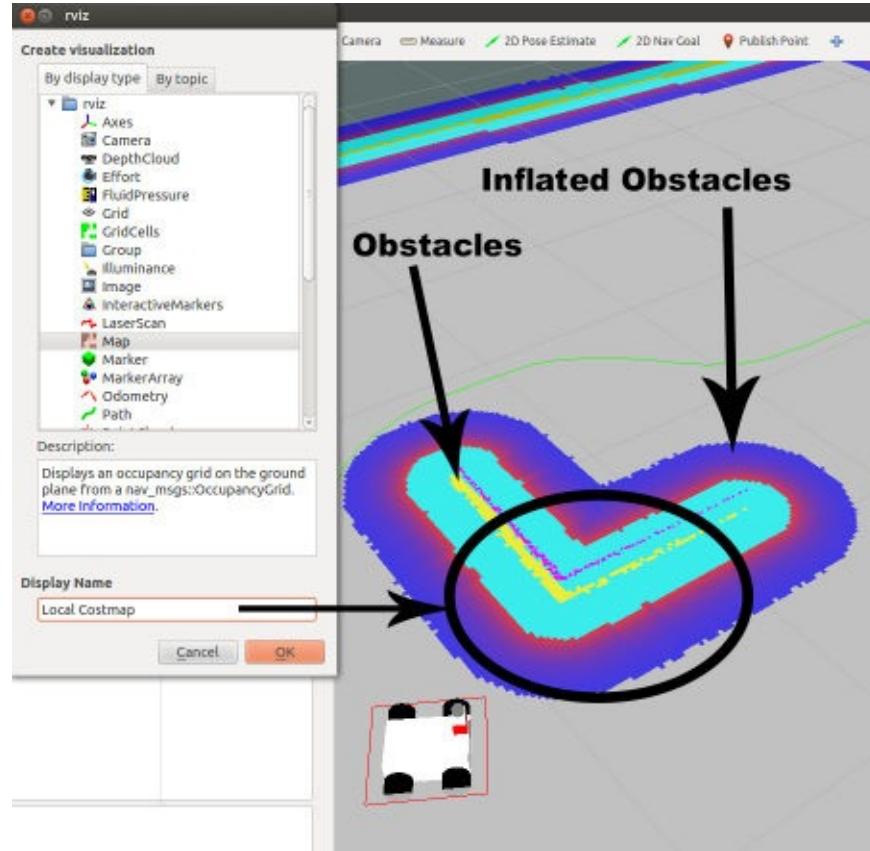
- **Topic:** `local_costmap/robot_footprint`
- **Type:** `geometry_msgs/Polygon`



The local costmap

This shows the local costmap that the navigation stack uses for navigation. The yellow line is the detected obstacle. For the robot to avoid collision, the robot's footprint should never intersect with a cell that contains an obstacle. The blue zone is the inflated obstacle. To avoid collisions, the center point of the robot should never overlap with a cell that contains an inflated obstacle:

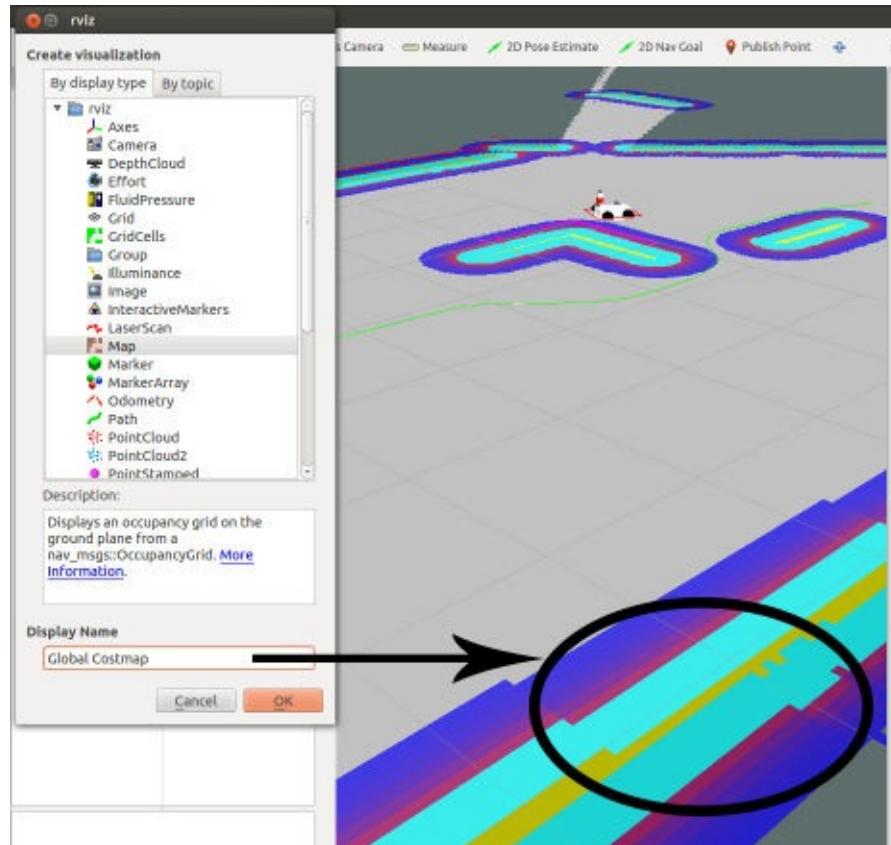
- **Topic:** /move_base/local_costmap/costmap
- **Type:** nav_msgs/OccupancyGrid



The global costmap

This shows the global costmap that the navigation stack uses for navigation. The yellow line is the detected obstacle. For the robot to avoid collision, the robot's footprint should never intersect with a cell that contains an obstacle. The blue zone is the inflated obstacle. To avoid collisions, the center point of the robot should never overlap with a cell that contains an inflated obstacle:

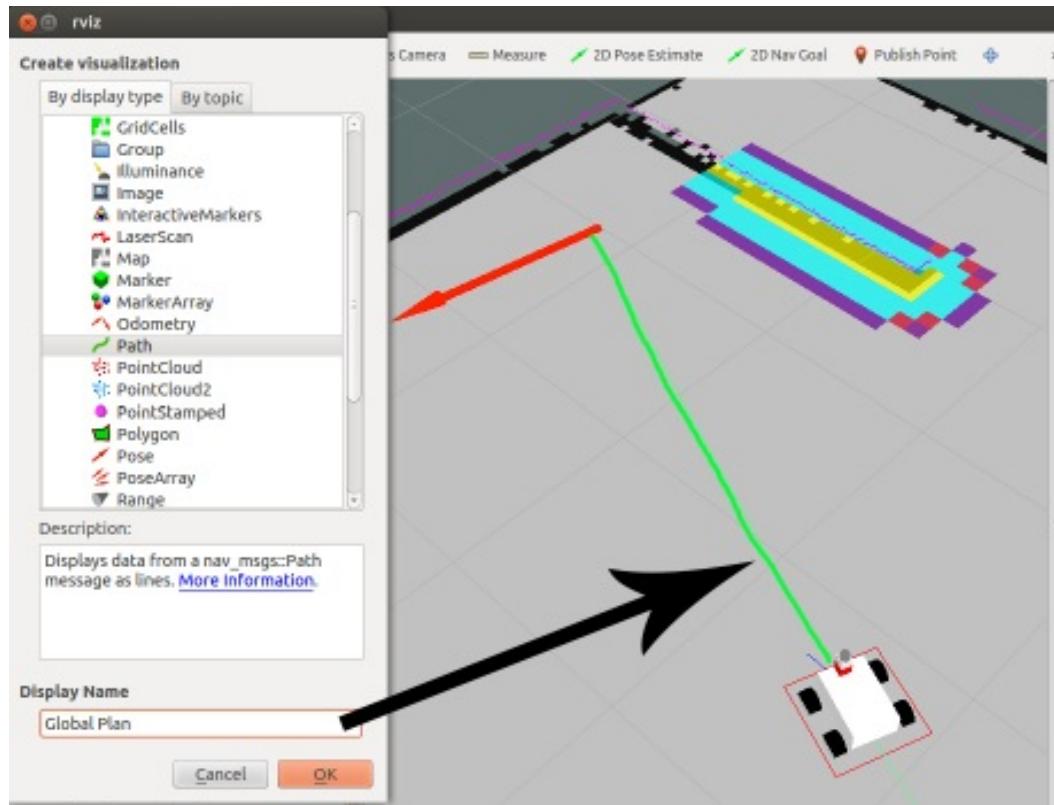
- **Topic:** /move_base/global_costmap/costmap
- **Type:** nav_msgs/OccupancyGrid



The global plan

This shows the portion of the global plan that the local planner is currently pursuing. You can see it in green in the next image. Perhaps the robot will find obstacles during its movement, and the navigation stack will recalculate a new path to avoid collisions and try to follow the global plan.

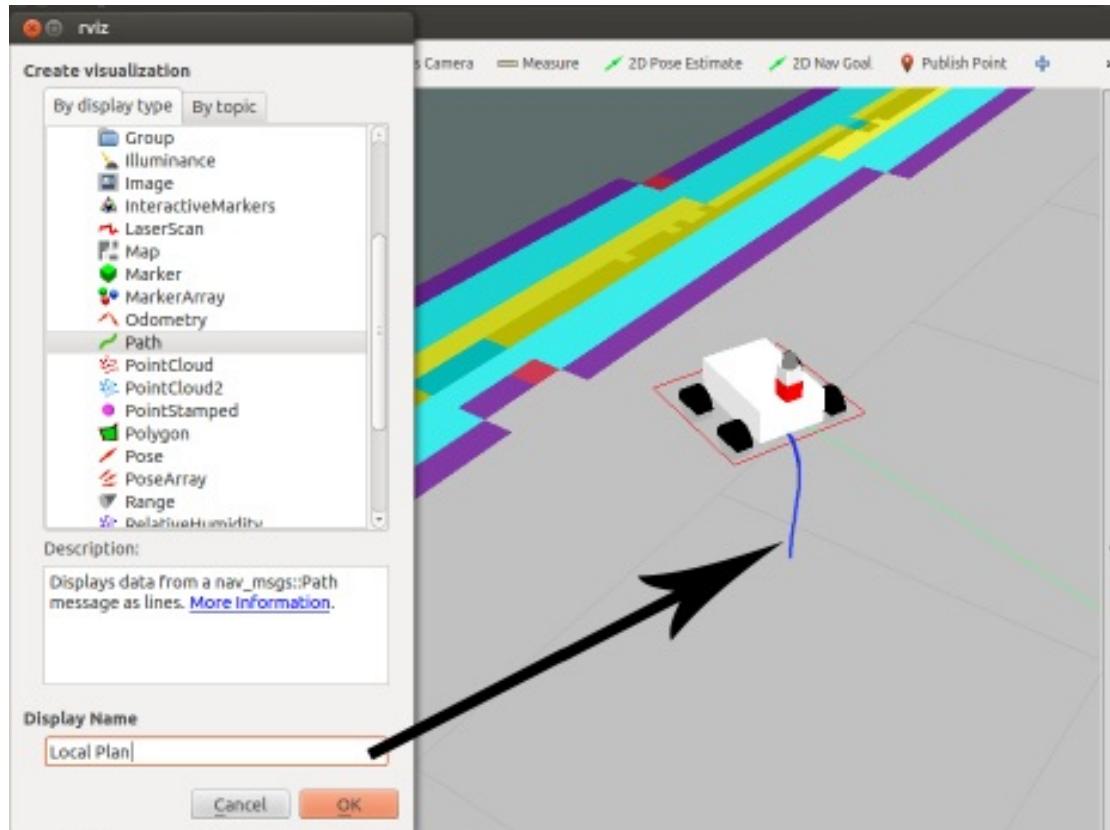
- **Topic:** TrajectoryPlannerROS/global_plan
- **Type:** nav_msgs/Path



The local plan

This shows the trajectory associated with the velocity commands currently being commanded to the base by the local planner. You can see the trajectory in blue in front of the robot in the next image. You can use this display to see whether the robot is moving, and the approximate velocity from the length of the blue line:

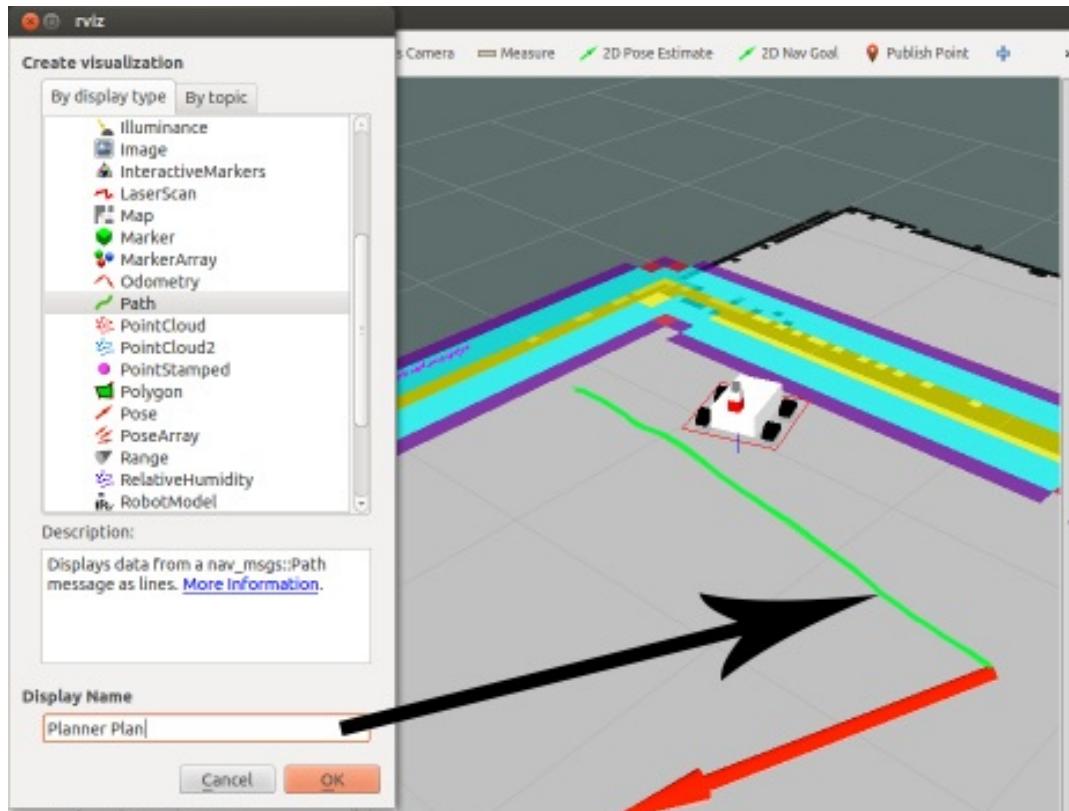
- **Topic:** TrajectoryPlannerROS/local_plan
- **Type:** nav_msgs/Path



The planner plan

This displays the full plan for the robot computed by the global planner. You will see that it is similar to the global plan:

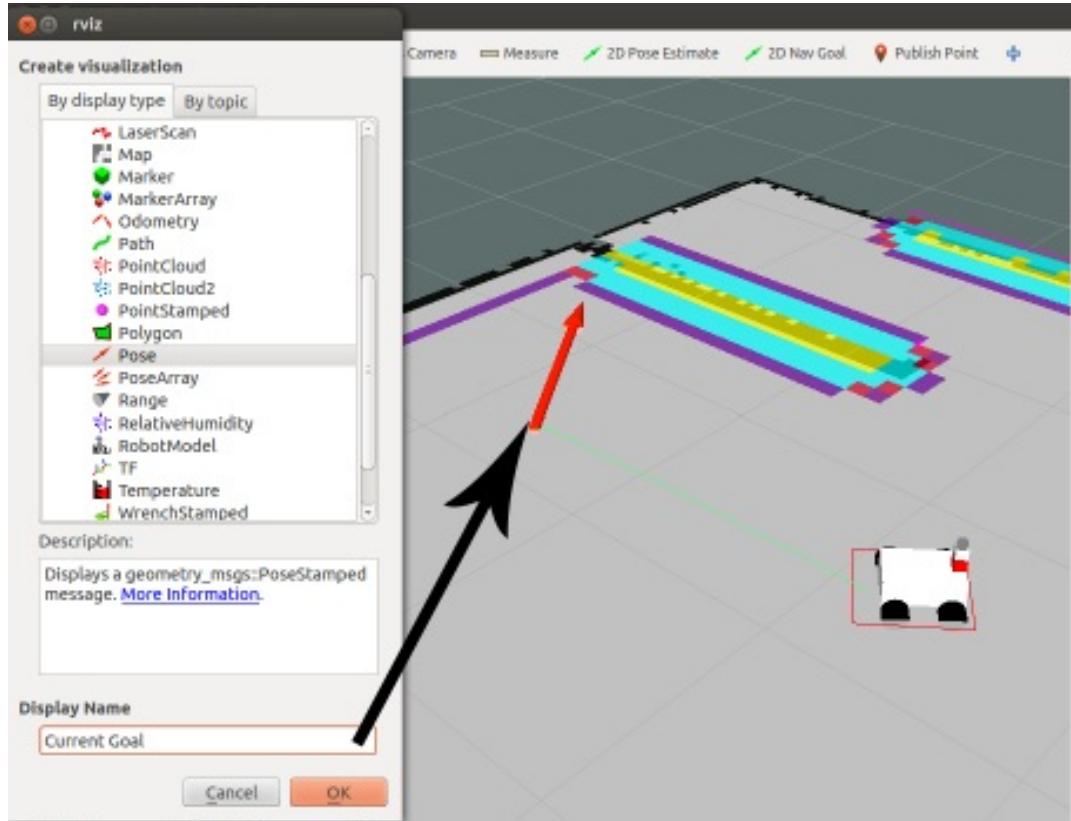
- **Topic:** NavfnROS/plan
- **Type:** nav_msgs/Path



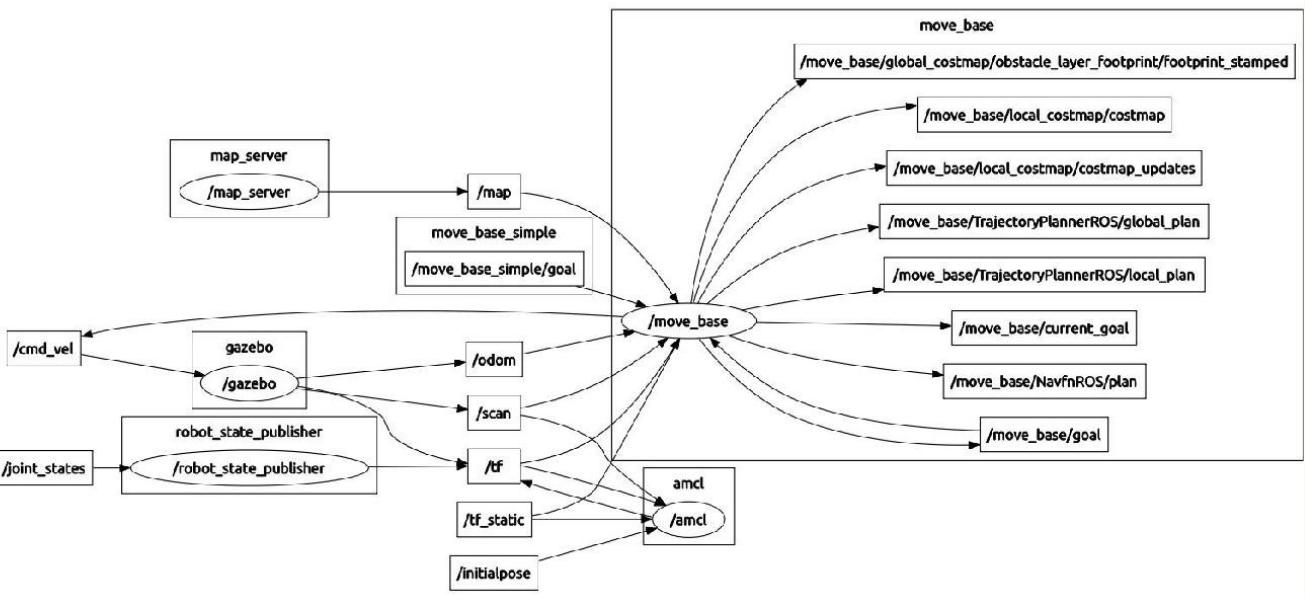
The current goal

This shows the goal pose that the navigation stack is attempting to achieve. You can see it as a red arrow, and it is displayed after you put in a new 2D nav goal. It can be used to find out the final position of the robot:

- **Topic:** current_goal
- **Type:** geometry_msgs/PoseStamped



These visualizations are all you need to see the navigation stack in `rviz`. With this, you can notice whether the robot is doing something strange. Now we are going to see a general image of the system. Run `rqt_graph` to see whether all the nodes are running and to see the relations between them.



Adaptive Monte Carlo Localization

In this chapter, we are using the **Adaptive Monte Carlo Localization (AMCL)** algorithm for the localization. The AMCL algorithm is a probabilistic localization system for a robot moving in 2D. This system implements the adaptive Monte Carlo Localization approach, which uses a particle filter to track the pose of a robot against a known map.

The AMCL algorithm has many configuration options that will affect the performance of localization. For more information on AMCL, please refer to the AMCL documentation at <http://wiki.ros.org/amcl> and also at <http://www.probabilistic-robotics.org/>.

The `amcl` node works mainly with laser scans and laser maps, but it could be extended to work with other sensor data, such as a sonar or stereo vision. So for this chapter, it takes a laser-based map and laser scans, transforms messages, and generates a probabilistic pose. On startup, the `amcl` node initializes its particle filter according to the parameters provided in the setup. If you don't set the initial position, `amcl` will start at the origin of the coordinates. Anyway, you can set the initial position in `rviz` using the 2D Pose Estimate button.

When we include the `amcl_diff.launch` file, we are starting the node with a series of configured parameters. This configuration is the default configuration and the minimum setting needed to make it work.

Next, we are going to see the content of the `amcl_diff.launch` launch file to explain some parameters:

```
&lt;launch>
  &lt;node pkg="amcl" type="amcl" name="amcl" output="screen">
    &lt;!-- Publish scans from best pose at a max of 10 Hz -->
    &lt;param name="odom_model_type" value="diff" />
    &lt;param name="odom_alpha5" value="0.1" />
    &lt;param name="transform_tolerance" value="0.2" />
    &lt;param name="gui_publish_rate" value="10.0" />
    &lt;param name="laser_max_beams" value="30" />
    &lt;param name="min_particles" value="500" />
    &lt;param name="max_particles" value="5000" />
    &lt;param name="kld_err" value="0.05" />
    &lt;param name="kld_z" value="0.99" />
    &lt;param name="odom_alpha1" value="0.2" />
    &lt;param name="odom_alpha2" value="0.2" />
    &lt;!-- translation std dev, m -->
    &lt;param name="odom_alpha3" value="0.8" />
    &lt;param name="odom_alpha4" value="0.2" />
    &lt;param name="laser_z_hit" value="0.5" />
    &lt;param name="laser_z_short" value="0.05" />
    &lt;param name="laser_z_max" value="0.05" />
    &lt;param name="laser_z_rand" value="0.5" />
    &lt;param name="laser_sigma_hit" value="0.2" />
    &lt;param name="laser_lambda_short" value="0.1" />
    &lt;param name="laser_lambda_short" value="0.1" />
    &lt;param name="laser_model_type" value="likelihood_field" />
    &lt;!--&lt;param name="laser_model_type" value="beam"/> -->
    &lt;param name="laser_likelihood_max_dist" value="2.0" />
    &lt;param name="update_min_d" value="0.2" />
    &lt;param name="update_min_a" value="0.5" />
    &lt;param name="odom_frame_id" value="odom" />
    &lt;param name="resample_interval" value="1" />
    &lt;param name="transform_tolerance" value="0.1" />
    &lt;param name="recovery_alpha_slow" value="0.0" />
    &lt;param name="recovery_alpha_fast" value="0.0" />
  &lt;/node>
```

| </launch>

The `min_particles` and `max_particles` parameters set the minimum and maximum number of particles that are allowed for the algorithm. With more particles, you get more accuracy, but this increases the use of the CPU.

The `laser_model_type` parameter is used to configure the laser type. In our case, we are using a `likelihood_field` parameter but the algorithm can also use beam lasers.

The `laser_likelihood_max_dist` parameter is used to set the maximum distance for obstacle inflation on the map, which is used in the `likelihood_field` model.

The `initial_pose_x`, `initial_pose_y`, and `initial_pose_a` parameters are not in the launch file, but they are interesting because they set the initial position of the robot when `amcl` starts; for example, if your robot always starts in the dock station and you want to set the position in the launch file.

Perhaps you should change some parameters to tune your robot and make it work fine. Visit <http://wiki.ros.org/amcl>, where you have a lot of information about the configuration and the parameters that you could change.

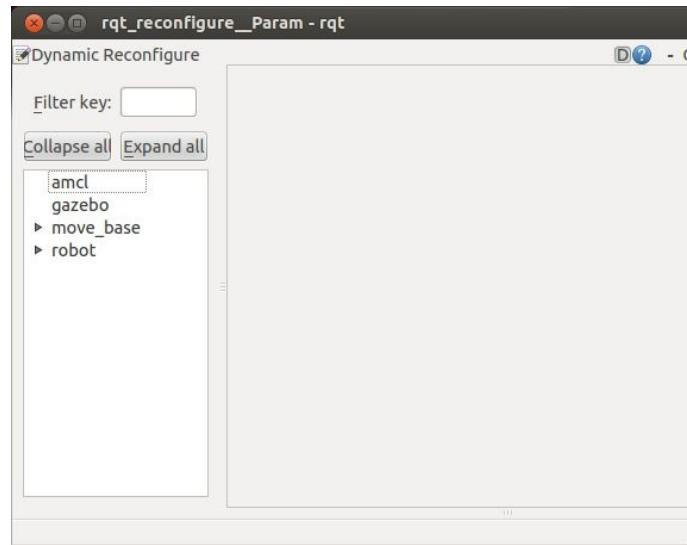
Modifying parameters with rqt_reconfigure

A good option for understanding all the parameters configured in this chapter, is by using `rqt_reconfigure` to change the values without restarting the simulation.

To launch `rqt_reconfigure`, use the following command:

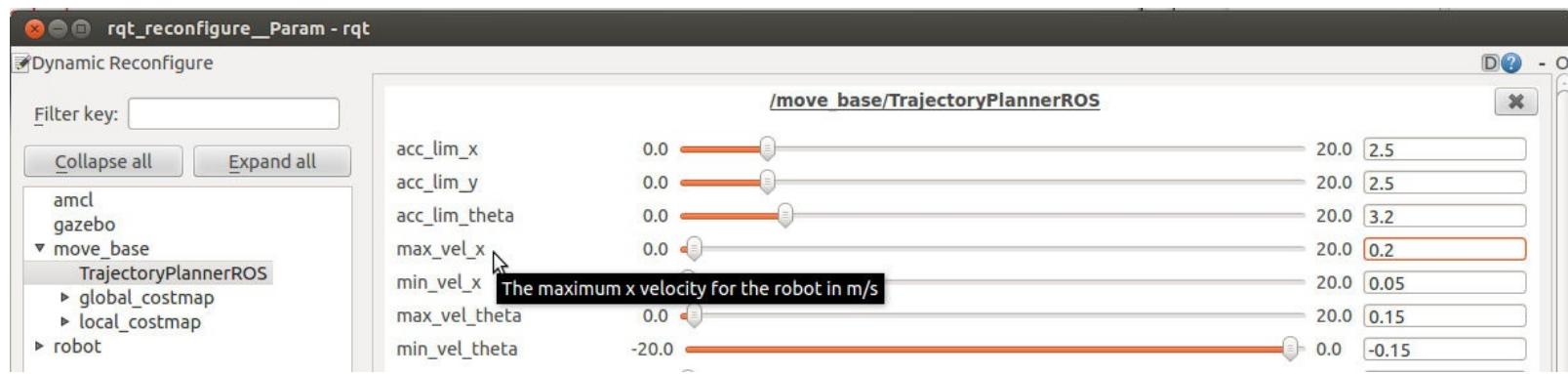
```
$ rosrun rqt_reconfigure rqt_reconfigure
```

You will see the screen as follows:



As an example, we are going to change the parameter `max_vel_x` configured in the file, `base_local_planner_params.yaml`. Click over the `move_base` menu and expand it. Then select `TrajectoryPlannerROS` in the menu tree. You will see a list of parameters. As you can see, the `max_vel_x` parameter has the same value that we assigned in the configuration file.

You can see a brief description for the parameter by hovering the mouse over the name for a few seconds. This is very useful for understanding the function of each parameter.

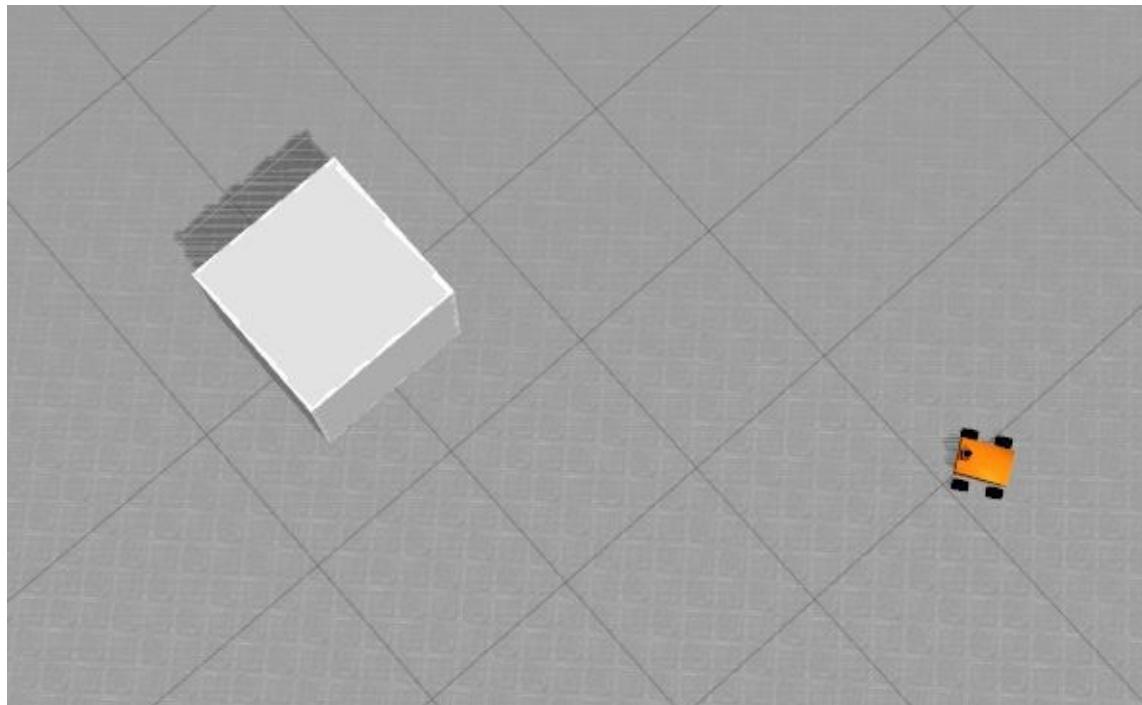


Avoiding obstacles

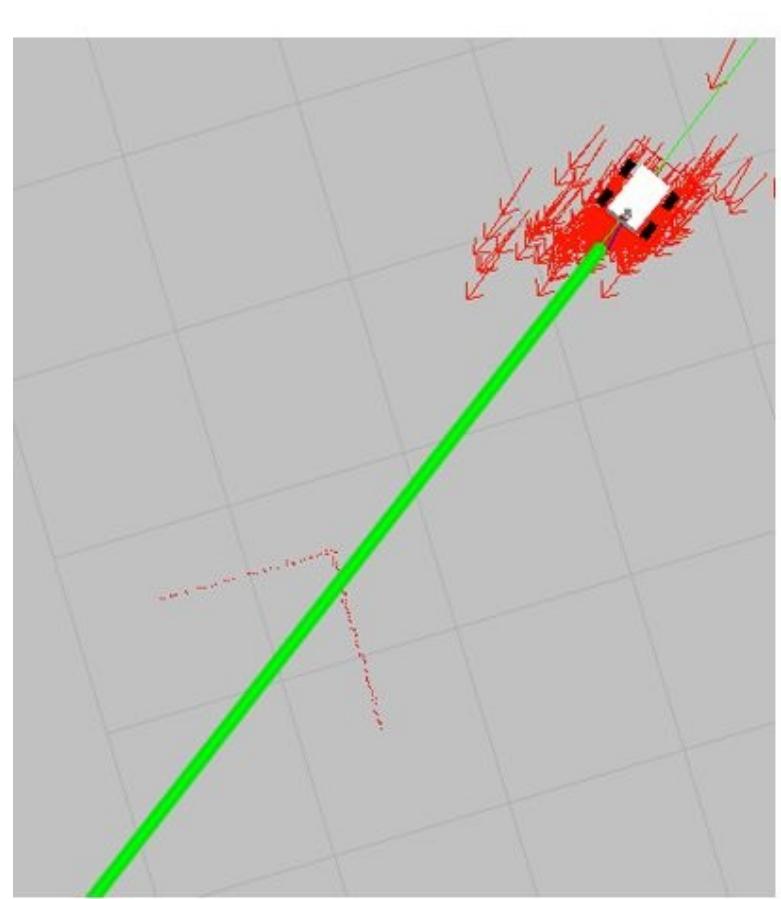
A great functionality of the navigation stack is the recalculation of the path if it finds obstacles during the movement. You can easily see this feature by adding an object in front of the robot in Gazebo. For example, in our simulation we added a big box in the middle of the path. The navigation stack detects the new obstacle, and automatically creates an alternative path.

In the next image, you can see the object that we added. Gazebo has some predefined 3D objects that you can use in the simulations with mobile robots, arms, humanoids, and so on.

To see the list, go to the Insert model section. Select one of the objects and then click at the location where you want to put it, as shown in the following screenshot:



If you go to the `rviz` windows now, you will see a new global plan to avoid the obstacle. This feature is very interesting when you use the robot in real environments with people walking around the robot. If the robot detects a possible collision, it will change the direction, and it will try to arrive at the goal. Recall that the detection of such obstacles is reduced to the area covered by the local planner costmap (for example, 4 x 4 meters around the robot). You can see this feature in the next screenshot:



Sending goals

We are sure that you have been playing with the robot by moving it around the map a lot. This is funny but a little tedious, and it is not very functional.

Perhaps you were thinking that it would be a great idea to program a list of movements and send the robot to different positions with only a button, even when we are not in front of a computer with `rviz`.

Okay, now you are going to learn how to do it using `actionlib`.

The `actionlib` package provides a standardized interface for interfacing with tasks. For example, you can use it to send goals for the robot to detect something at a place, make scans with the laser, and so on. In this section, we will send a goal to the robot, and we will wait for this task to end.

It could look similar to services, but if you are doing a task that has a long duration, you might want the ability to cancel the request during the execution, or get periodic feedback about how the request is progressing. You cannot do this with services. Furthermore, `actionlib` creates messages (not services), and it also creates topics, so we can still send the goals through a topic without taking care of the feedback and the result, if we do not want to.

The following code is a simple example for sending a goal to move the robot. Create a new file in the `chapter6_tutorials/src` folder, and add the following code in a file with the name `sendGoals.cpp`:

```
#include <ros/ros.h>
#include <move_base_msgs/MoveBaseAction.h>
#include <actionlib/client/simple_action_client.h>
#include <tf/transform_broadcaster.h>
#include <iostream>

typedef actionlib::SimpleActionClient<move_base_msgs::MoveBaseAction> MoveBaseClient;

int main(int argc, char** argv){
    ros::init(argc, argv, "navigation_goals");

    MoveBaseClient ac("move_base", true);

    while(!ac.waitForServer(ros::Duration(5.0))){
        ROS_INFO("Waiting for the move_base action server");
    }

    move_base_msgs::MoveBaseGoal goal;

    goal.target_pose.header.frame_id = "map";
    goal.target_pose.header.stamp = ros::Time::now();

    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 goal");
    ac.sendGoal(goal);

    ac.waitForResult();

    if(ac.getState() == actionlib::SimpleClientGoalState::SUCCEEDED)
        ROS_INFO("You have arrived to the goal position");
    else{
        ROS_INFO("The base failed for some reason");
    }
}
```

```
| }  
| return 0;  
| }
```

Add the next file in the `CMakeList.txt` file to generate the executable for our program:

```
| add_executable(sendGoalssrc/sendGoals.cpp)  
| target_link_libraries(sendGoals ${catkin_LIBRARIES})
```

Now compile the package with the following command:

```
| $ catkin_make
```

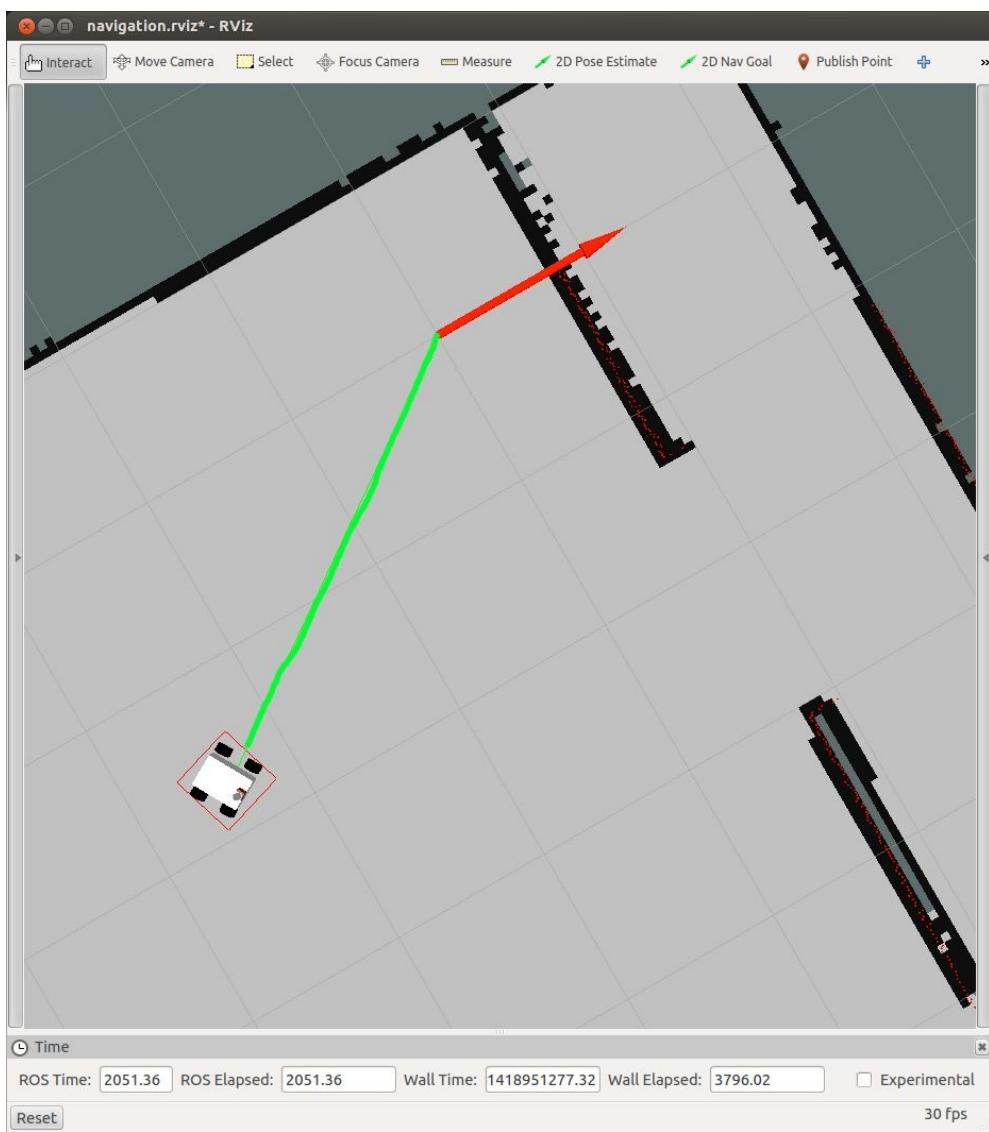
Now launch everything to test the new program. Use the following commands to launch all the nodes and the configurations:

```
| $ roslaunch chapter6_tutorials chapter6_configuration_gazebo.launch  
| $ roslaunch chapter6_tutorials move_base.launch
```

Once you have configured the 2D pose estimate, run the `sendGoal` node with the next command in a new shell:

```
| $ rosrun chapter6_tutorials sendGoals
```

If you go to the `rviz` screen, you will see a new global plan (green line) over the map. This means that the navigation stack has accepted the new goal and it will start to execute it.



When the robot arrives at the goal, you will see the following message in the shell where you ran the node:

```
[ INFO ] [....]: You have arrived to the goal position
```

You can make a list of goals or waypoints, and create a route for the robot. This way you can program missions, guardian robots, or collect things from other rooms with your robot.

Summary

At the end of this chapter, you should have a robot-simulated or real-moving autonomously through the map (which models the environment), using the navigation stack. You can program the control and the localization of the robot by following the ROS philosophy of code reusability, so that you can have the robot completely configured without much effort. The most difficult part of this chapter is to understand all the parameters and learn how to use each one of them appropriately. The correct use of them will determine whether your robot works fine or not; for this reason, you must practice changing the parameters and look for the reaction of the robot.

In the next chapter, you will learn how to use MoveIt! with some tutorials and examples. If you don't know what MoveIt! is, it is a software for building mobile manipulation applications. With it, you can move your articulated robot in an easy way.

Manipulation with MoveIt!

MoveIt! is a set of tools for mobile manipulation in ROS. The main web page (<http://moveit.ros.org>) contains documentation, tutorials, and installation instructions as well as example demonstrations with several robotic arms (or robots) that use MoveIt! for manipulation tasks, such as grasping, picking and placing, or simple motion planning with inverse kinematics.

The library incorporates a fast inverse kinematics solver (as part of the motion planning primitives), state-of-the-art algorithms for manipulation, grasping 3D perception (usually in the form of point clouds), kinematics, control, and navigation. Apart from the backend, it provides an easy-to-use GUI to configure new robotic arms with the MoveIt! and **RViz** plugins to develop motion planning tasks in an intuitive way.

In this chapter, we will see how we can create a simple robotic arm in the URDF format and how we can define motion planning groups with the MoveIt! configuration tool. For a single arm, we will have a single group, so that later we can use the inverse kinematics solvers to perform manipulation tasks specified from the RViz interface. A pick and place task is used to illustrate the capabilities and tools of MoveIt!.

The first section explains the MoveIt! architecture, explaining the basic concepts used in the framework, such as joint groups and planning scene, and general concepts such as trajectory planning, (inverse) kinematics, and collision checking concerns. Then, we will show how you can integrate an arm into MoveIt!, creating the planning groups and scene. Next, we will show you how you can perform motion planning with collisions and how you can incorporate point clouds, which will allow you to avoid collisions with dynamic obstacles.

Finally, perception and object recognition tools will be explained and later used in a pick and place demonstration. For this demonstration, we will use the MoveIt! plugin for RViz.

The MoveIt! architecture

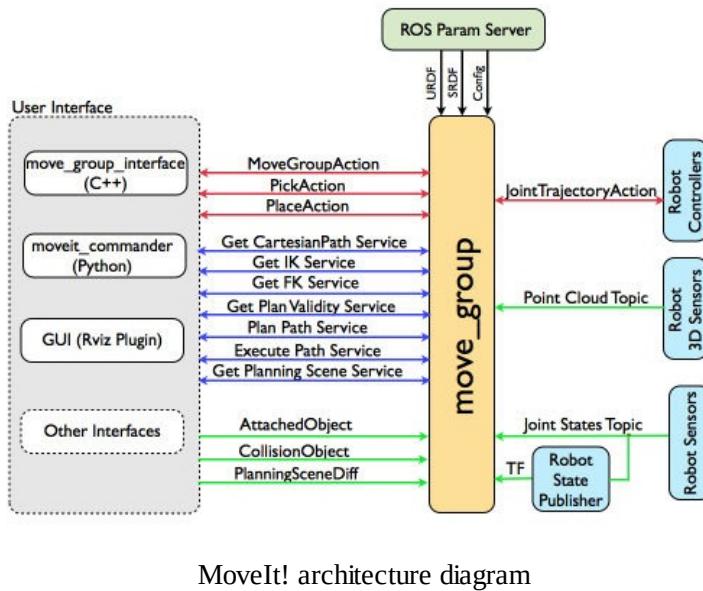
The architecture of MoveIt! is depicted in the following diagram taken from the concepts sections of its official documentation at <http://moveit.ros.org/documentation/concepts/>. Here, we describe the main concepts in brief. In order to install MoveIt!, you only have to run this command:

```
| $ sudo apt-get install ros-kinetic-moveit-full
```

Alternatively, you can install all the dependencies of the code that comes with this chapter by running the following command from a workspace that contains it:

```
| $ rosdep install --from-paths src -iy
```

The following diagram shows the architecture of MoveIt!:



MoveIt! architecture diagram

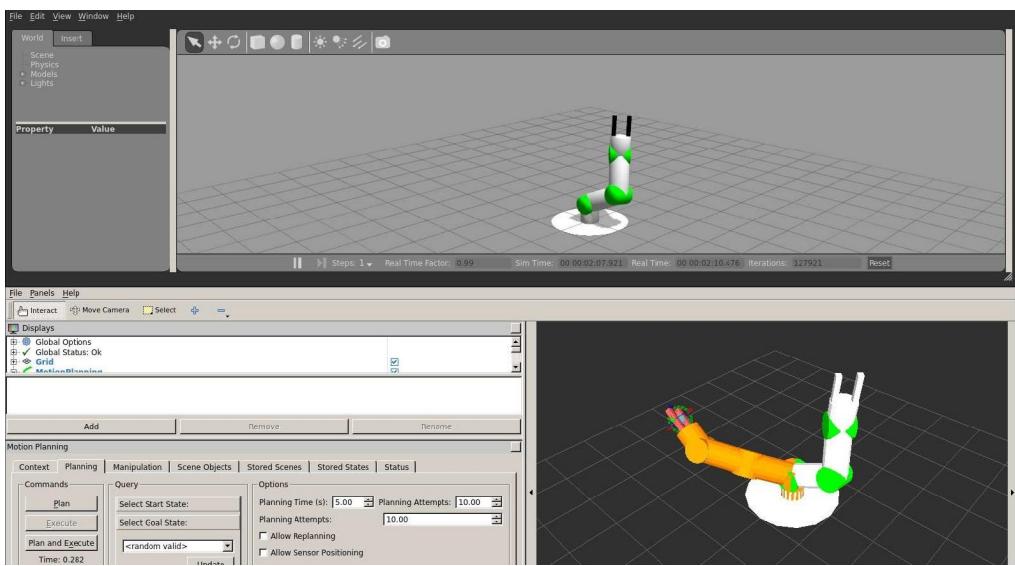
In the center of the architecture, we have the `move_group` element. The main idea is that we have to define groups of joints and other elements to perform moving actions using motion planning algorithms. These algorithms consider a scene with objects to interact with and the joints characteristics of the group.

A group is defined using standard ROS tools and definition languages, such as YAML, URDF, and SDF. In brief, we have to define the joints that are part of a group with their joint limits. Similarly, we define the end effector tools, such as a gripper and perception sensors. The robot must expose `JointTrajectoryAction` controllers so that the output of the motion planning can be planned and executed on the robot hardware (or simulator). In order to monitor the execution, `/joint_states` is also needed by means of the robot state publisher. All this is provided by the ROS control as well as specific sensor drivers. Note that MoveIt! provides a GUI wizard to define the joint groups for a given robot that can be called directly as follows:

```
| $ roslaunch moveit_setup_assistant setup_assistant.launch
```

Once `move_group` is configured properly, we can interface with it. MoveIt! provides a C++ and a Python API to do so, as well as an RViz plugin that integrates seamlessly and allows us to send motion goals, plan

them, and send (execute) them on the robot, as shown in the following figure:



MoveIt! integration for simulated manipulator in Gazebo

Motion planning

Motion planning deals with the problem of moving the arm to a configuration, allowing you to reach a pose with the end effector without crashing the move group with any obstacle, that is, the links themselves or other objects perceived by sensors (usually as point clouds) or violating the joint limits. The MoveIt! user interface allows you to use different libraries for motion planning, such as OMPL (<http://ompl.kavrakilab.org>), using ROS actions, or services.

A motion plan request is sent to the motion planning module, which takes care of avoiding collisions (including self-collisions) and finds a trajectory for all the joints in the groups that move the arm so that it reaches the goal requested. Such a goal consists of a location in joint space or an end effector pose, which could include an object (for example, if the gripper picks up something) as well as kinematic constraints, such as position, orientation, visibility and user-specified constraints.

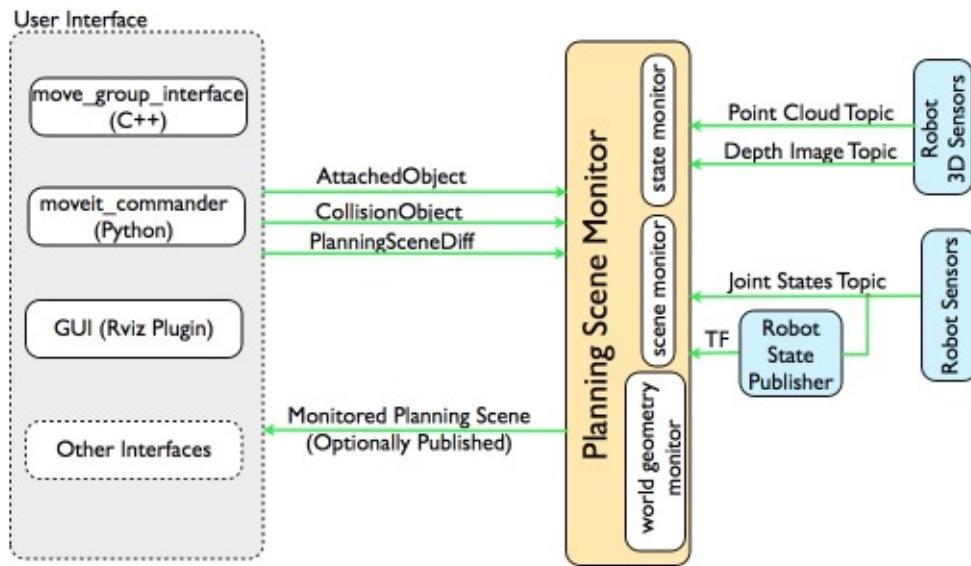
The result of the motion plan is a trajectory that moves the arm to the target goal location. This trajectory also avoids collisions and satisfies the velocity and acceleration constraints at the joint level.

Finally, MoveIt! has a motion planning pipeline made of motion planners and plan request adapters. The latter are components that allow the pre-processing and post-processing of the motion plan request. For example, pre-processing is useful when the initial state of the arm is outside joint limits; post-processing is useful to convert paths into time-parameterized trajectories.

The planning scene

The planning scene represents the world around the robot as well as the robot state. This is maintained by the planning scene monitor shown in the next diagram, taken from the concepts section of its official documentation at <http://moveit.ros.org/documentation/concepts/>. It is a subpart of `move_group`, which listens to `joint_states`, the sensor information (usually point clouds), and the world geometry, which is provided by the user input on the `planning_scene` topic.

This is shown in the following figure:



MoveIt! planning scene diagram

World geometry monitor

As the name suggests, the world geometry monitor takes care of keeping track of the different aspects of what we consider to be the world. It uses an occupancy map monitor to build a 3D representation of the environment around the robot and augments it with the `plannin_scene` topic information, such as objects (for example, grasping objects); an octomap is used to register all this information. In order to generate the 3D representation, MoveIt! supports different sensors to perceive the environment by means of plugins supplying two kinds of inputs: point clouds and depth images.

Kinematics

Forward kinematics and its Jacobians are integrated in the `RobotState` class. On the other hand, for inverse kinematics, MoveIt! provides a default plugin that uses a numerical Jacobian-based solver that is automatically configured by the Setup Assistant. As with other components of MoveIt!, users can write their own inverse kinematics plugins, such as **IKFast**.

Collision checking

The `collisionWorld` object of the planning scene is used to configure collision checking using the **Flexible Collision Library (FCL)** package. The collision objects supported are meshes, primitive shapes (for example, boxes, cylinders, cones, spheres, and planes) and an octomap.

Integrating an arm in MoveIt!

In this section, we will go through the different steps required to get a robotic arm working with MoveIt! There are several elements that need to be provided beforehand, such as the arm description file (URDF), as well as the components required to make it work in Gazebo, although some of these will be covered in this chapter.

What's in the box?

In order to make it easier to understand how we can integrate a robotic arm with MoveIt!, we have provided a set of packages containing all of the necessary configurations, robot descriptions, launch scripts, and modules to integrate MoveIt! with ROS, Gazebo, and RViz. We will not cover the details of how to integrate a robot with Gazebo as that has been covered in other chapters, but an explanation on how to integrate MoveIt! with Gazebo will be provided. The following packages are provided in the repository for this chapter, in the `chapter7_tutorials` directory:

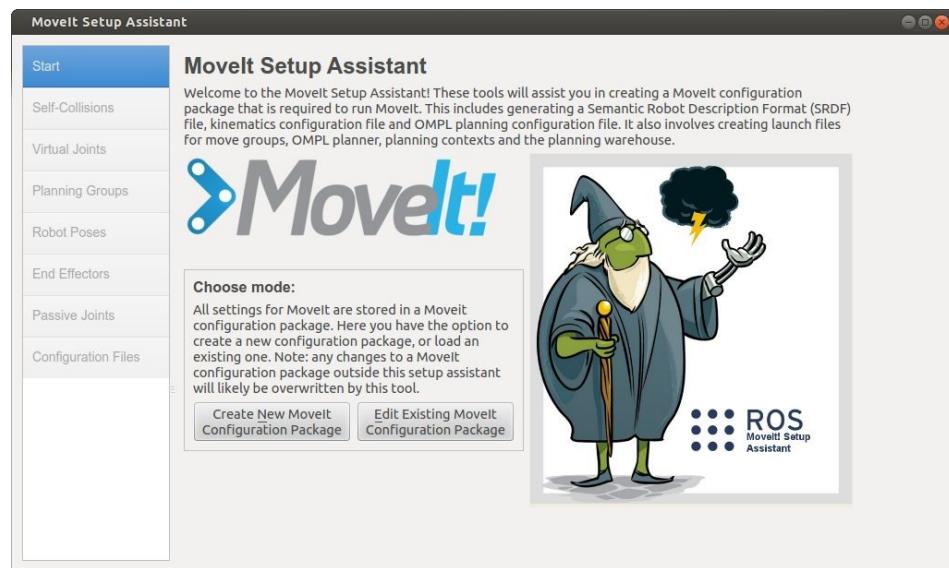
- `chapter7_tutorials`: This repository acts as a container for the rest of the packages that will be used in this chapter. This sort of structure usually requires a metapackage to let catkin know that the packages are loosely related; hence, this package is the metapackage of the repository.
- `rosbook_arm_bringup`: This package centralizes the launching of the controllers and MoveIt!. It brings up the robot-either the real one or in simulation.
- `rosbook_arm_controller_configuration`: This package contains the launch files to load the controllers required to move the arm. These are trajectory (`JointTrajectoryController`) controllers used to support the MoveIt! motion planning.
- `rosbook_arm_controller_configuration_gazebo`: This package contains the configuration for the joint trajectory controllers. This configuration also includes the PID values required to control the arm in Gazebo.
- `rosbook_arm_description`: This package contains all of the required elements to describe the robotic arm, including URDF files (actually Xacro), meshes, and configuration files.
- `rosbook_arm_gazebo`: This package is one of the most important packages, containing the launch files for Gazebo (which will take care of launching the simulation environment as well as MoveIt!) and the controllers, as well as taking care of running the launch files required (mainly calling the launch file in `rosbook_arm_bringup` but also all the previous packages). It also contains the world's descriptions in order to include objects to interact with.
- `rosbook_arm_hardware_gazebo`: This package uses the ROS Control plugin used to simulate the joints in Gazebo. This package uses the robot description to register the different joints and actuators, in order to be able to control their position. This package is completely independent of MoveIt!, but it is required for the integration with Gazebo.
- `rosbook_arm_moveit_config`: This package is generated through the MoveIt! Setup Assistant. This contains most of the launch files required for both MoveIt! and the RViz plugins as well as several configuration files for MoveIt!.
- `rosbook_arm_snippets`: Except for the pick and place example, this package contains all of the snippets used throughout the chapter.
- `rosbook_arm_pick_and_place`: This package is the biggest and most complex example in the section, containing a demonstration of how you can perform object picking and placing with MoveIt!.

Generating a MoveIt! package with the Setup Assistant

MoveIt! provides a user-friendly graphical interface for the purpose of integrating a new robotics arm into it. The **Setup Assistant** takes care of generating all of the configuration files and launch scripts based on the information provided by the user. In general, it is the easiest way to start using MoveIt! as it also generates several demonstration launch scripts, which can be used to run the system without a physical arm or simulation in place.

In order to launch the Setup Assistant, the following command needs to be executed in a terminal:

```
| $ roslaunch moveit_setup_assistant setup_assistant.launch
```

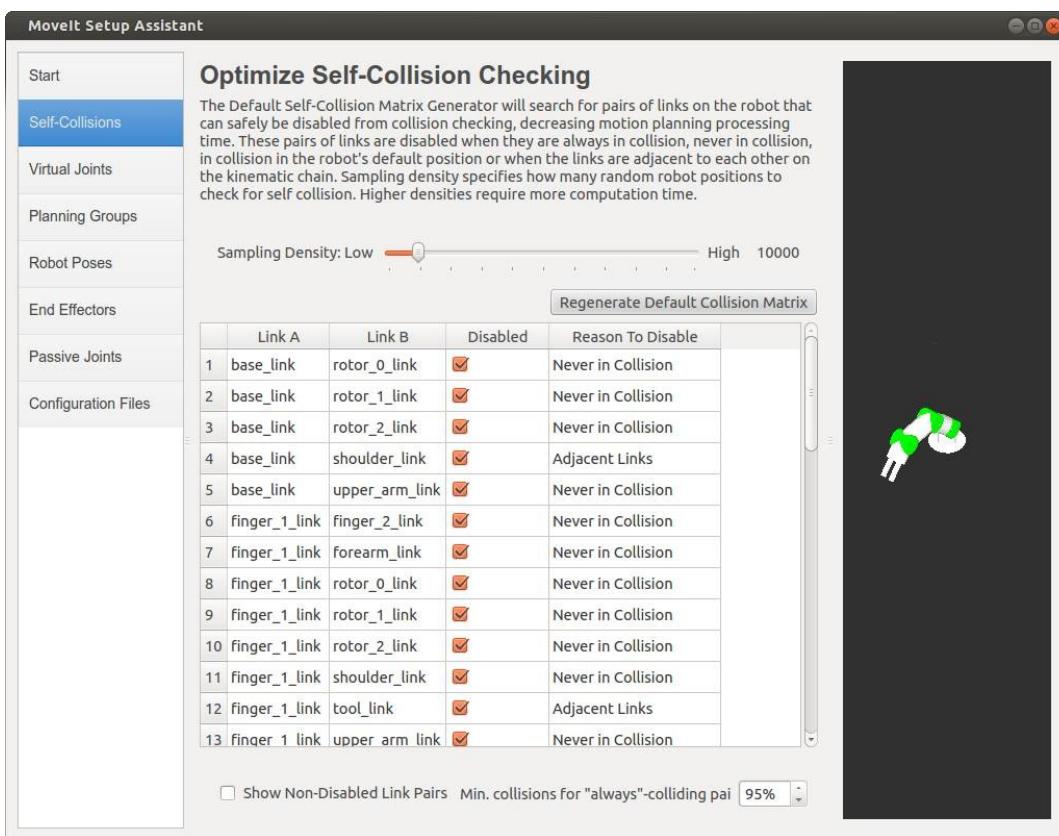


Initial screen of MoveIt! setup assistant

Once the command has been executed, a window similar to the one shown above should appear; in this particular case, the goal is to generate a new configuration, so that's the button we should aim for. Once the button has been pressed, the assistant will request a URDF or COLLADA model of the robotic arm, which for our example arm can be found at `rosbook_arm_description/robots/rosbook_arm_base.urdf.xacro` inside the repository package.

Please note that the robot description provided is in the **XML Macros (Xacro)** format, which makes it easier to generate complex URDF files. Once the robot description has been loaded, the reader needs to go through each tab and add the required information. The first tab, as seen in following figure, is used to generate the self-collision matrix.

Fortunately for the user, this process is performed automatically by simply setting the sampling density (or using the default value), and clicking on the Regenerate Default Collision Matrix button. The collision matrix contains information about how and when links collide in order to improve the performance of the motion planner. The figure below shows this in detail:



Self-collision tab of MoveIt! Setup Assistant

The second tab, as seen in the following figure, is used to assign virtual joints to the robot. A virtual joint is used to attach the robotic arm to the world as the pose of a robot can vary with respect to it, but in this particular case we won't need a virtual joint because the base of the arm does not move. We need virtual joints when the manipulator is not fixed in one place. For example, if the arm is on top of a mobile platform, we need a virtual joint for the odometry since `base_link` (base frame) moves with respect to the `odom` frame.

| |
|-----------------------|
| Start |
| Self-Collisions |
| Virtual Joints |
| Planning Groups |
| Robot Poses |
| End Effectors |
| Passive Joints |
| Configuration Files |

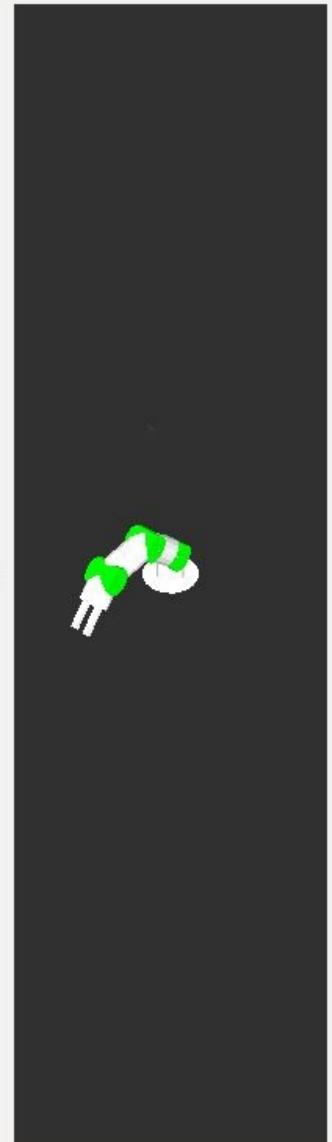
Virtual Joints

Define a virtual joint between a robot link and an external frame of reference (considered fixed with respect to the robot).

| Virtual Joint Name | Child Link | Parent Frame | Type |
|--------------------|------------|--------------|------|
| | | | |

[Delete Selected](#)

[Add Virtual Joint](#)



Virtual joints tab of MoveIt! Setup Assistant

In the third tab, which can be seen in the following figure, we need to define the planning groups of the robotic arm. Planning groups, as the name suggests, are sets of joints that need to be planned as a group in order to achieve a given goal on a specific link or end effector. In this particular case, we need to define two planning groups: one for the arm itself and another for the gripper. The planning will then be performed separately for the arm positioning and the gripper action.

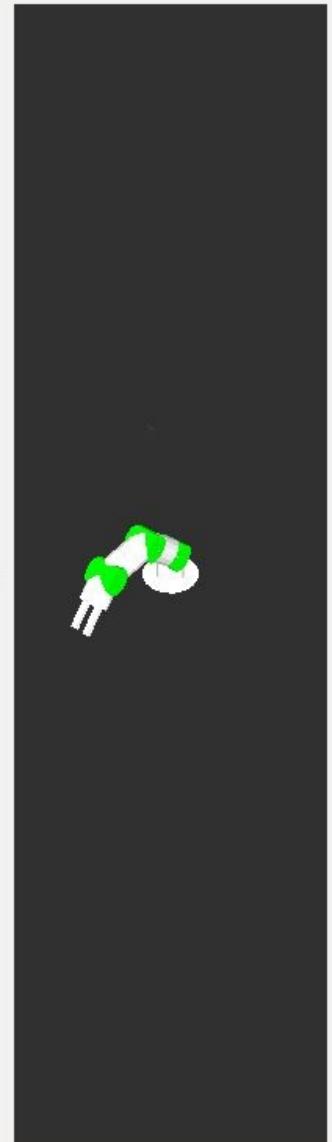
- Start
- Self-Collisions
- Virtual Joints
- Planning Groups**
- Robot Poses
- End Effectors
- Passive Joints
- Configuration Files

Planning Groups

Create and edit planning groups for your robot based on joint collections, link collections, kinematic chains and subgroups.

Current Groups

- ▼ arm
 - ▼ Joints
 - shoulder_joint - Revolute
 - rotor_0_joint - Revolute
 - upper_arm_joint - Revolute
 - rotor_1_joint - Revolute
 - forearm_joint - Revolute
 - rotor_2_joint - Revolute
 - tool_joint - Revolute
 - grasping_frame_joint - Fixed
 - Links
 - Chain
 - Subgroups
- ▼ gripper
 - ▼ Joints
 - finger_1_joint - Prismatic
 - finger_2_joint - Prismatic
 - Links
 - Chain
 - Subgroups

[Expand All](#) [Collapse All](#)[Delete Selected](#)[Edit Selected](#)[Add Group](#)

Planning Groups tab of MoveIt! Setup Assistant

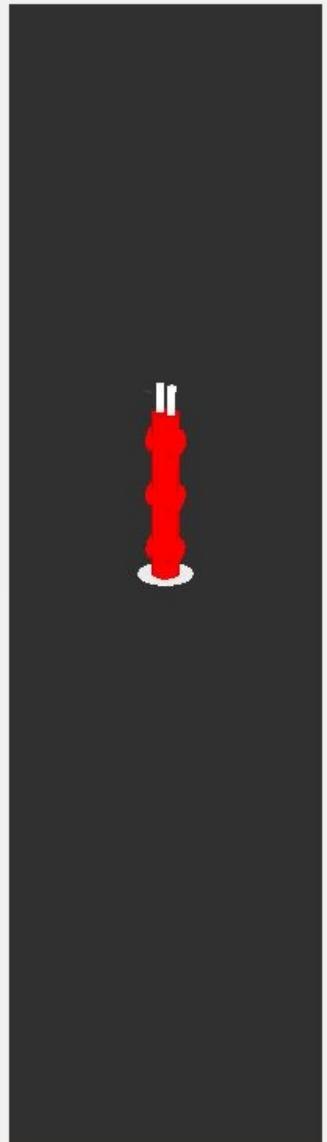
The fourth tab, as shown in the next screenshot, gives us the ability to define known robot poses in order to be able to reference them later; these predefined poses are also referred to as group states. As we can see, we have set up two different poses: the home position, which corresponds to the *stored* position of the arm, and the grasping position, which, as the name suggests, should allow the robot to grasp elements in the scene. Setting known poses can have multiple benefits in a real-life situation; for example, it is common to have an initial position from which planning happens, a position where the arm is safe to be stored in a container, or even a set of known positions with which to compare the position accuracy over time.

- Start
- Self-Collisions
- Virtual Joints
- Planning Groups
- Robot Poses**
- End Effectors
- Passive Joints
- Configuration Files

Robot Poses

Create poses for the robot. Poses are defined as sets of joint values for particular planning groups. This is useful for things like *folded arms*.

| | Pose Name | Group Name |
|---|-----------|------------|
| 1 | home | arm |
| 2 | grasp | arm |



[Show Default Pose](#) [MoveIt!](#) [Edit Selected](#) [Delete Selected](#) [Add Pose](#)

Robot Poses tab of MoveIt! Setup Assistant

The fifth tab is used to define the robotic arm's, can be seen in the following figure. As we discussed earlier, the robotic arm usually has an end effector, which is used to perform an action, such as a gripper or some other tool. In our case, the end effector is a gripper, which allows us to pick objects from the scene. In this tab, we need to define the gripper's end effector by assigning it a name, a planning group, and the parent link containing the end effector.

Start

Self-Collisions

Virtual Joints

Planning Groups

Robot Poses

End Effectors

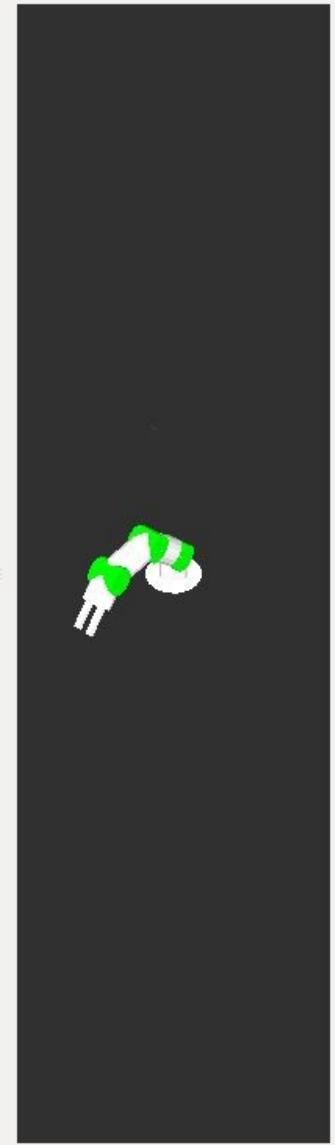
Passive Joints

Configuration Files

End Effectors

Setup grippers and other end effectors for your robot

| | End Effector Name | Group Name | Parent Link | Parent Group |
|---|-------------------|------------|----------------|--------------|
| 1 | gripper_eef | gripper | grasping_frame | |

[Edit Selected](#) [Delete Selected](#) [Add End Effector](#)

End Effectors tab of MoveIt! Setup Assistant

The sixth tab, shown in the following screenshot, is an optional configuration step, which allows us to define joints that cannot be actuated. An important feature of these joints is that MoveIt! doesn't need to plan for them and our modules don't need to publish information about them. An example of a passive joint in a robot could be a caster, but in this case, we'll skip this step as all of our passive joints have been defined as fixed joints, which will eventually produce the same effect on motion planning.

| |
|---------------------|
| Start |
| Self-Collisions |
| Virtual Joints |
| Planning Groups |
| Robot Poses |
| End Effectors |
| Passive Joints |
| Configuration Files |

Passive Joints

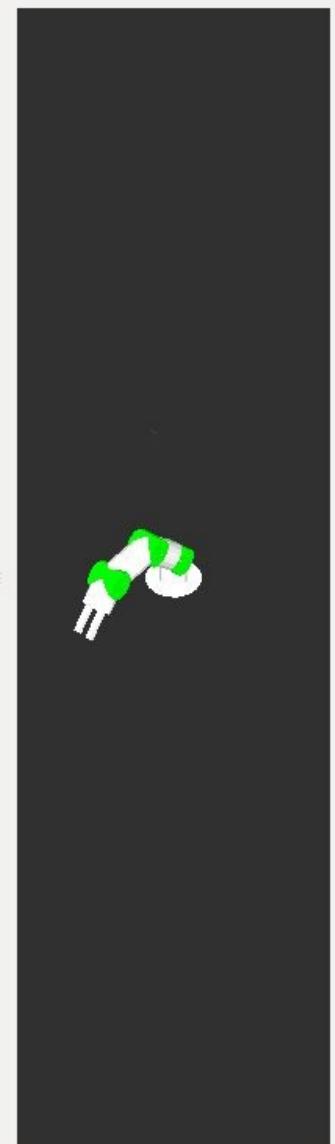
Specify the set of passive joints (not actuated). Joint state is not expected to be published for these joints.

Active Joints

| | Joint Names |
|---|-----------------|
| 1 | shoulder_joint |
| 2 | rotor_0_joint |
| 3 | upper_arm_joint |
| 4 | rotor_1_joint |
| 5 | forearm_joint |
| 6 | rotor_2_joint |
| 7 | tool_joint |
| 8 | finger_1_joint |
| 9 | finger_2_joint |

Passive Joints

| | Joint Names |
|--|-------------|
| | |



Passive Joints tab of MoveIt! Setup Assistant

Finally, as seen in the following diagram, the last step in the Setup Assistant is generating the configuration files. The only thing required in this step is to provide the path of the configuration package, which will be created by MoveIt!, and which will contain most of the launch and configuration files required to properly start controlling our robotic arm from MoveIt!

Start

Self-Collisions

Virtual Joints

Planning Groups

Robot Poses

End Effectors

Passive Joints

Configuration Files

Generate Configuration Files

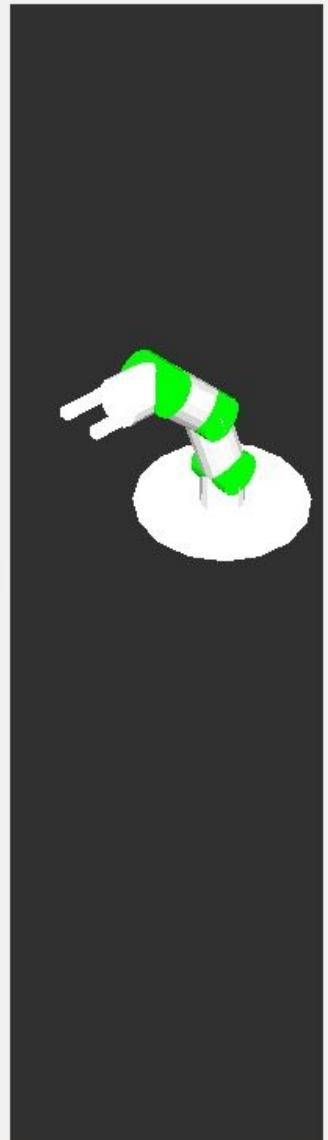
Create or update the configuration files package needed to run your robot with MoveIt. Uncheck files to disable them from being generated - this is useful if you have made custom changes to them. Files in orange have been automatically detected as changed.

Configuration Package Save Path

Specify the desired directory for the MoveIt configuration package to be generated. Overwriting an existing configuration package directory is acceptable. Example:
`/u/robot/ros/pr2_moveit_config`

Files to be generated: (checked)

- | | |
|--|-----------------------|
| <input checked="" type="checkbox"/> package.xml | Defines a ROS package |
| <input checked="" type="checkbox"/> CMakeLists.txt | |
| <input checked="" type="checkbox"/> config/ | |
| <input checked="" type="checkbox"/> config/rosbook_arm.srdf | |
| <input checked="" type="checkbox"/> config/ompl_planning.yaml | |
| <input checked="" type="checkbox"/> config/kinematics.yaml | |
| <input checked="" type="checkbox"/> config/joint_limits.yaml | |
| <input checked="" type="checkbox"/> config/fake_controllers.yaml | |
| <input checked="" type="checkbox"/> launch/ | |
| <input checked="" type="checkbox"/> launch/move_group.launch | |
| <input checked="" type="checkbox"/> launch/planning_context.launch | |
| <input checked="" type="checkbox"/> launch/moveit_rviz.launch | |
| <input checked="" type="checkbox"/> launch/ompl_planning_pipeline.launch.xml | |
| <input checked="" type="checkbox"/> launch/planning_pipeline.launch.xml | |
| <input checked="" type="checkbox"/> launch/warehouse_settings.launch.xml | |
| <input checked="" type="checkbox"/> launch/warehouse.launch | |
| <input checked="" type="checkbox"/> launch/default_warehouse_db.launch | |



Generate Configuration Files tab of MoveIt! Setup Assistant

It is important to take into account that the configuration generated by the Setup Assistant has already been provided in the repository and that even though it is recommended that you go through the process, the result can be discarded in favor of the provided package, which is already being referenced by the rest of the launch scripts and configuration files in the repository.

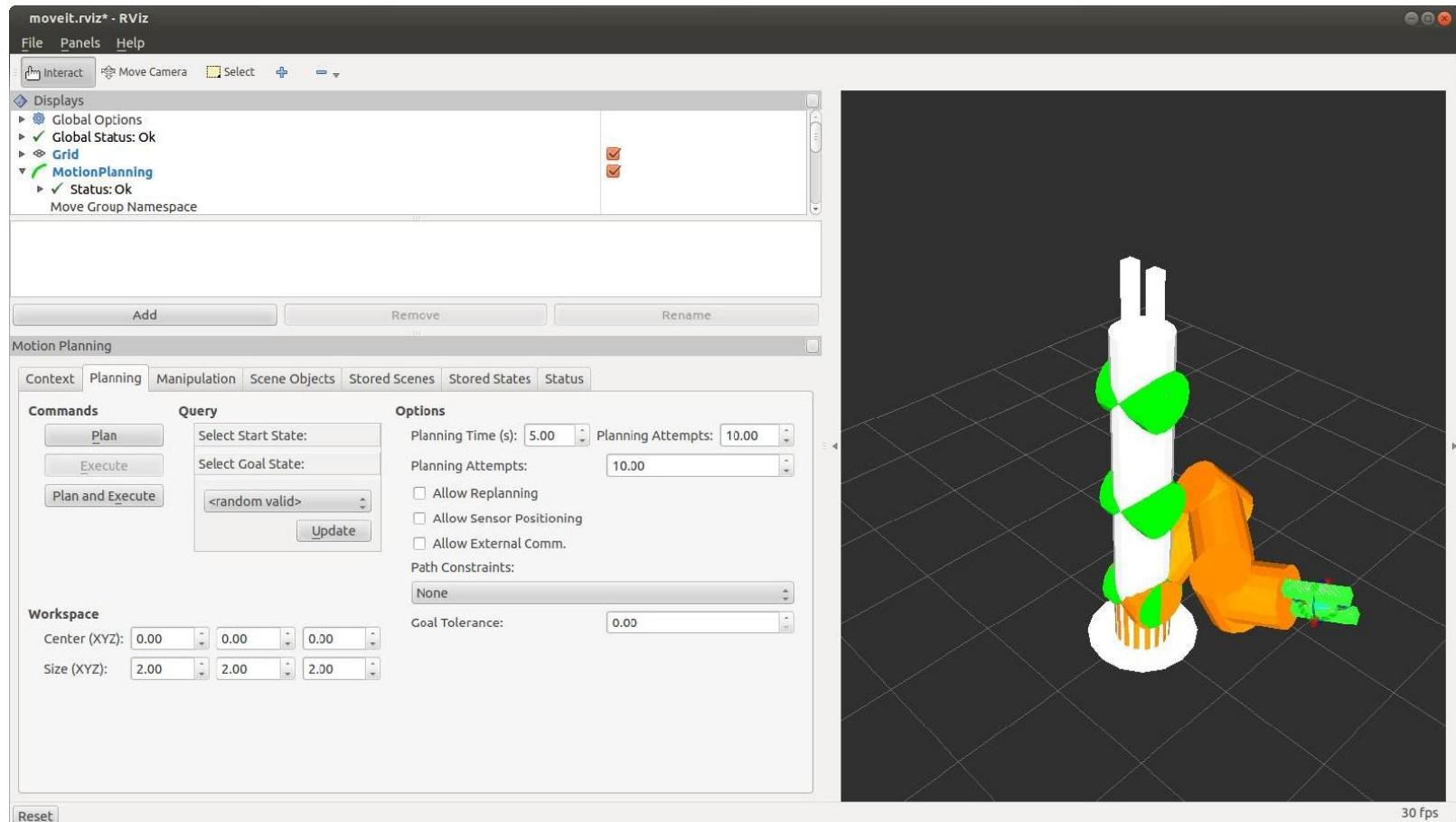
Integration into RViz

MoveIt! provides a very useful and complete RViz plugin that gives the user the ability to perform several actions, such as plan different goals, add and remove objects to the scene, and so on. The Setup Assistant usually creates a number of launch files, among which there is one called `demo`, which takes care of launching MoveIt! as well as the fake controllers, RViz, and the plugin. In order to start the demonstration, run the following command:

```
| $ rosrun rosbook_arm_moveit_config demo.launch
```

Once RViz launches, a motion planning panel should appear as well as the visualization of the robotic arm. The important tabs we need to consider are the Planning tab and the Scene objects tab. In the Planning tab, the user will be able to plan different goal positions, execute them, and set some of the common planning options. In the latter, objects can be inserted and removed from the planning scene.

The following figure shows the Planning tab as well as a visualization of the robotic arm in both white and orange. The former is the current state of the arm, and the latter is the goal position defined by the user. In this particular case, the goal position has been generated using the tools in the Query panel. Once the user is happy with the goal state, the next step can be to either plan to visualize how the arm is going to move or plan to execute it to not only visualize the movement but also move the arm itself.

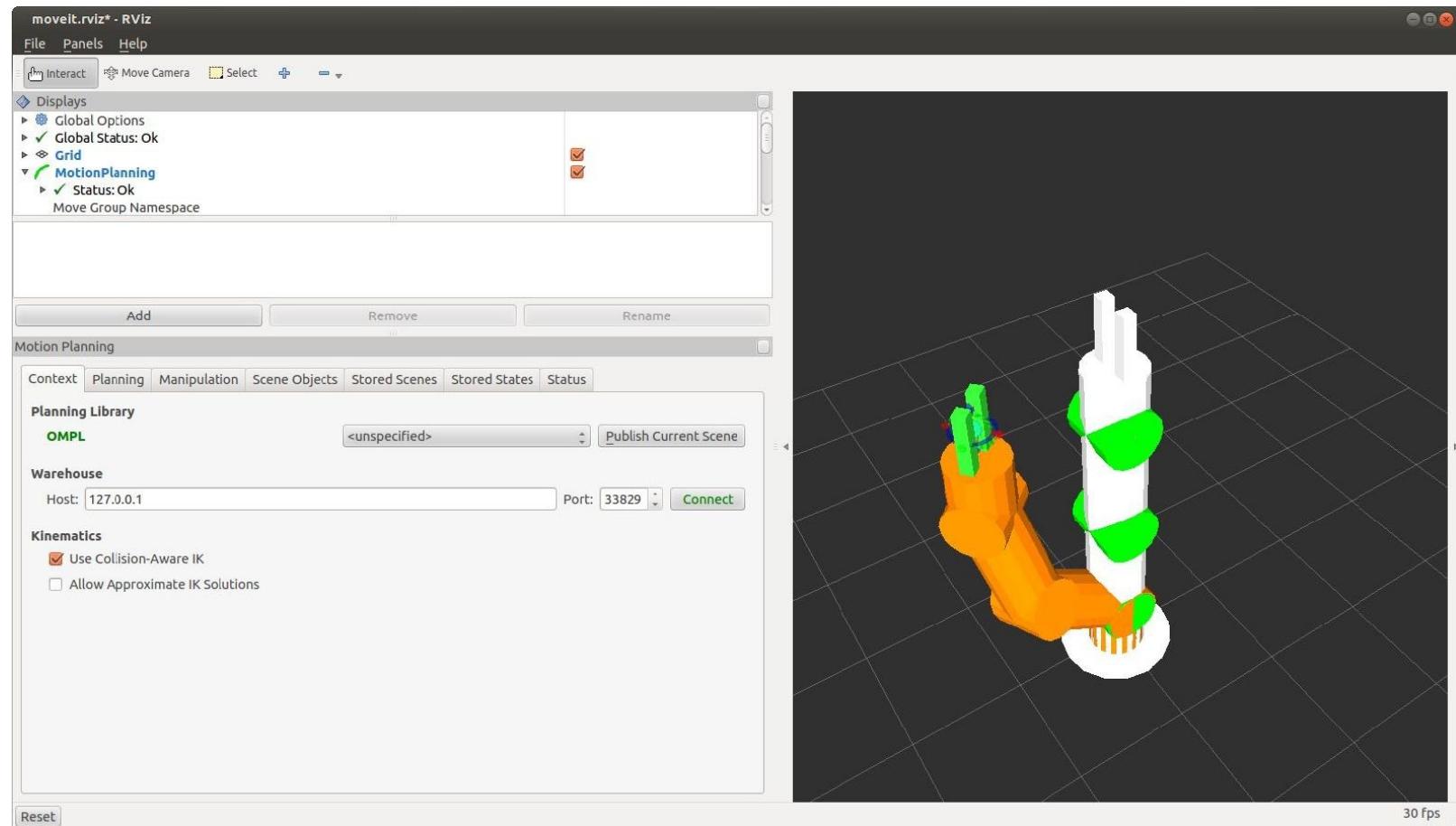


Planning tab and goal position visualization in RViz plugin

Other options, such as the planning time or the number of planning attempts, can be tweaked in order to

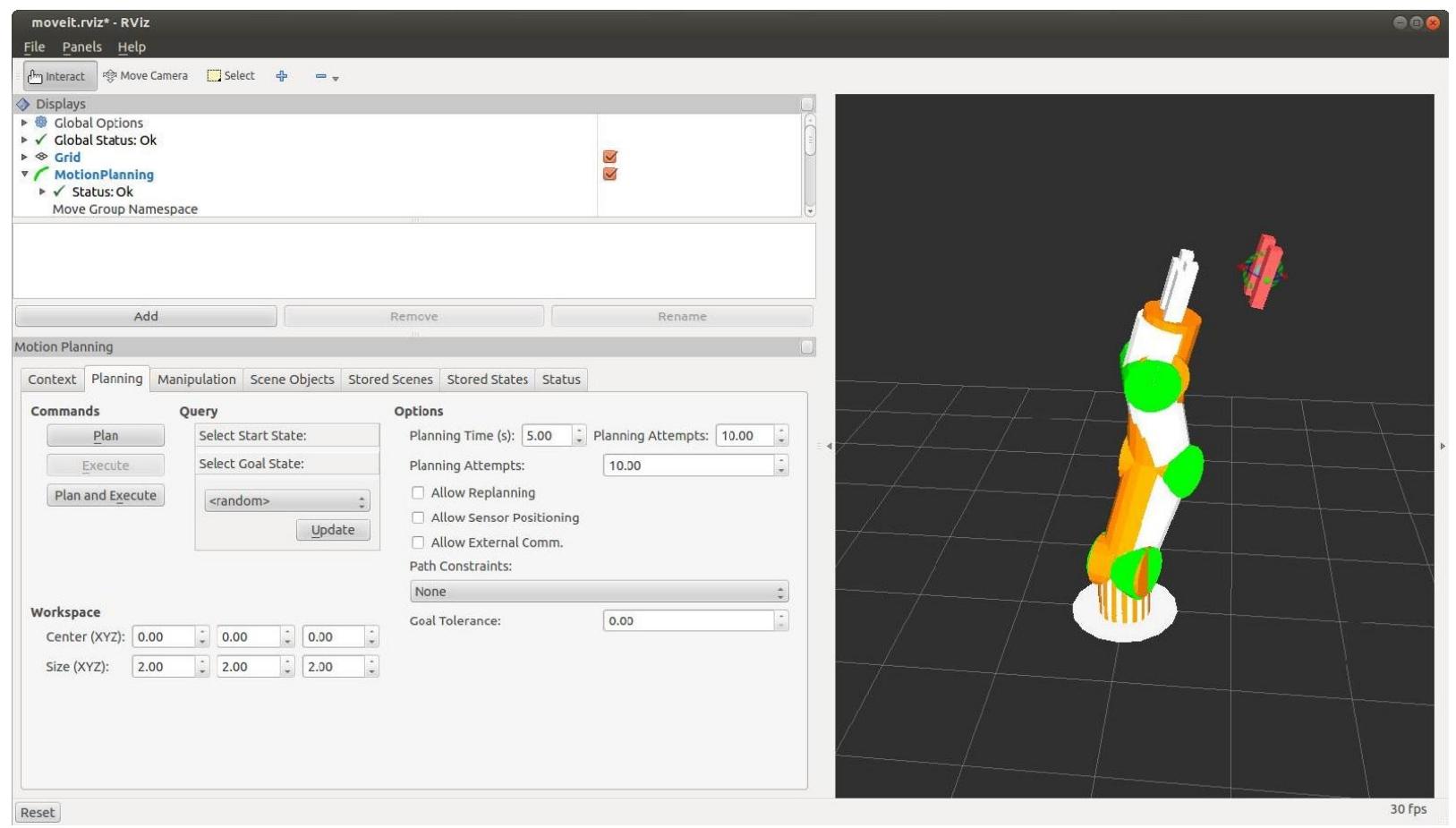
account for complex goals, but for most of the cases in the demonstration, changing these parameters won't be required. Another important parameter is the goal tolerance, which defines how close to the goal position we require the robotic arm to be in order to consider the position as having been achieved.

Planning random goals might be of some interest, but another level of planning is provided by the RViz plugin. As illustrated in the following figure, the robotic arm visualization has a marker on the end effector. This marker allows us to position the end effector of the arm as well as rotate it on each axis. You can now make use of this marker to position the arm towards more interesting configurations.



Using markers to set the goal position in RViz plugin

In many cases, planning by positioning the marker might produce no movement at all and show the robotic arm in the same position, but the marker and the end effector might be in other positions. An example of this behavior can be seen in the following figure, and it usually happens when the desired position is out of the range of motion of the robotic arm (when there are not enough degrees of freedom, too many constraints, and so on).



Markers out of bounds in RViz plugin

Similarly, when the arm is positioned in a state in which it collides with elements in the scene or with itself, the arm will show the collision zone in red. Finally, the following screenshot shows the different options provided by the MoveIt! plugin's visualization:



Motion planning plugin options in RViz plugin

As the names suggest, all of these options are meant to provide a way to tweak the visualization as well as add more information to it. Other interesting options that the user might want to modify are Trajectory Topic, which is the topic on which the visualization trajectory is published, and Query Start State, which will also show the state from which the arm is about to execute the plan. In most cases, the start state is usually the current state of the arm, but having a visualization cue can help spot issues in our algorithms.

Integration into Gazebo or a real robotic arm

The MoveIt! integration into Gazebo is a relatively straightforward process, which can be divided into two different steps: first of all, we need to provide all of the sensors required by MoveIt!, such as the RGB-D sensor, so that motion planning can take the environment into account, and secondly, we also need to provide a controller as well as the current joint states periodically.

When a sensor is created in Gazebo, it interacts with the system as a normal sensor would, by simply producing the required data. This data is then used by MoveIt! in exactly the same way that data produced by a real sensor would in order to generate collision artifacts in the planning scene. The process of making MoveIt! aware of those sensors will be explained later in this chapter.

As regards the manipulator's (arm and gripper) definition, a URDF description is provided using Xacro files as with any robot in ROS. In MoveIt!, we need to configure the controllers for the manipulator joints as `JointTrajectoryController` because the motion plans provide the output with messages for that type of controller. For the manipulator used in this chapter, we need two controllers of this type: one for the arm and another for the gripper. The controller configuration is organized in the `rosbook_arm_controller_configuration` and `rosbook_arm_controller_configuration_gazebo` packages with the `launch` and `config` YAML files, respectively.

This type of controller is provided by the ROS control. Consequently, we need a `RobotHardware` interface for our arm to actually move in Gazebo or in the real hardware. The implementation for Gazebo and the real arm is different, and here we only provide the first. The `rosbook_arm_hardware_gazebo` package has the C++ implementation of `RobotHardware` for the manipulator used in this chapter. This is done by implementing the interface, so we create a new class that inherits from it.

Then, the joints are properly handled by writing the desired target positions (using position control) and reading the actual ones, along with the effort and velocity for each joint. For the sake of simplicity, we omit the explanation of the details of this implementation, which is not needed to understand MoveIt! However, if the number manipulator is drastically changed, the implementation must be changed although it is generic enough to detect the number of joints automatically from the robot description.

Simple motion planning

The RViz plugin provides a very interesting mechanism to interact with MoveIt! but it could be considered quite limited or even cumbersome due to the lack of automation. In order to fully make use of the capabilities included in MoveIt!, several APIs have been developed, which allow us to perform a range of operations over it, such as motion planning, accessing the model of our robot, and modifying the planning scene.

In the following section, we will cover a few examples on how to perform different sorts of simple motion planning. We will start by planning a single goal, continue with planning a random target, proceed with planning a predefined group state, and finally, explain how to improve the interaction of our snippets with RViz.

In order to simplify the explanations, a set of launch files have been provided to launch everything required. The most important one takes care of launching Gazebo, MoveIt!, and the arm controllers:

```
| $ rosrun rosbook_arm_gazebo rosbook_arm_empty_world.launch
```

Another interesting launch file has been provided by the Setup Assistant, which launches RViz and the motion planning plugin. This particular one is optional, but it is useful to have, as RViz will be used further in this section:

```
| $ rosrun rosbook_arm_moveit_config moveit_rviz.launch config:=true
```

A number of snippets have also been provided, which cover everything explained in this section. The snippets can be found in the `rosbook_arm_snippets` package. The snippets package doesn't contain anything other than code, and launching the snippets will be done by calling `rosrun` instead of the usual `roslaunch`.

Every snippet of code in this section follows the same pattern, starting with the typical ROS initialization, which won't be covered here. After the initialization, we need to define the planning group on which motion planning is going to be performed. In our case, we only have two planning groups, the arm and the gripper, but in this case, we only care about the arm. This will instantiate a planning group interface, which will take care of the interaction with MoveIt!:

```
| moveit::planning_interface::MoveGroup plan_group("arm");
```

After the instantiation of the planning group interface, there is usually some code dedicated to deciding on the goal that will be specific to each of the types of goals covered in this section. After the goal has been decided, it needs to be conveyed to MoveIt! so that it gets executed. The following snippet of code takes care of creating a plan and using the planning group interface to request MoveIt! to perform motion planning and, if successful, also execute it:

```
| moveit::planning_interface::MoveGroup::Plan goal_plan;
| if (plan_group.plan(goal_plan))
| {
|   .plan_group.move();
| }
```

Planning a single goal

To plan a single goal, we literally only need to provide MoveIt! with the goal itself. A goal is expressed by a `Pose` message from the `geometry_msgs` package. We need to specify both the orientation and the pose. For this particular example, this goal was obtained by performing manual planning and checking the state of the arm. In a real situation, goals will probably be set depending on the purpose of the robotic arm:

```
geometry_msgs::Pose goal;
goal.orientation.x = -0.000764819;
goal.orientation.y = 0.0366097;
goal.orientation.z = 0.00918912;
goal.orientation.w = 0.999287;
goal.position.x = 0.775884;
goal.position.y = 0.43172;
goal.position.z = 2.71809;
```

For this particular goal, we can also set the tolerance. We are aware that our PID is not incredibly accurate, which could lead to MoveIt! believing that the goal hasn't been achieved. Changing the goal tolerance makes the system achieve the waypoints with a higher margin of error in order to account for inaccuracies in the control:

```
| plan_group.setGoalTolerance(0.2);
```

Finally, we just need to set the planning group target pose, which will then be planned and executed by the snippet of code shown at the beginning of this section:

```
| plan_group.setPoseTarget(goal);
```

We can run this snippet of code with the following command; the arm should position itself without any issues:

```
| $ rosrun rosbook_arm_snippets move_group_plan_single_target
```

Planning a random target

Planning a random target can be effectively performed in two steps: first of all, we need to create the random target itself and then check its validity. If the validity is confirmed, then we can proceed by requesting the goal as usual; otherwise, we will cancel (although we could retry until we find a valid random target). In order to verify the validity of the target, we need to perform a service call to a service provided by MoveIt! for this specific purpose. As usual, to perform a service call, we need a service client:

```
ros::ServiceClient validity_srv =  
nh.serviceClient<moveit_msgs::GetStateValidity>("/check_state_vali  
dity");
```

Once the service client is set up, we need to create the random target. To do so, we need to create a robot state object containing the random positions, but to simplify the process, we can start by acquiring the current robot state object:

```
robot_state::RobotState current_state =  
*plan_group.getCurrentState();
```

We will then set the current robot state object to random positions, but to do so, we need to provide the joint model group for this robot state. The joint model group can be obtained using the already created robot state object as follows:

```
current_state.setToRandomPositions(current_state.getJointModelGrou  
p("arm"));
```

Up to this point, we have a service client waiting to be used as well as a random robot state object, which we want to validate. We will create a pair of messages: one for the request and another for the response. Fill in the request message with the random robot state using one of the API conversion functions, and request the service call:

```
moveit_msgs::GetStateValidity::Request validity_request;  
moveit_msgs::GetStateValidity::Response validity_response;  
  
robot_state::robotStateToRobotStateMsg(current_state,  
validity_request.robot_state);  
validity_request.group_name = "arm";  
  
validity_srv.call(validity_request, validity_response);
```

Once the service call is complete, we can check the response message. If the state appears to be invalid, we would simply stop running the module; otherwise, we will continue. As explained earlier, at this point, we could retry until we get a valid random state; this can be an easy exercise for the reader:

```
if (!validity_response.valid)  
{  
ROS_INFO("Random state is not valid");  
ros::shutdown();  
return 1;  
}
```

Finally, we will set the robot state we just created as the goal using the planning group interface, which

will then be planned and executed as usual by MoveIt!:

```
| plan_group.setJointValueTarget(current_state);
```

We can run this snippet of code with the following command, which should lead to the arm repositioning itself on a random configuration:

```
| $ rosrun rosbook_arm_snippets move_group_plan_random_target
```

Planning a predefined group state

As we commented during the configuration generation step, when initially integrating our robotic arm, MoveIt! provides the concept of predefined group states, which can later be used to position the robot with a predefined pose. Accessing predefined group states requires creating a robot state object as a target; in order to do so, the best approach is to start by obtaining the current state of the robotic arm from the planning group interface:

```
| robot_state::RobotState current_state =  
| *plan_group.getCurrentState();
```

Once we have obtained the current state, we can modify it by setting it to the predefined group state, with the following call, which takes the model group that needs to be modified and the name of the predefined group state:

```
| current_state.setToDefaultValues(current_state.getJointModelGroup(  
| "arm"), "home");
```

Finally, we will use the new robot state of the robotic arm as our new goal and let MoveIt! take care of planning and execution as usual:

```
| plan_group.setJointValueTarget(current_state);
```

We can run this snippet of code with the following command, which should lead to the arm repositioning itself to achieve the predefined group state:

```
| $ rosrun rosbook_arm_snippets move_group_plan_group_state
```

Displaying the target motion

MoveIt! provides a set of messages that can be used to communicate visualization information, essentially providing it with the planned path in order to get a nice visualization of how the arm is going to move to achieve its goal. As usual, communication is performed through a topic, which needs to be advertised:

```
ros::Publisher display_pub =
nh.advertise<moveit_msgs::DisplayTrajectory>("/move_group/display_
planned_path", 1, true);
```

The message we need to publish requires the start state of the trajectory and the trajectory itself. In order to obtain such information, we always need to perform planning using the planning group interface first, and using the created plan, we can proceed to fill in the message:

```
moveit_msgs::DisplayTrajectory display_msg;
display_msg.trajectory_start = goal_plan.start_state_;
display_msg.trajectory.push_back(goal_plan.trajectory_);
display_pub.publish(display_msg);
```

Once the message has been filled in, publishing it to the correct topic will cause the RViz visualization to show the trajectory that the arm is about to perform. It is important to take into account that, when performing a call to plan, it will also show the same type of visualization, so you shouldn't be confused if the trajectory is displayed twice.

Motion planning with collisions

It might be interesting for the reader to know that MoveIt! provides motion planning with collisions out of the box, so in this section we will cover how you can add elements to the planning scene that could potentially collide with our robotic arm. First, we will start by explaining how to add basic objects to the planning scene, which is quite interesting as it allows us to perform planning even if a real object doesn't exist in our scene. For completion, we will also explain how to remove those objects from the scene. Finally, we will explain how to add an RGBD sensor feed, which will produce point clouds based on real-life (or simulated) objects, thus making our motion planning much more interesting and realistic.

Adding objects to the planning scene

To start adding an object, we need to have a planning scene; this is only possible when MoveIt! is running, so the first step is to start Gazebo, MoveIt!, the controllers, and RViz. Since the planning scene only exists in MoveIt!, RViz is required to be able to visualize objects contained in it. In order to launch all of the required modules, we need to run the following commands:

```
| $ roslaunch rosbook_arm_gazebo rosbook_arm_empty_world.launch  
| $ roslaunch rosbook_arm_moveit_config moveit_rviz.launch config:=true
```

The snippet of code then starts by instantiating the planning scene interface object, which can be used to perform actions on the planning scene itself:

```
| moveit::planning_interface::PlanningSceneInterface current_scene;
```

The next step is to create the collision object message that we want to send through the planning scene interface. The first thing we need to provide for the collision object is a name, which will uniquely identify this object and will allow us to perform actions on it, such as removing it from the scene once we're done with it:

```
| moveit_msgs::CollisionObject box;  
| box.id = "rosbook_box";
```

The next step is to provide the properties of the object itself. This is done through a solid primitive message, which specifies the type of object we are creating, and depending on the type of object, it also specifies its properties. In our case, we are simply creating a box, which essentially has three dimensions:

```
| shape_msgs::SolidPrimitive primitive;  
| primitive.type = primitive.BOX;  
| primitive.dimensions.resize(3);  
| primitive.dimensions[0] = 0.2;  
| primitive.dimensions[1] = 0.2;  
| primitive.dimensions[2] = 0.2;
```

To continue, we need to provide the pose of the box in the planning scene. Since we want to produce a possible collision scenario, we have placed the box close to our robotic arm. The pose itself is specified with a pose message from the standard geometry messages package:

```
| geometry_msgs::Pose pose;  
| pose.orientation.w = 1.0;  
| pose.position.x = 0.7;  
| pose.position.y = -0.5;  
| pose.position.z = 1.0;
```

We then add the primitive and the pose to the message and specify that the operation we want to perform is to add it to the planning scene:

```
| box.primitives.push_back(primitive);  
| box.primitive_poses.push_back(pose);  
| box.operation = box.ADD;
```

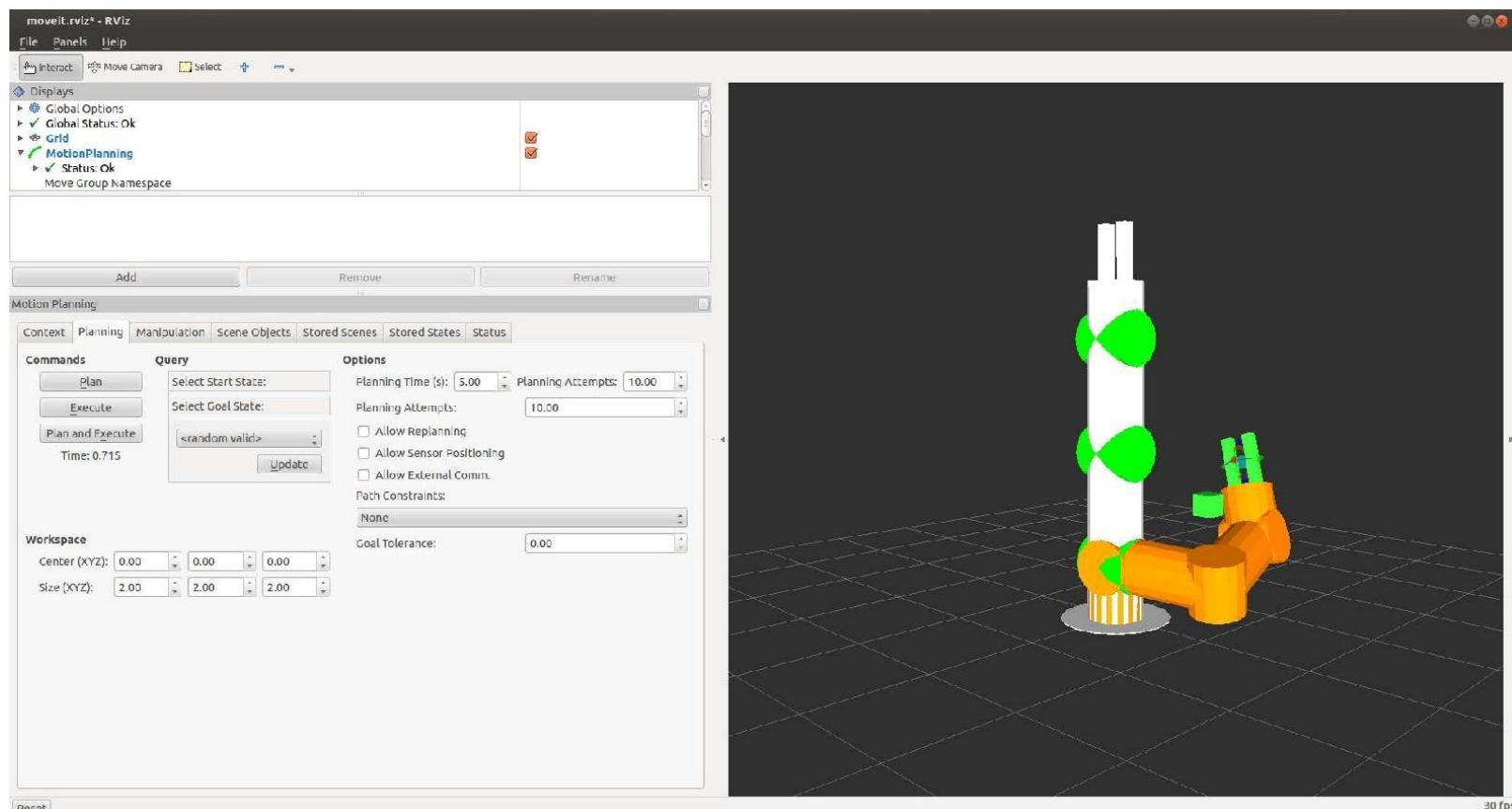
Finally, we add the collision object to a vector of collision object messages and call the `addCollisionObjects` method from the planning scene interface. This will take care of sending the required messages through the appropriate topics, in order to ensure that the object is created in the current planning scene:

```
std::vector<moveit_msgs::CollisionObject> collision_objects;
collision_objects.push_back(box);
current_scene.addCollisionObjects(collision_objects);
```

We can test this snippet by running the following command in a terminal, as said earlier. Since the object is added to the planning scene, it is important to have the RViz visualization running; otherwise, the reader won't be able to see the object:

```
| $ rosrun rosbook_arm_snippets move_group_add_object
```

The result can be seen in the following figure as a simple, green, squared box in between the arm's goal and the current position of the arm:



Scene collision object in RViz

Removing objects from the planning scene

Removing the added object from the planning scene is a very simple process. Following the same initialization as in the previous example, we only need to create a string vector containing the IDs of the objects we want to remove and call the `removeCollisionObjects` function from the planning scene interface:

```
| std::vector<std::string> object_ids;  
| object_ids.push_back("rosbook_box");  
| current_scene.removeCollisionObjects(object_ids);
```

We can test this snippet by running the following command, which will remove the object created with the previous snippet from the planning scene:

```
| $ rosrun rosbook_arm_snippets move_group_remove_object
```

Alternatively, we can also use the Scene objects tab in the RViz plugin to remove any objects from the scene.

Motion planning with point clouds

Motion planning with point clouds is much simpler than it would appear to be. The main thing to take into account is that we need to provide a point cloud feed as well as tell MoveIt! to take this into account when performing planning. The Gazebo simulation we have set up for this chapter already contains an RGBD sensor, which publishes a point cloud for us. To start with this example, let's launch the following commands:

```
| $ roslaunch rosbook_arm_gazebo rosbook_arm_grasping_world.launch  
| $ roslaunch rosbook_arm_moveit_config moveit_rviz.launch config:=true
```

The user might have noticed that the Gazebo simulation now appears to include several objects in the world. Those objects are scanned by an RGBD sensor, and the resulting point cloud is published to the `/rgbd_camera/depth/points` topic. What we need to do in this case is tell MoveIt! where to get the information from and what the format of that information is. The first file we need to modify is the following one:

```
| rosbook_arm_moveit_config/config/sensors_rgbd.yaml
```

This file will be used to store the information of the RGBD sensor. In this file, we need to tell MoveIt! which plugin it needs to use to manage the point cloud as well as some other parameters specific to the sensor plugin itself. In this particular case, the plugin to use is **Octomap Updater**, which will generate an octomap with the point cloud provided, downsample it, and publish the resulting cloud. With this first step we have set up a plugin which will provide MoveIt! with enough information to plan, while also taking into account possible collisions with the point cloud:

```
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
```

As you might have suspected, the file itself is nothing more than a configuration file. The next step we need to perform is to load this configuration file into the environment so that MoveIt! is aware of the new sensor we have added. In order to do so, we will need to modify the following XML file:

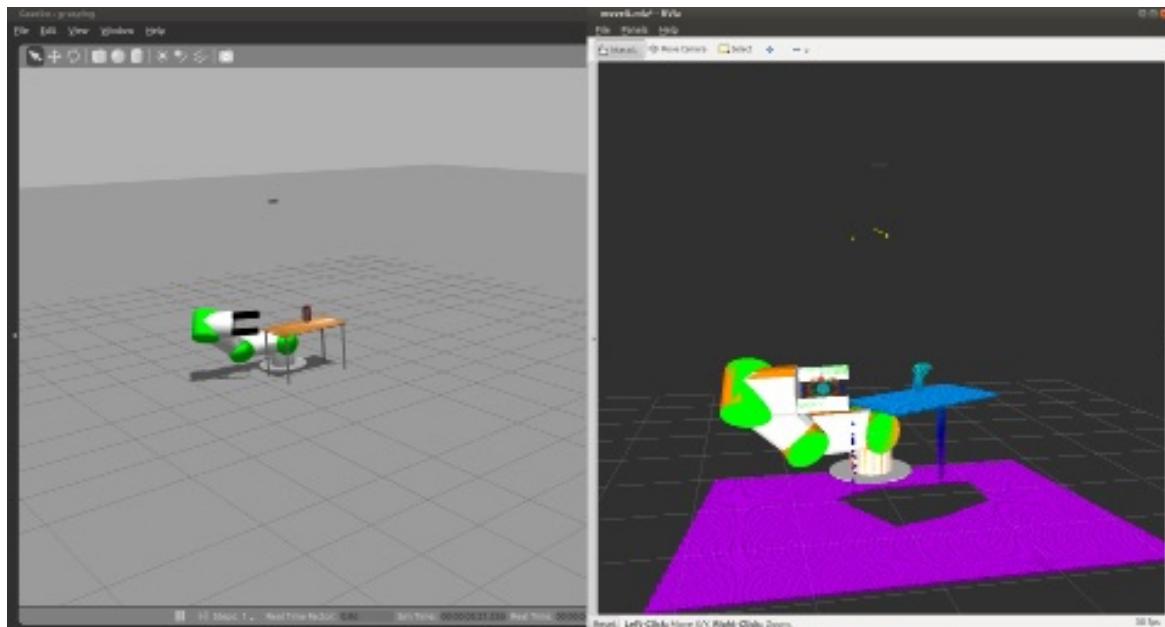
```
| $ rosbook_arm_moveit_config/launch/ rosbook_arm_moveit_sensor_manager.launch.xml
```

In this XML file, we can potentially specify a few parameters that will be used by the sensor plugin, such as the cloud resolution and the frame of reference. It is important to take into account that some of these parameters might be redundant and can be omitted. Finally, we need to add a command to load the configuration file into the environment:

```
&lt;launch>  
  &lt;rosparam command="load" file="$(find  
    rosbook_arm_moveit_config)/config/sensors_rgbd.yaml" />  
&lt;/launch>
```

The result of running the commands specified in the beginning with the new changes we have added can

be seen in the following screenshot. In this particular case, we can see both the Gazebo simulation and the RViz visualization. The RViz simulation contains a point cloud, and we have already performed some manual motion planning, which successfully took the point cloud into account to avoid any collisions:



Gazebo simulation (left), point cloud in RViz (right)

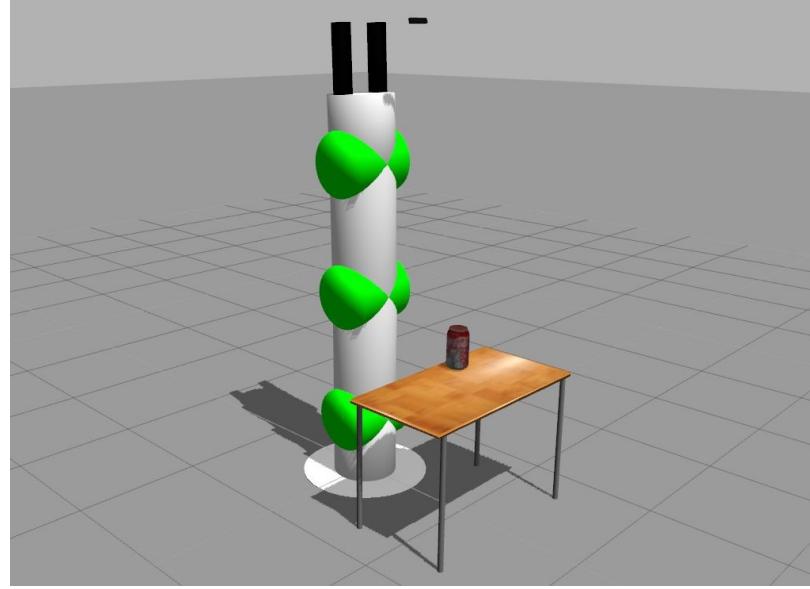
The pick and place task

In this section, we are going to explain how to perform a very common application or task with a manipulator robot. A pick and place task consists of picking up a target object, which includes grasping it, and placing it somewhere else. Here, we assume that the object is initially on top of a supporting surface that is flat or planar, such as a table, but it is easy to generalize it to more complex environments. As regards the object to grasp, we will consider a cylinder that is approximated by a box, because the gripper we are going to use to grasp is very simple; for more complex objects, you will need a better gripper or even a hand.

In the further sections, we will start by describing how to set up the planning scene that MoveIt! needs in order to identify the objects that are there, apart from the arm itself. These objects are considered during motion planning to avoid obstacles, and they can also be targets for picking up or grasping. In order to simplify the problem, we will omit the perception part, but we will explain how it can be done and integrated. Once the planning scene is defined, we will describe how to perform the pick and place task using the MoveIt! API. Finally, we will explain how to run this task in demonstration mode using fake controllers so that we do not need the actual robot (either simulated on Gazebo or a real one). We will also show how you can actually see the motion on the simulated arm in Gazebo while it is interacting with the simulated objects in the environment.

The planning scene

The first thing we have to do is define the objects in the environment since MoveIt! needs this information to make the arm interact with an object without colliding with it, and to reference them to do certain actions. Here, we will consider the following scene:



Environment with manipulator and objects in Gazebo

This scene has the arm with the gripper and the **RGB-D** sensor as the robotic manipulator. There is also a table and a can of Coke; the flat support surface and the cylindrical object, respectively. You can run this scene in Gazebo with the following command:

```
| $ roslaunch rosbook_arm_gazebo rosbook_arm_grasping_world.launch
```

This scene is just a simple example that models a real use case. However, we still have to tell MoveIt! about the planning scene. At this moment, it only knows about the robotic manipulator. We have to tell it about the table and the can of Coke. This can be done either by using 3D perception algorithms, which take the point cloud of the RGB-D sensor, or programmatically, by specifying the pose and shape of the objects with some basic primitives. We will see how we can define the planning scene following the latter approach.

The code to perform the pick and place task is the `pick_and_place.py` Python program located in the scripts folder of the `rosbook_arm_pick_and_place` package. The important part to create the planning scene is in the `__init__` method of the `CokeCanPickAndPlace` class:

```
| self._scene = PlanningSceneInterface()
```

In the following sections, we will add the table and the can of Coke to this planning scene.

The target object to grasp

In this case, the target object to grasp is the can of Coke. It is a cylindrical object that we can approximate as a box, which is one of the basic primitives in the MoveIt! planning scene API:

```
# Retrieve params:  
self._grasp_object_name = rospy.get_param('~grasp_object_name',  
    'coke_can')  
  
# Clean the scene:  
self._scene.remove_world_object(self._grasp_object_name)  
  
# Add table and Coke can objects to the planning scene:  
self._pose_coke_can = self._add_coke_can(self._grasp_object_name)
```

The objects in the planning scene receive a unique identifier, which is a string. In this case, `coke_can` is the identifier for the can of Coke. We remove it from the scene to avoid having duplicate objects, and then we add to the scene. The `_add_coke_can` method does that by defining the `pose` and `shape` dimensions:

```
def _add_coke_can(self, name):  
    p = PoseStamped()  
    p.header.frame_id = self._robot.get_planning_frame()  
    p.header.stamp = rospy.Time.now()  
  
    p.pose.position.x = 0.75 - 0.01  
    p.pose.position.y = 0.25 - 0.01  
    p.pose.position.z = 1.00 + (0.3 + 0.03) / 2.0  
  
    q = quaternion_from_euler(0.0, 0.0, 0.0)  
    p.pose.orientation = Quaternion(*q)  
  
    self._scene.add_box(name, p, (0.15, 0.15, 0.3))  
  
    return p.pose
```

The important part here is the `add_box` method that adds a `box` object to the planning scene we created earlier. The box is given a name, its pose, and dimensions, which in this case are set to match the ones in the Gazebo world shown earlier, with the table and the can of Coke. We also have to set `frame_id` to the planning frame ID and the timestamp to `now`. In order to use the planning frame, we need `RobotCommander`, which is the MoveIt! interface to command the manipulator programmatically:

```
| self._robot = RobotCommander()
```

The support surface

We proceed similarly to create the object for the table, which is also approximated by a box. Therefore, we simply remove any previous object and add the table. In this case, the object name is `table`:

```
# Retrieve params:  
self._table_object_name = rospy.get_param('~table_object_name',  
    'table')  
  
# Clean the scene:  
self._scene.remove_world_object(self._table_object_name)  
  
# Add table and Coke can objects to the planning scene:  
self._pose_table = self._add_table(self._table_object_name)
```

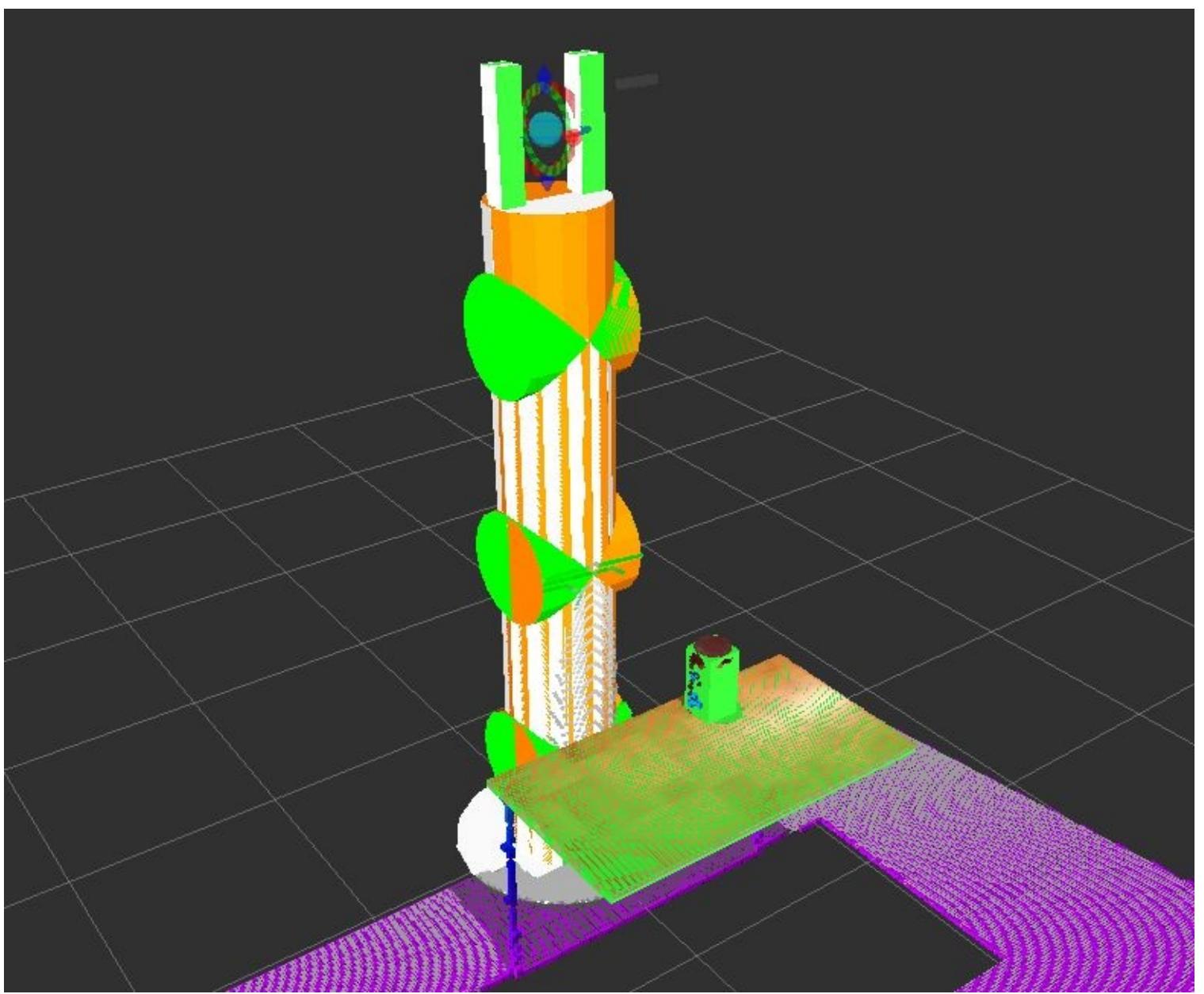
The `_add_table` method adds the table to the planning scene:

```
def _add_table(self, name):  
    p = PoseStamped()  
    p.header.frame_id = self._robot.get_planning_frame()  
    p.header.stamp = rospy.Time.now()  
  
    p.pose.position.x = 1.0  
    p.pose.position.y = 0.0  
    p.pose.position.z = 1.0  
  
    q = quaternion_from_euler(0.0, 0.0, numpy.deg2rad(90.0))  
    p.pose.orientation = Quaternion(*q)  
  
    self._scene.add_box(name, p, (1.5, 0.8, 0.03))  
  
    return p
```

We can visualize the planning scene objects in RViz running the following commands:

```
$ roslaunch rosbook_arm_gazebo rosbook_arm_grasping_world.launch  
$ roslaunch rosbook_arm_moveit_config moveit_rviz.launch config:=true  
$ roslaunch rosbook_arm_pick_and_place grasp_generator_server.launch  
$ rosrun rosbook_arm_pick_and_place pick_and_place.py
```

This actually runs the whole pick and place task, which we will continue to explain later. Right after starting the `pick_and_place.py` program, you will see the boxes that model the table and the can of Coke in green, matching perfectly with the point cloud seen by the RGB-D sensor, as shown in the following figure:



Point cloud seen by the RGB-D sensor of the environment

Perception

Adding the objects manually to the planning scenes can be avoided by perceiving the supporting surface. In this case, the table can be detected as a horizontal plane on the point cloud. Once the table is recognized, it can be subtracted from the original point cloud to obtain the target object, which can be approximated with a cylinder or a box. We will use the same method to add boxes to the planning scene as before, but in this case, the pose and dimensions (and the classification) of the objects will come from the output of the 3D perception and segmentation algorithm used.

This sort of perception and segmentation in the point cloud provided by the RGB-D sensor can be easily done using the concepts and algorithms. However, in some cases the accuracy will not be enough to grasp the object properly. The perception can be helped using fiducial markers placed on the object to grasp, such as **ArUco** (<http://www.uco.es/investiga/grupos/ava/node/26>) which has the ROS wrapper which can be found at https://github.com/pal-robotics/aruco_ros.

Here, we set the planning scene manually and leave the perception part to you. As we saw, the target object to grasp and the support surface is defined in the code manually, by comparing the correspondence with the point cloud in RViz until we have a good match.

Grasping

Now that we have the target object defined in the scene, we need to generate grasping poses to pick it up. To achieve this aim, we use the grasp generator server from the `moveit_simple_grasps` package, which can be found at https://github.com/davetcoleman/moveit_simple_grasps.

Unfortunately, there isn't a debian package available in Ubuntu for ROS Kinetic. Therefore, we need to run the following commands to add the `kinetic-devel` branch to our workspace (inside the `src` folder of the workspace):

```
$ wstool set moveit_simple_grasps --git https://github.com/davetcoleman/moveit_simple_grasps.git -v kinetic-devel  
$ wstool up moveit_simple_grasps
```

We can build this using the following commands:

```
$ cd ..  
$ caktin_make
```

Now we can run the grasp generator server as follows (remember to source `devel/setup.bash`):

```
$ roslaunch rosbook_arm_pick_and_place grasp_generator_server.launch
```

The grasp generator server needs the following grasp data configuration in our case:

```
base_link: base_link  
  
gripper:  
end_effector_name: gripper  
  
# Default grasp params  
joints: [finger_1_joint, finger_2_joint]  
  
pregrasp_posture: [0.0, 0.0]  
pregrasp_time_from_start: &time_from_start 4.0  
  
grasp_posture: [1.0, 1.0]  
grasp_time_from_start: *time_from_start  
  
postplace_time_from_start: *time_from_start  
  
# 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: tool_link
```

This defines the gripper we are going to use to grasp objects and the pre- and post-grasp postures, basically.

Now we need an action client to query for the grasp poses. This is done inside the `pick_and_place.py` program, right before we try to pick up the target object. So, we create an action client using the following code:

```

# Create grasp generator 'generate' action client:
self._grasps_ac =
    SimpleActionClient('/moveit_simple_grasps_server/generate',
    GenerateGraspsAction)
if not self._grasps_ac.wait_for_server(rospy.Duration(5.0)):
    rospy.logerr('Grasp generator action client not available!')
    rospy.signal_shutdown('Grasp generator action client not
available!')
    return

```

Inside the `_pickup` method, we use the following code to obtain the grasp poses:

```
| grasps = self._generate_grasps(self._pose_coke_can, width)
```

Here, the `width` argument specifies the width of the object to grasp. The `_generate_grasps` method does the following:

```

def _generate_grasps(self, pose, width):
    # Create goal:
    goal = GenerateGraspsGoal()

    goal.pose = pose
    goal.width = width

    # Send goal and wait for result:
    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

```

To summarize, it sends an `actionlib` goal to obtain a set of grasping poses for the target goal pose (usually at the object centroid). In the code provided with the section, there are some options commented upon, but they can be enabled to query only for particular types of grasps, such as some angles, or pointing up or down. The outputs of the function are all the grasping poses that the pickup action will try later. Having multiple grasping poses increases the possibility of a successful grasp.

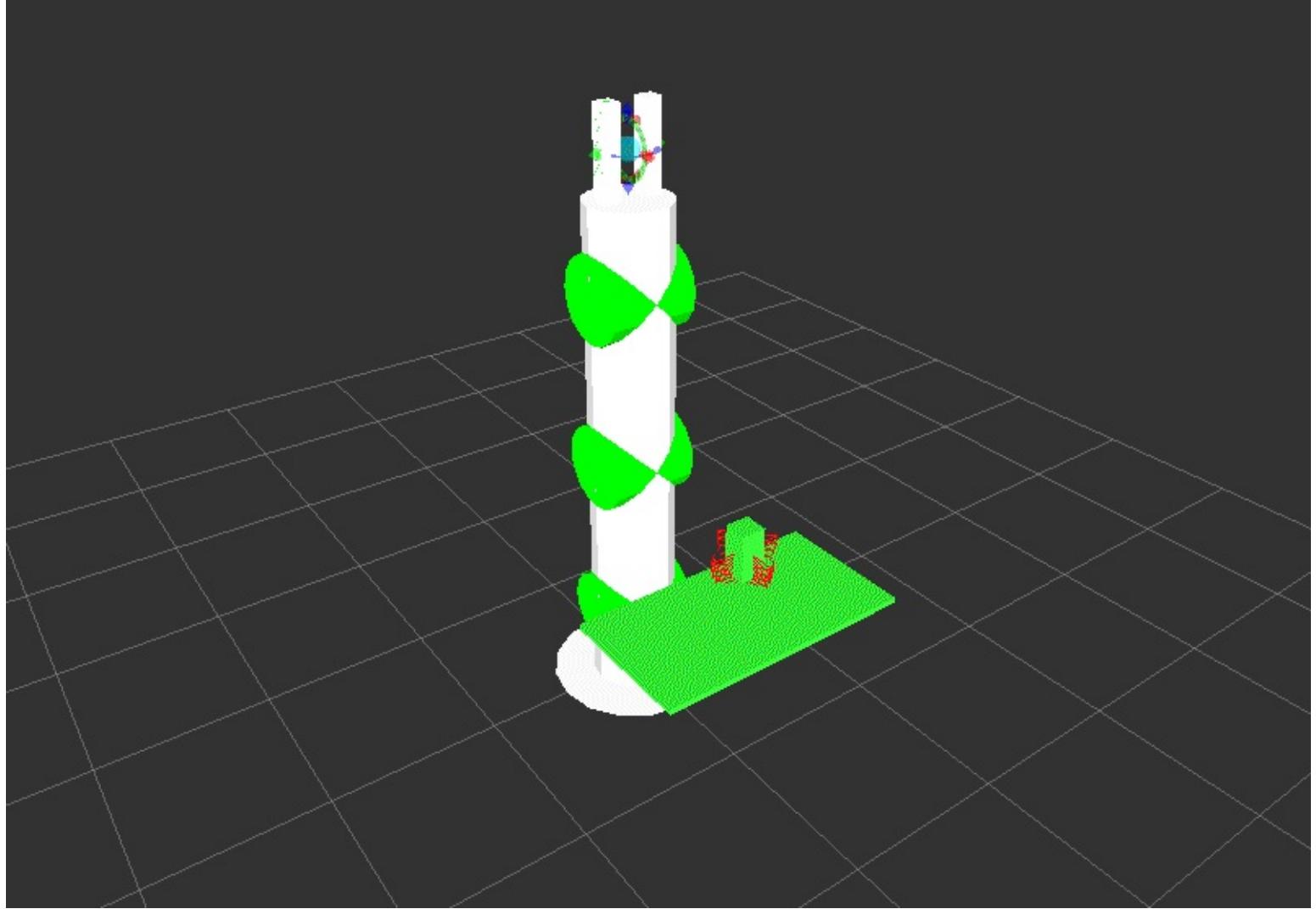
The grasp poses provided by the grasp generation server are also published as `PoseArray` using the `_publish_grasps` method for visualization and debugging purposes. We can see them on RViz running the whole pick and place task as before:

```

$ roslaunch rosbook_arm_gazebo rosbook_arm_grasping_world.launch
$ roslaunch rosbook_arm_moveit_config moveit_rviz.launch config:=true
$ roslaunch rosbook_arm_pick_and_place grasp_generator_server.launch
$ rosrun rosbook_arm_pick_and_place pick_and_place.py

```

A few seconds after running the `pick_and_place.py` program, we will see multiple arrows on the target object which correspond to the grasp pose that will be tried in order to pick it up. This is shown in the following figure as follows:



Visualization of grasping poses

The pickup action

Once we have the grasping poses, we can use the MoveIt! /pickup action server to send a goal passing all of them. As before, we will create an action client:

```
# Create move group 'pickup' action client:  
self._pickup_ac = SimpleActionClient('/pickup', PickupAction)  
if not self._pickup_ac.wait_for_server(rospy.Duration(5.0)):  
    rospy.logerr('Pick up action client not available!')  
    rospy.signal_shutdown('Pick up action client not available!')  
    return
```

Then, we will try to pick up the can of Coke as many times as needed until we finally do it:

```
# Pick Coke can object:  
while not self._pickup(self._arm_group, self._grasp_object_name,  
    self._grasp_object_width):  
    rospy.logwarn('Pick up failed! Retrying ...')  
    rospy.sleep(1.0)
```

Inside the `_pickup` method, we create a pickup goal for MoveIt!, right after generating the grasps poses, as explained earlier:

```
# Create and send Pickup goal:  
goal = self._create_pickup_goal(group, target, grasps)  
  
state = self._pickup_ac.send_goal_and_wait(goal)  
if state != GoalStatus.SUCCEEDED:  
    rospy.logerr('Pick up goal failed!: %s' %  
        self._pickup_ac.get_goal_status_text())  
    return None  
  
result = self._pickup_ac.get_result()  
  
# Check for error:  
err = result.error_code.val  
if err != MoveItErrorCodes.SUCCESS:  
    rospy.logwarn('Group %s cannot pick up target %s!: %s' % (group,  
        target, str(moveit_error_dict[err])))  
    return False  
  
return True
```

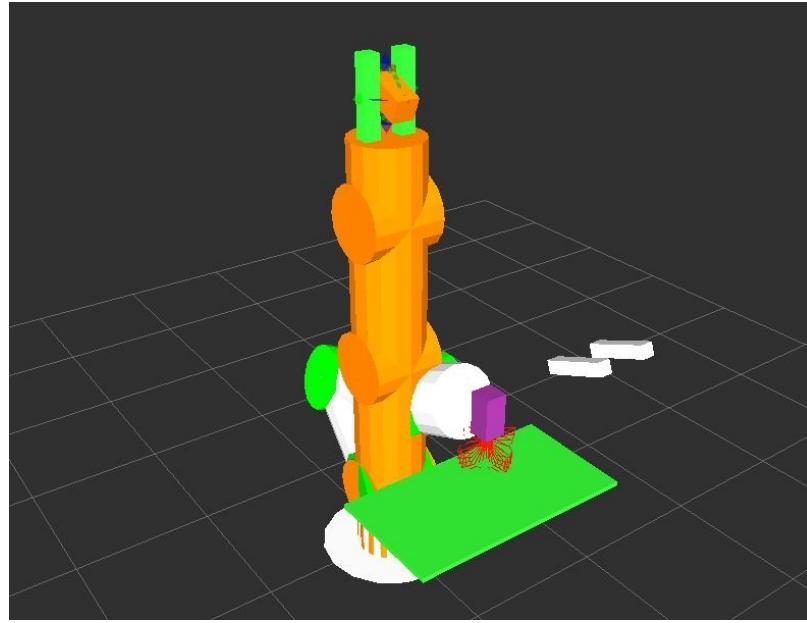
The goal is sent and the state is used to check whether the robot manipulator picks up the object or not. The pickup goal is created in the `_create_pickup_goal` method as follows:

```
def _create_pickup_goal(self, group, target, grasps):  
    # Create goal:  
    goal = PickupGoal()  
  
    goal.group_name = group  
    goal.target_name = target  
  
    goal.possible_grasps.extend(grasps)  
  
    # Configure goal planning options:  
    goal.allowed_planning_time = 5.0  
  
    goal.planning_options.planning_scene_diff.is_diff = True  
    goal.planning_options.planning_scene_diff.robot_state.is_diff =  
        True  
    goal.planning_options.plan_only = False  
    goal.planning_options.replan = True
```

```
goal.planning_options.replan_attempts = 10  
return goal
```

The goal needs the planning group (`arm` in this case) and the target name (`coke_can` in this case). Then, all the possible grasps are set, and several planning options, including the allowed planning time, can be increased if needed.

When the target object is successfully picked up, we will see the box corresponding to it attached to the gripper's grasping frame with a purple color, as shown in the following figure (note that it might appear like a ghost gripper misplaced, but that is only a visualization artifact):



Arm picking up an object

The place action

Right after the object has been picked up, the manipulator will proceed with the place action. MoveIt! provides the `/place` action server, so the first step consists of creating an action client to send a place goal in the desired location, in order to place the object picked up:

```
# Create move group 'place' action client:  
self._place_ac = SimpleActionClient('/place', PlaceAction)  
if not self._place_ac.wait_for_server(rospy.Duration(5.0)):  
    rospy.logerr('Place action client not available!')  
    rospy.signal_shutdown('Place action client not available!')  
    return
```

Then, we will try to place the object until we finally manage to do it:

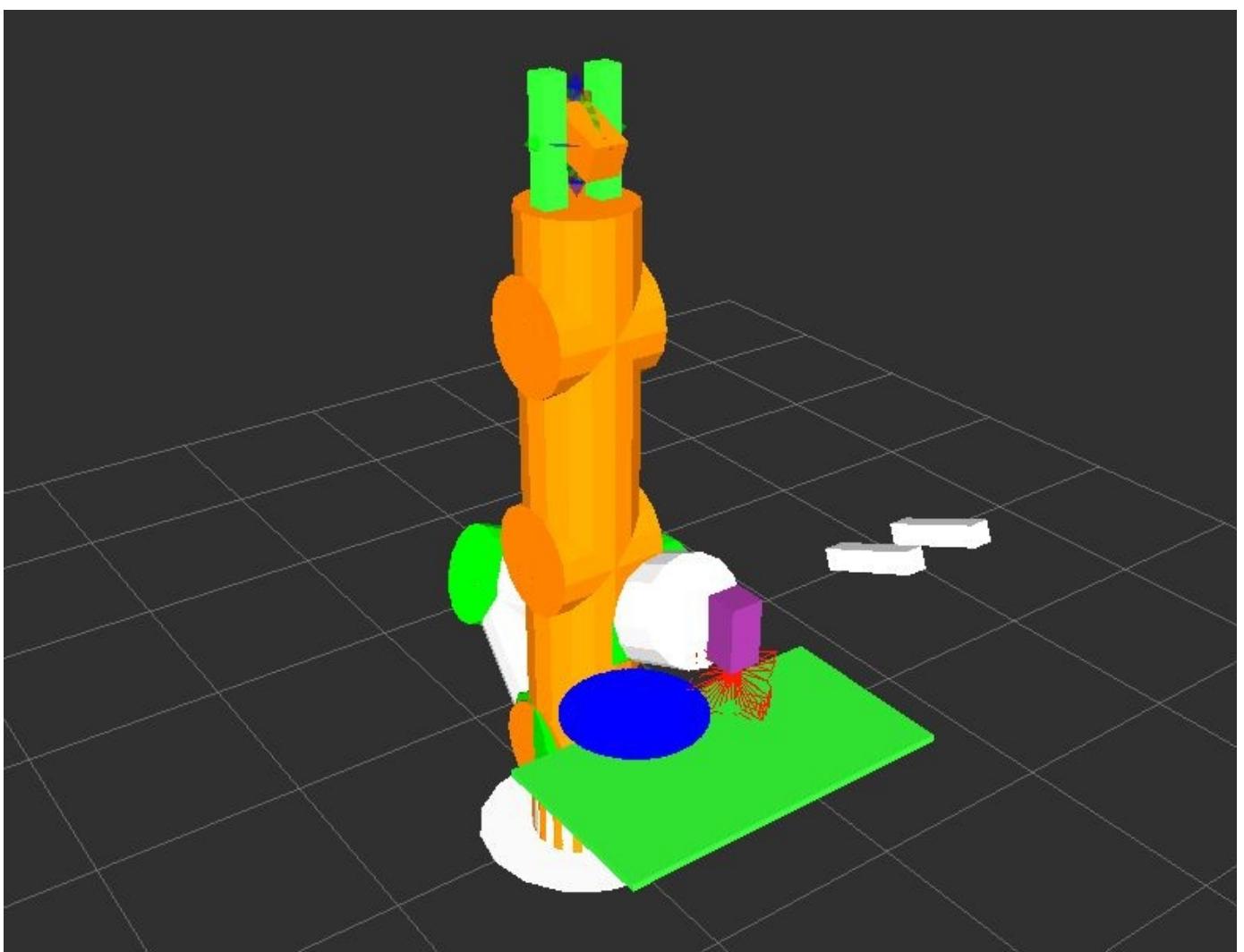
```
# Place Coke can object on another place on the support surface  
(table):while not self._place(self._arm_group, self._grasp_object_name,  
self._pose_place):  
    rospy.logwarn('Place failed! Retrying ...')  
    rospy.sleep(1.0)
```

The `_place` method uses the following code:

```
def _place(self, group, target, place):  
    # Obtain possible places:  
    places = self._generate_places(place)  
  
    # Create and send Place goal:  
    goal = self._create_place_goal(group, target, places)  
  
    state = self._place_ac.send_goal_and_wait(goal)  
    if state != GoalStatus.SUCCEEDED:  
        rospy.logerr('Place goal failed!: %s' %  
                    self._place_ac.get_goal_status_text())  
        return None  
  
    result = self._place_ac.get_result()  
  
    # Check for error:  
    err = result.error_code.val  
    if err != MoveItErrorCode.SUCCESS:  
        rospy.logwarn('Group %s cannot place target %s!: %s' % (group,  
                                                               target, str(moveit_error_dict[err])))  
        return False  
  
    return True
```

The method generates multiple possible places to leave the object, creates the place goal, and sends it. Then, it checks the result to verify whether the object has been placed or not. To place an object, we can use a single place pose, but it is generally better to provide several options. In this case, we have the `_generate_places` method, which generates places with different angles at the position given.

When the places are generated, they are also published as `PoseArray` so we can see them with blue arrows, as shown in the following screenshot:



Visualization of place poses

Once the places are obtained, the `_create_place_goal` method creates a place goal as follows:

```
def _create_place_goal(self, group, target, places):
    # Create goal:
    goal = PlaceGoal()

    goal.group_name = group
    goal.attached_object_name = target

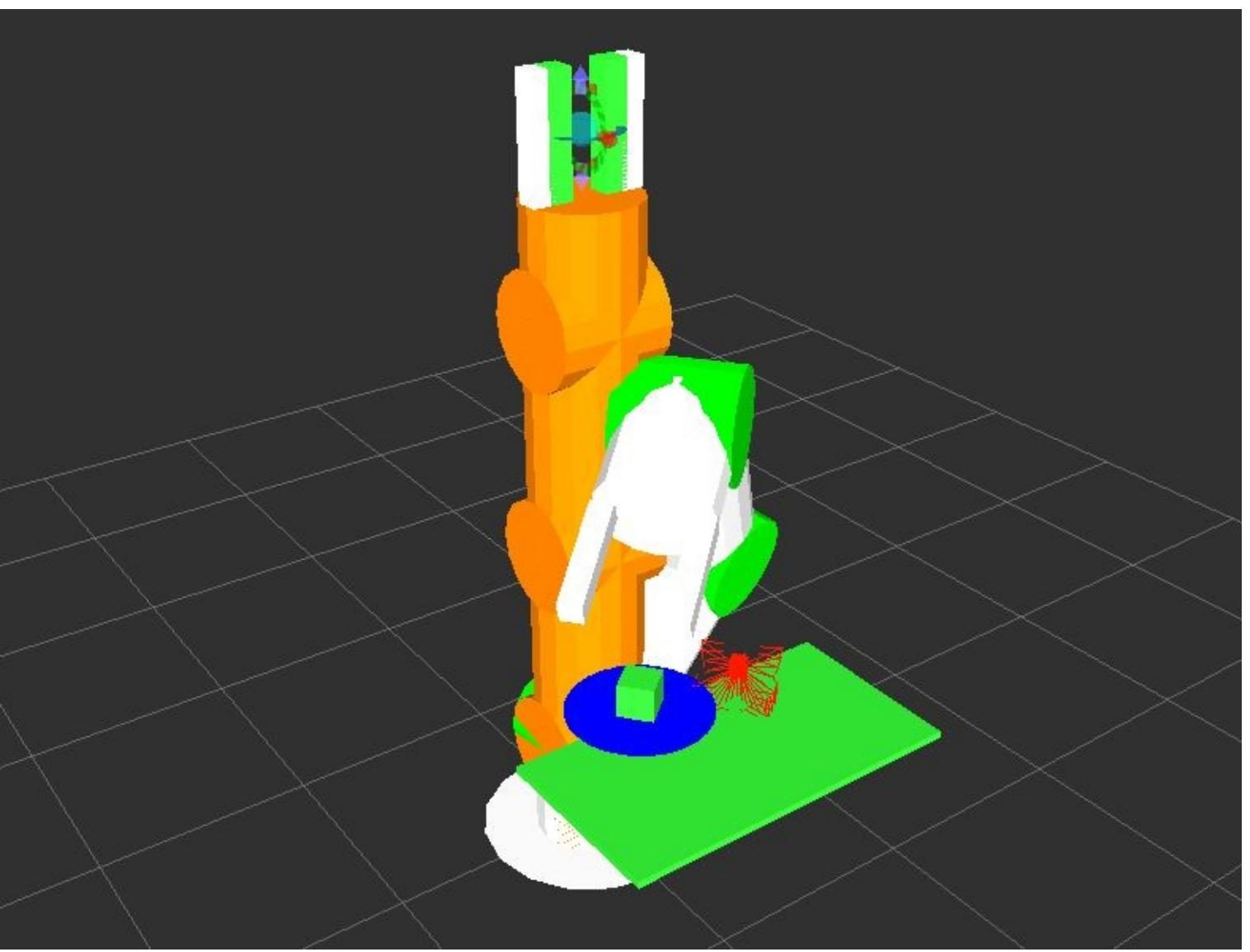
    goal.place_locations.extend(places)

    # Configure goal planning options:
    goal.allowed_planning_time = 5.0

    goal.planning_options.planning_scene_diff.is_diff = True
    goal.planning_options.planning_scene_diff.robot_state.is_diff =
        True
    goal.planning_options.plan_only = False
    goal.planning_options.replan = True
    goal.planning_options.replan_attempts = 10

    return goal
```

In brief, the place goal has the group (`arm` in this case) and the target object (`coke_can` in this case), which are attached to the gripper and the place or places (poses). Additionally, several planning options are provided, along with the allowed planning time, which can be increased if needed. When the object is placed, we will see the box representing it in green again, and on top of the table. The arm will be up again, as shown in the following screenshot:



Arm after placing an object

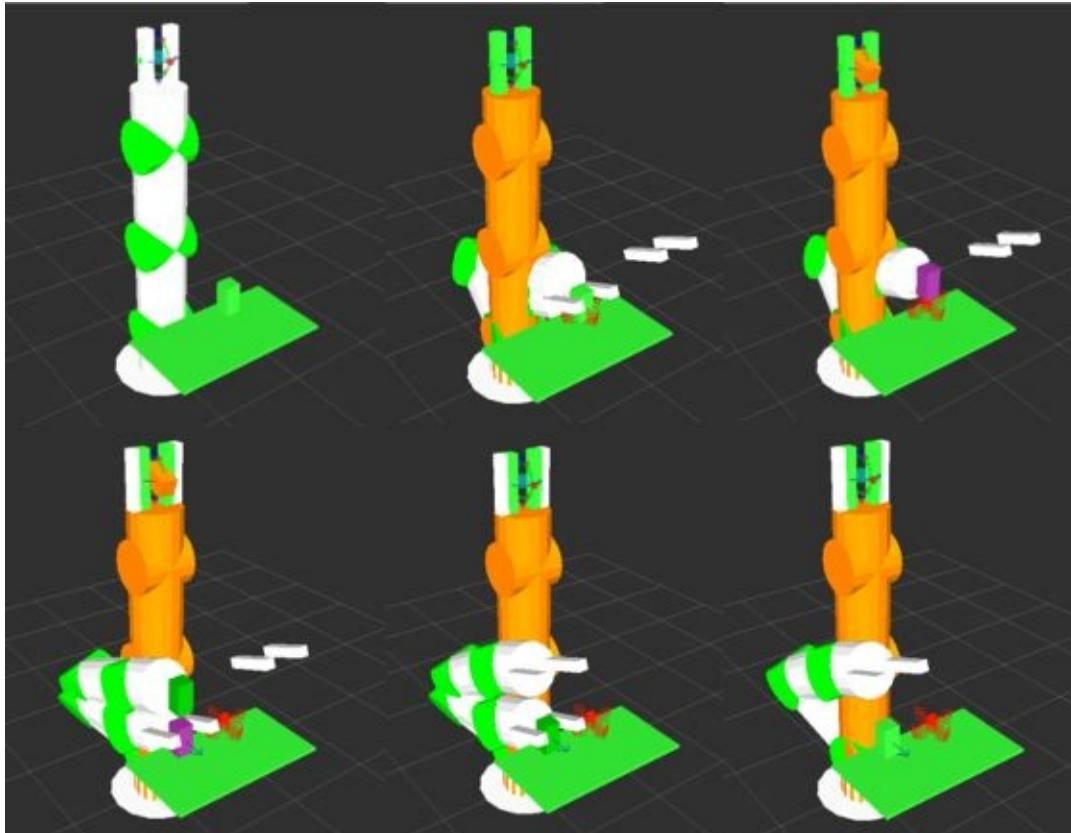
The demo mode

We can do the whole *pick and place* task without perception in demo mode, that is, without actually actuating on the Gazebo simulation, or the real robotic arm. In this mode, we will use fake controllers to move the arm once the motion plan has been found by MoveIt! to do the pick and place actions, including grasping the object. The same code can be used directly on the actual controllers.

In order to run pick and place in the demo mode, run the following commands:

```
$ roslaunch rosbook_arm_moveit_config demo.launch  
$ roslaunch rosbook_arm_pick_and_place grasp_generator_server.launch  
$ rosrunros book_arm_pick_and_place pick_and_place.py
```

The special part is the first launch file, which simply opens RViz and loads fake controllers instead of spawning the robotic manipulator on Gazebo. The following figure shows several snapshots of the arm moving and doing the pick and place after running the preceding commands:



Arm doing pick and place task in demo mode

Simulation in Gazebo

Using the same code as in demo mode, we can actually move the real controllers, either in simulation (Gazebo) or using the real hardware. The interface is the same, so using a real arm or Gazebo is exactly the same. In these cases, the joints will actually move and the grasping will actually make the gripper come in contact with the grasping object (the can of Coke). This requires a proper definition of the objects and the gripper in Gazebo to work properly.

The commands to run the pick and place in this case (as shown previously) are:

```
$ roslaunch rosbook_arm_gazebo rosbook_arm_grasping_world.launch
$ roslaunch rosbook_arm_moveit_config moveit_rviz.launch config:=true
$ roslaunch rosbook_arm_pick_and_place grasp_generator_server.launch
$ rosrun rosbook_arm_pick_and_place pick_and_place.py
```

It is only the first part, to launch files, that changes with respect to the demo mode; it replaces `demo.launch` in the demo mode. In this case, we spawn the robotic manipulator in Gazebo with the environment containing the table and the can of Coke as well as the RGB-D camera. Then, the `moveit_rviz.launch` file opens RViz with the MoveIt! plugin providing the same interface as with `demo.launch`. However, in this case, when the `pick_and_place.py` program is run, the arm in Gazebo is moved.

Summary

In this chapter, we have covered most of the aspects involved in integrating a robotic arm with MoveIt! and Gazebo, which gives us a realistic view of how a robotic arm could be used in a real-life environment. MoveIt! provides us with very simple and concise tools for motion planning with robotic arms using an Inverse Kinematics (IK) solver as well as ample documentation in order to facilitate this process, but given the complexity of the architecture, it can only be done properly once all of the different interactions between MoveIt!, the sensors, and the actuators in our robot have been properly understood.

We have glanced through the different high-level elements in the MoveIt! API, which, to cover in detail, would require an entire book of their own. In an attempt to avoid the cost of understanding a full API to perform very simple actions, the approach taken in this section has been to limit ourselves to very simple motion planning and interacting with both artificially created objects in the planning scene and RGB-D sensors that generate a point cloud.

Finally, a very detailed explanation has been provided for how to perform an object pick and place task. Although not being the sole purpose of a robotic arm, this is one that you might enjoy experimenting with as it is very common in industrial robots, but using MoveIt! motion planning and 3D perception allows you to do so in complex and dynamic environments. For this particular purpose of the robotic arm, a deeper look at the APIs and enhanced understanding of the MoveIt! architecture was required, giving you much more depth and understanding as to the possibilities of MoveIt!

Using Sensors and Actuators with ROS

When you think of a robot, you would probably think of a human-sized one with arms, a lot of sensors, and a wide field of locomotion systems.

Now that we know how to write programs in ROS and that we know how to create robot models and simulate their behavior, we are going to work with real sensors and actuators-things that can interact with the real world. In the first section, we are going to learn how to control a DIY robot platform using ROS. Then, we are going to see how to interface different sensors and actuators that you can typically find in robotics laboratories.



You can find a wide list of devices supported by ROS at <http://www.ros.org/wiki/Sensors>.

Sensors and actuators can be organized into different categories: rangefinders, cameras, pose estimation devices, and so on. They will help you find what you are looking for more quickly.

In this chapter, we will deal with the following topics:

- Cheap and common actuators and sensors for your projects
- Using Arduino to connect them
- 2D laser rangefinders, Kinect 3D sensors, GPS, or servomotors

We know that it is impossible to explain all the types of sensors in this chapter. For this reason, we have selected some of the most commonly used sensors that are affordable to most users-regular, sporadic, or amateur. With them, we are going to connect all the elements needed to have a small robot platform working.

Using a joystick or a gamepad

I am sure that, at one point or another, you have used a joystick or a gamepad of a video console.

A joystick is nothing more than a series of buttons and potentiometers. With this device, you can perform or control a wide range of actions.



In ROS, a joystick is used to telecontrol a robot to change its velocity or direction.

Before we start, we are going to install some packages. To install these packages in Ubuntu, execute the following command:

```
| $ sudo apt-get install ros-kinetic-joystick-drivers  
| $ rosstack profile & rospack profile
```

In these packages, you will find code to learn how to use the joystick and a guide to create our packages.

First of all, connect your joystick to your computer. Now, we are going to check whether the joystick is recognized using the following command:

```
| $ ls /dev/input/
```

We will see the following output:

```
| by-id event0 event2 event4 event6 event8 js0 mouse0  
| by-path event1 event3 event5 event7 event9 mice
```

The port created is `js0`; with the `jstest` command, we can check whether it is working, using the following code:

```
| $ sudo jstest /dev/input/js0  
| Axes: 0: 0 1: 0 2: 0 Buttons: 0:off 1:off 2:off 3:off 4:off 5:off 6:off 7:off 8:off 9:off 10:off
```

Our joystick, *Logitech F710*, has 8 axes and 11 buttons, and if we move the joystick, the values change.

Once you have checked the joystick, we are going to test it in ROS. To do this, you can use the `joy` and `joy_node` packages:

```
| $ rosrun joy joy_node
```

If everything is OK, you will see the following output:

```
| [ INFO] [1357571588.441808789]: opened joystick: /dev/input/js0. deadzone_: 0.050000.
```

How does joy_node send joystick movements?

With the `joy_node` package active, we are going to see the messages sent by this node. This will help us understand how it sends information about axes and buttons.

To see the messages sent by the node, we can use this command:

```
| $ rostopic echo /joy
```

And then, we can see each message sent as follows:

```
header:  
  seq: 33  
  stamp:  
    secs: 1480289803  
    nsecs: 599782892  
    frame_id: ''  
  axes: [-0.0, -0.2219386249780055, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]  
  buttons: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

You will see two main vectors: one for axes and the other for buttons. Obviously, these vectors are used to publish the states of the buttons and axes of the real hardware.

If you want to know the message type, type the following command line in a shell:

```
| $ rostopic type /joy
```

You will then obtain the type used by the message; in this case, it is `sensor_msgs/Joy`.

Now, to see the fields used in the message, use the following command line:

```
| $ rosmsg show sensor_msgs/Joy
```

You will see the following output:

```
std_msgs/Header header  
  uint32 seq  
  time stamp  
  string frame_id  
float32[] axes  
int32[] buttons
```

This is the message structure that you must use if you want to use a joystick with your developments. In the next section, you will learn how to write a node that subscribes to the joystick topic and how to generate moving commands to move a robot model.

Using joystick data to move our robot model

Now, we are going to create a node that gets data from `joy_node` and published topics to control a robot model.

Now, it is necessary to create a robot 3D model like we have seen in the previous chapters and a node with the odometry. In the repository, you can check the `urdf` model in the `chapter8_tutorials/urdf` folder and you can check the code for odometry in the `/src` folder where the file is named `c8_odom.cpp`.

In this code, odometry is calculated using an initial position $0, 0, 0$ and using velocity information from a `geometry_msgs::Twist`. The velocity of each wheel is calculated, and the position is updated using differential drive kinematics. You can run the odometry node after a `roscore` command typing the following in a terminal:

```
| $ rosrun chapter8_tutorials c8_odom
```

To understand the topics used in this node, we can display the topic list; use the following command line:

```
| $ rosnode info /odom
```

You will then see the following output, where `/cmd_vel` is the topic published by the node:

```
-----
Node [/odom]
Publications:
 * /odom [nav_msgs/Odometry]
 * /rosout [rosgraph_msgs/Log]
 * /tf [tf2_msgs/TFMessage]

Subscriptions:
 * /cmd_vel [unknown type]

Services:
 * /odom/get_loggers
 * /odom/set_logger_level

contacting node http://daneel:35582/ ...
Pid: 30375
Connections:
 * topic: /rosout
   * to: /rosout
   * direction: outbound
   * transport: TCPROS
```

We can check the topic type of `/cmd_vel`. Use the following command line to know it:

```
| $ rostopic type /cmd_vel
```

You will see this output:

```
| geometry_msgs/Twist
```

To know the contents of this message, execute the following command line:

```
| $ rosmsg show geometry_msgs/Twist
```

You will then see the two fields that are used to send the velocity:

```
geometry_msgs/Vector3 linear
  float64 x
  float64 y
  float64 z
geometry_msgs/Vector3 angular
  float64 x
  float64 y
  float64 z
```

OK, now that we have localized the topic and the structure to use, it is time to create a program to generate velocity commands using data from the joystick.

Create a new file, `c8_teleop_joy.cpp`, in the `chapter8_tutorials/src` directory and type in the following code snippet:

```
#include<ros/ros.h>
#include<geometry_msgs/Twist.h>
#include<sensor_msgs/Joy.h>
#include<iostream>

using namespace std;
float max_linear_vel = 0.2;
float max_angular_vel = 1.5707;

class TeleopJoy{
public:
TeleopJoy();
private:
void callBack(const sensor_msgs::Joy::ConstPtr& joy);
ros::NodeHandle n;
ros::Publisher pub;
ros::Subscriber sub;
int i_velLinear , i_velAngular;
};

TeleopJoy::TeleopJoy()
{ i_velLinear = 1;
i_velAngular = 0;
n.param("axis_linear",i_velLinear,i_velLinear);
n.param("axis_angular",i_velAngular,i_velAngular);
pub = n.advertise<geometry_msgs::Twist>("/cmd_vel",1);
sub = n.subscribe<sensor_msgs::Joy>("joy", 10,
&TeleopJoy::callBack, this);
}

void TeleopJoy::callBack(const sensor_msgs::Joy::ConstPtr& joy)
{
geometry_msgs::Twist vel;
vel.angular.z = max_angular_vel*joy->axes[0];
vel.linear.x = max_linear_vel*joy->axes[1];
pub.publish(vel);
}

int main(int argc, char** argv)
{
ros::init(argc, argv, "c8_teleop_joy");
TeleopJoy teleop_joy;
ros::spin();
}
```

Now, we are going to break the code to explain how it works. In the `main` function, we create an instance of the `TeleopJoy` class:

```
int main(int argc, char** argv)
{
  TeleopJoy teleop_joy;
```

In the constructor, four variables are initialized. The first two variables are filled using data from Parameter Server. These variables are joystick axes. The next two variables are the advertiser and the subscriber. The advertiser will publish a topic with the `geometry_msgs::Twist` type. The subscriber will get data from the topic with the name `joy`. The node that is handling the joystick sends this topic:

```
TeleopJoy::TeleopJoy()
{ i_velLinear = 1;
  i_velAngular = 0;
  n.param("axis_linear", i_velLinear, i_velLinear);
  n.param("axis_angular", i_velAngular, i_velAngular);
  pub = n.advertise<geometry_msgs::Twist>("/cmd_vel", 1);
  sub = n.subscribe<sensor_msgs::Joy>("joy", 10,
    &TeleopJoy::callBack, this);
}
```

Each time the node receives a message, the `callBack()` function is called. We create a new variable with the name `vel`, which will be used to publish data. The values of the axes of the joystick are assigned to the `vel` variable using coefficients to assign the maximum velocities. In this part, you can create a process with the data received before publishing it:

```
void TeleopJoy::callBack(const sensor_msgs::Joy::ConstPtr& joy)
{
  geometry_msgs::Twist vel;
  vel.angular.z = max_angular_vel*joy->axes[0];
  vel.linear.x = max_linear_vel*joy->axes[1];
  pub.publish(vel);
}
```

Finally, the topic is published using `pub.publish(vel)`. We are going to create a launch file for this example. In the launch file, we declare data for Parameter Server and launch the `joy` and `c8_teleop_joy` nodes:

```
<?xml version="1.0" ?>
<launch>
  <node pkg="chapter8_tutorials" type="c8_teleop_joy"
    name="c8_teleop_joy" />
  <param name="axis_linear" value="1" type="int" />
  <param name="axis_angular" value="0" type="int" />
  <node respawn="true" pkg="joy" type="joy_node" name="joy_node">
    <param name="dev" type="string" value="/dev/input/js0" />
    <param name="deadzone" value="0.12" />
  </node>
</launch>
```

There are four parameters in the launch file; these parameters will add data to Parameter Server, and it will be used by our node. The `axis_linear` and `axis_angular` parameters will be used to configure the axes of the joystick. If you want to change the axes configuration, you only need to change the value and put the number of the axes you want to use. The `dev` and `deadzone` parameters will be used to configure the port where the joystick is connected, and the dead zone is the region of movement that is not recognized by the device.

```
$ rosrun chapter8_tutorials chapter8_teleop_joy.launch
```

You can check that the code is working using `rostopic echo` that `/cmd_vel` topic is published when you move the joystick axes.

Now, we are going to prepare a larger launch to visualize a robot model that moves with the joystick. Copy the following code step to a new file, `chapter8_tutorials_robot_model.launch`, in the

chapter8_tutorials/launch directory:

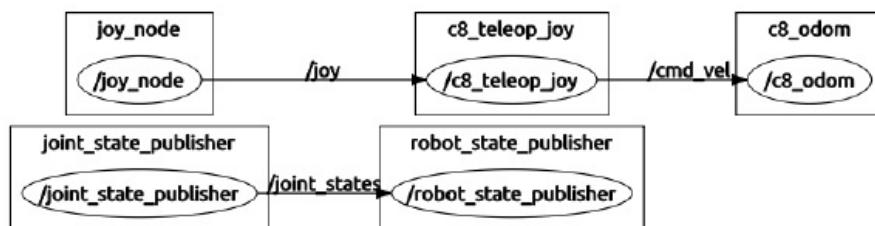
```
<?xml version="1.0"?>
<launch>
  <arg name="model" />
  <arg name="gui" default="False" />
  <param name="robot_description" textfile="$(find chapter8_tutorials)/urdf/robot2.urdf" />
  <param name="use_gui" value="$(arg gui)" />
  <node name="joint_state_publisher" pkg="joint_state_publisher" type="joint_state_publisher" /> <node name="robot_state_publisher" pkg="robot_state_publisher" type="state_publisher" /> <node name="rviz" pkg="rviz" type="rviz" args="-d $(find chapter8_tutorials)/config/config.rviz" />
  <node name="c8_odom" pkg="chapter8_tutorials" type="c8_odom" />
  <node name="joy_node" pkg="joy" type="joy_node" />
  <node name="c8_teleop_joy" pkg="chapter8_tutorials" type="c8_teleop_joy" />
</launch>
```

You will note that, in the launch file, there are four different nodes: `c8_teleop_joy`, `joy_node`, `c8_odom`, `joint_state_publisher`, and `rviz`. You can note in the launch file that the robot model is named `robot2.urdf` and you can find it in `chapter8_tutorials/urdf`.

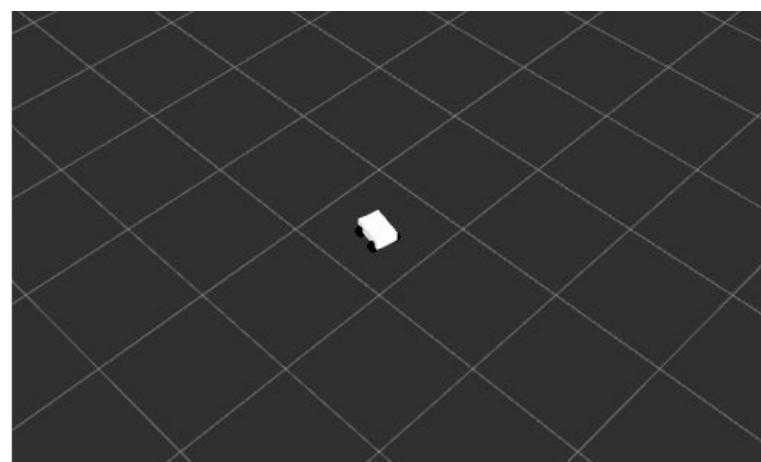
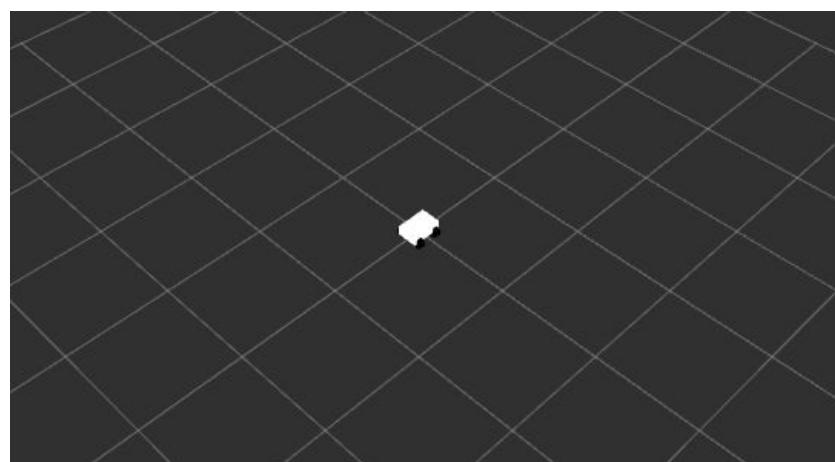
To launch the example, use the following command line:

```
| $ roslaunch chapter8_tutorials chapter8_robot_model.launch
```

You can see whether everything is fine by checking the running nodes and the topic list using the `rosnode` and `rostopic` lists. If you want to see it graphically, use `rqt_graph`.



If all has been successful, you should see in the RViz visualizer a robot model that you can control with the joystick!



In the next sections, we will learn how to use Arduino with ROS. Then, we will connect the joystick example nodes to Arduino node to control real robot motors.

Using Arduino to add sensors and actuators

Arduino is an open source electronics prototyping platform based on flexible, easy-to-use hardware and software. It's intended for artists, designers, hobbyists, and anyone interested in creating interactive objects or environments.

The following image shows how an Arduino board looks:



ROS can use this type of device with the `rosserial` package. Basically, Arduino is connected to the computer using a serial connection, and data is transmitted using this port. With `rosserial`, you can also use a lot of devices controlled by a serial connection, for example, GPS and servo controllers.

First, we need to install the packages. To do this, we use the following command lines:

```
$ sudo apt-get install ros-kinect-rosserial-arduino  
$ sudo apt-get install ros-kinect-rosserial
```

Then, for the `catkin` workspace, we need to clone the `rosserial` repository into the workspace. The `rosserial` messages are created, and `ros_lib` is compiled with the following command lines:

```
$ cd dev/catkin_ws/src/  
$ git clone https://github.com/ros-drivers/rosserial.git  
$ cd dev/catkin_ws/  
$ catkin_make  
$ catkin_make install  
$ source install/setup.bash
```

OK, we assume that you have the Arduino IDE installed. If not, just follow the steps described at <http://arduino.cc/en/Main/Software>. For ROS Kinetic, Arduino core is installed with the `rosserial_arduino` package. You can download new versions of Arduino IDE from www.arduino.cc.

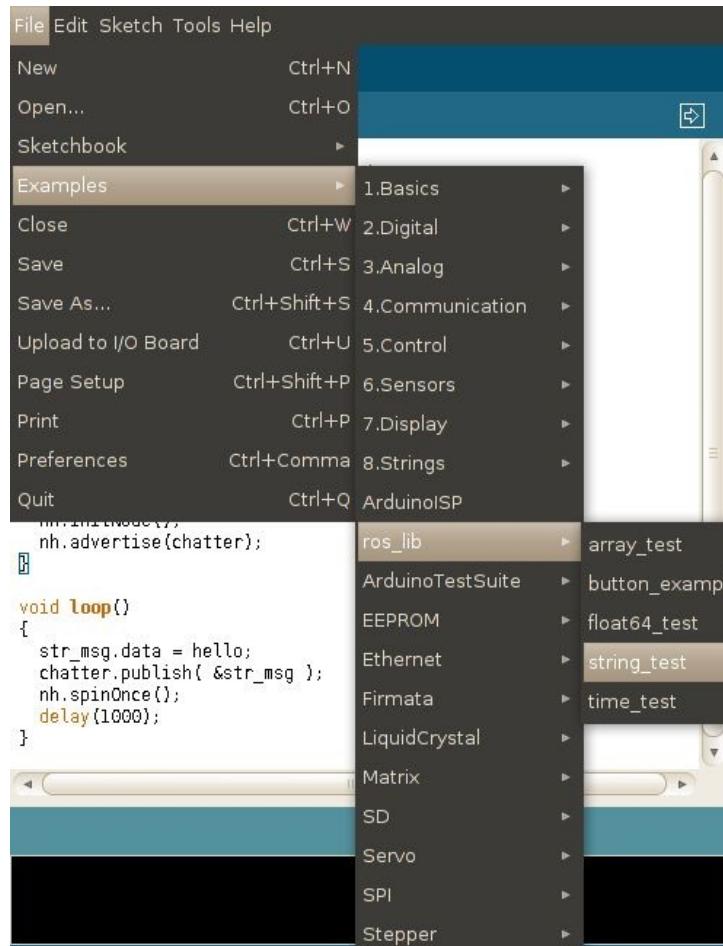
In the new versions of Arduino IDE, the `sketchbook` folder is now named `Arduino` and is in your home folder.

Once you have the package and the IDE installed, it is necessary to copy `ros_lib` from the `rosserial` package to the `sketchbook/libraries` folder, which is created on your computer after running the Arduino IDE. Then, you have to run `make_libraries.py`:

```
$ cd ~/Arduino/libraries  
$ sudo rm -r roslib  
$ rosrun rosserial_arduino make_libraries.py .
```

Creating an example program to use Arduino

Now, we are going to upload an example program from the IDE to Arduino. Select the `Hello World` sample and upload the sketch:



Arduino IDE with roslib examples

The code in the preceding screenshot is very similar to the following code. In the following code, you can see an include line with the `ros.h` library. This library is the `roserial` library, which we have installed before. Also, you can see a library with the message to send with a topic; in this case, it is the `std_msgs/String` type.

The following code snippet is present in the `c8_arduino_string.ino` file:

```
#include <ros.h>
#include <std_msgs/String.h>

ros::NodeHandle nh;

std_msgs::String str_msg;
ros::Publisher chatter("chatter", &str_msg);
char hello[19] = "chapter8_tutorials";

void setup()
{
    nh.initNode();
    nh.advertise(chatter);
}
```

```

void loop()
{
    str_msg.data = hello;
    chatter.publish( &str_msg );
    nh.spinOnce();
    delay(1000);
}

```

The Arduino code is divided into two functions: `setup()` and `loop()`. The `setup()` function is executed once and is usually used to set up the board. After `setup()`, the `loop()` function runs continuously. In the `setup()` function, the name of the topic is set; in this case, it is named `chatter`. Now, we need to start a node to hear the port and publish the topics sent by Arduino on the ROS network. Type the following command in a shell and remember to run `roscore`:

```
| $ rosrun rosserial_python serial_node.py /dev/ttyACM0
```

Now, you can see the messages sent by Arduino with the `rostopic echo` command:

```
| $ rostopic echo chatter
```

You will see the following data in the shell:

```
| data: chapter8_tutorials
```

The last example is about the data sent from Arduino to the computer. Now, we are going to use an example where Arduino will subscribe to a topic and will change the LED state connected to the pin number 13. The name of the example that we are going to use is `blink`; you can find this in the Arduino IDE by navigating to File | Examples | `ros_lib` | `Blink`.

The following code snippet is present in the `c8_arduino_led.ino` file:

```

#include <ros.h>
#include <std_msgs/Empty.h>

ros::NodeHandle nh;
void messageCb( const std_msgs::Empty& toggle_msg){
    digitalWrite(13, HIGH-digitalRead(13)); // blink the led
}

ros::Subscriber<std_msgs::Empty> sub("toggle_led", &messageCb );
void setup()
{
    pinMode(13, OUTPUT);
    nh.initNode();
    nh.subscribe(sub);
}

void loop()
{
    nh.spinOnce();
    delay(1);
}

```

Remember to launch the node to communicate with the Arduino board:

```
| $ rosrun rosserial_python serial_node.py /dev/ttyACM0
```

Now, if you want to change the LED status, you can use the `rostopic pub` command to publish the new state:

```
| $ rostopic pub /toggle_led std_msgs/Empty "{}" -once
publishing and latching message for 3.0 seconds
```

You will note that the LED has changed its status; if the LED was on, it will now turn off. To change the status again, you only have to publish the topic once more:

```
| $ rostopic pub /toggle_led std_msgs/Empty "{}" -once  
| publishing and latching message for 3.0 seconds
```

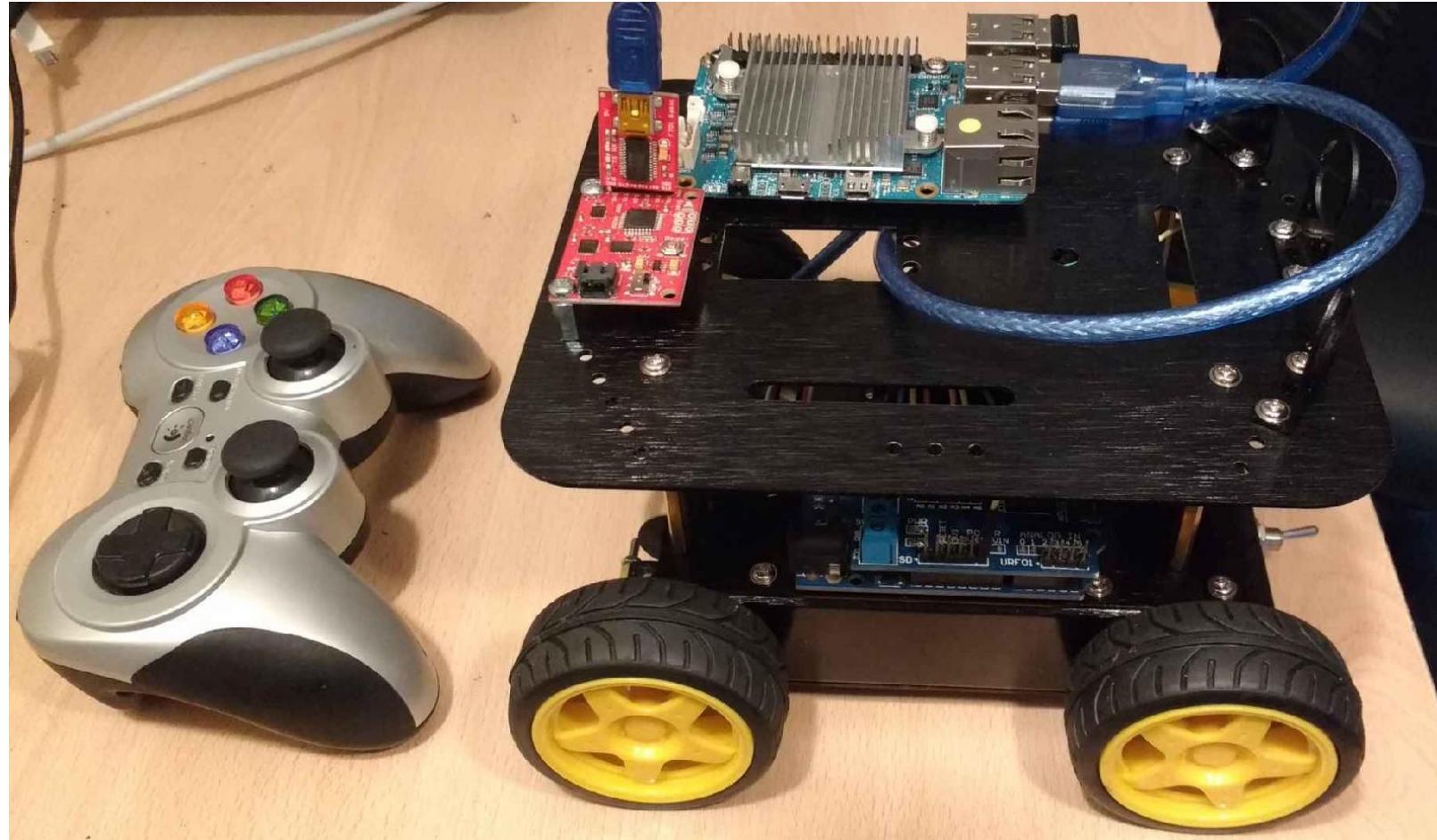
Now, you can use all the devices available to Arduino on ROS. This is very useful because you have access to cheap sensors and actuators to implement your robots.



When we were writing the chapter, we noted that Arduino does not work with `rosserial`, for instance, in the case of Arduino Leonardo. So, be careful with the selection of the device to use with this package. We didn't face any problems while working with Arduino UNO R3, Genuino, Mega, Arduino Duemilanove, or Arduino Nano.

Robot platform controlled by ROS and Arduino

Now, we have understood how to use Arduino with ROS. In this section, we are prepared to connect our first actuators to ROS. There are a lot of low-cost kits for Arduino users that we can choose to make a robot controlled by ROS. During this section, we are going to use a 4 x 4 robot kit chassis.



4 x 4 robot platform with Odroid C1, Arduino, 9 DOF Razor IMU, and joystick

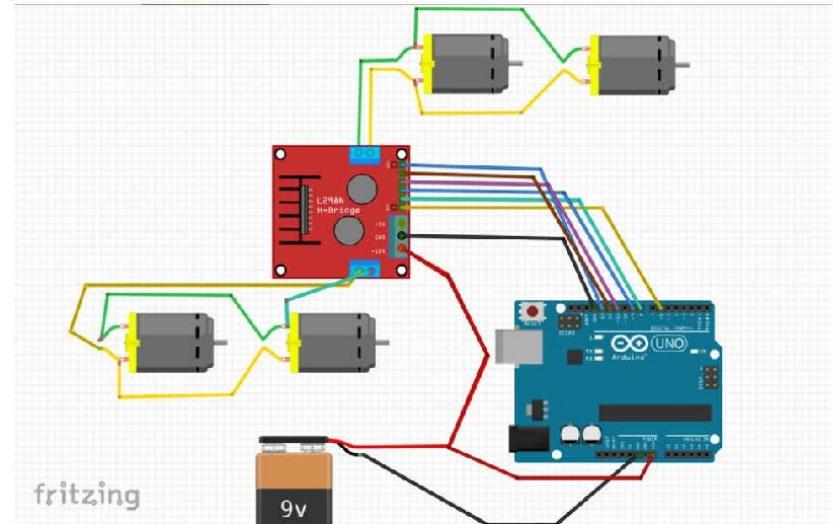
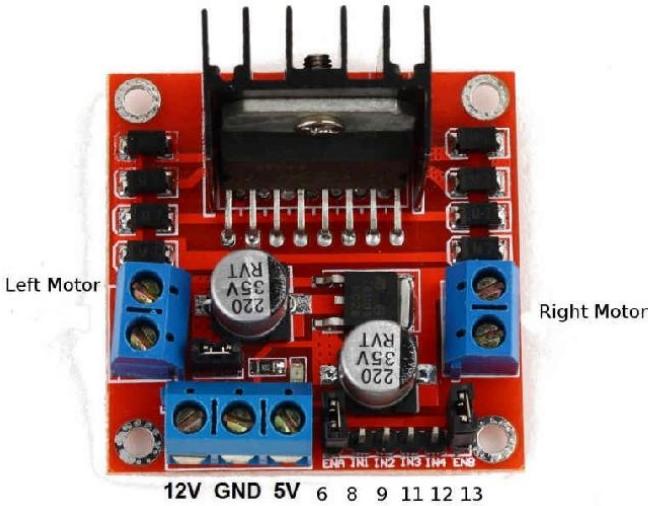
In order to have ROS in our robot, not only an Arduino is needed, but we will also need an embedded computer with ROS installed. There is a lot of ARM based-boards compatible with ROS, such as Dragonboard, Raspberry Pi, or BeagleBone Black. In this case, there is an Odroid C1 with Ubuntu Xenial and ROS Kinetic. During the chapter, we are going to learn how to connect different sensors to the platform such as an **Inertial Measurement Unit (IMU)** or some wheel encoders to calculate the odometry.

The robot platform has four motors that have to be controlled. For this purpose, we are going to use an Arduino with a motor controller. There is a lot of motor controllers that can be interfaced with Arduino. In this case, we are going to use a very common board based on L298N dual motor driver. A kit of two wheels, two motors, and two magnetic encoders are shown in the figure:



Connecting your robot motors to ROS using Arduino

As we have four motors, we will connect the motors from the same side of the platform to the same output. We will connect digital signals to the L298N board to control the motors' behavior.



Each channel needs three signals to be controlled. **IN1** and **IN2** are digital signals used to set the rotation sense of the motor. **ENA** is used to control the intensity of the motor using Arduino **Pulse Width Modulation (PWM)** signal. This signal will control the left motor. For the right motor, **IN3**, **IN4**, and **ENB** will be the control signals. In the figure mentioned earlier, you have a diagram of the connection. Be careful connecting the wires from Arduino to L298N and programming INx signals. Once you have connected the motors to the L298N motor controller and the control signals to Arduino.

We can start a program to control the motors; you can find the entire code in `chapter8_tutorials/src/robot_motors.ino`. We are going to use Arduino IDE to code the sketch. First, we are going to declare the dependencies and define the Arduino pins connected to the L298N motor controller:

```
#include <ros.h>
#include <std_msgs/Int16.h>

#define ENA 6
#define ENB 11
#define IN1 8
#define IN2 9
#define IN3 12
#define IN4 13
```

Then, we are going to declare two callback functions to activate each motor. When a command message for the left wheel is received, the direction of the motors is set depending of the sign of the command. Digital signals **IN1** and **IN2** set the motor direction forward or backward. **ENA** is a PWM signal that controls the motor voltage; this signal is able to regulate the speed of the motor. The callback for the right wheel

command is similar to the first callback:

```
void cmdLeftWheelCB( const std_msgs::Int16& msg)
{
    if(msg.data >= 0)
    {
        analogWrite(ENA,msg.data);
        digitalWrite(IN1, LOW);
        digitalWrite(IN2, HIGH);
    }
    else
    {
        analogWrite(ENA,-msg.data);
        digitalWrite(IN1, HIGH);
        digitalWrite(IN2, LOW);
    }
}

void cmdRightWheelCB( const std_msgs::Int16& msg)
{
    if(msg.data >= 0)
    {
        analogWrite(ENB,msg.data);
        digitalWrite(IN3, LOW);
        digitalWrite(IN4, HIGH);
    }
    else
    {
        analogWrite(ENB,-msg.data);
        digitalWrite(IN3, HIGH);
        digitalWrite(IN4, LOW);
    }
}
```

Then, two subscribers are declared, the topics are named `cmd_left_wheel`, we are using `std::msgs::Int16` topic type, and we have just seen the callback functions:

```
ros::Subscriber<std_msgs::Int16> subCmdLeft("cmd_left_wheel",
cmdLeftWheelCB );ros::Subscriber<std_msgs::Int16>
subCmdRight("cmd_right_wheel",cmdRightWheelCB );
```

In the Arduino `setup()` function, the pins are declared as outputs and `nh` node is initialized and subscription to the topics starts:

```
void setup() {
    // put your setup code here, to run once:
    pinMode(ENA, OUTPUT);
    pinMode(ENB, OUTPUT);
    pinMode(IN1, OUTPUT);
    pinMode(IN2, OUTPUT);
    pinMode(IN3, OUTPUT);
    pinMode(IN4, OUTPUT);
    digitalWrite(ENA,0);
    digitalWrite(ENB, 0);
    digitalWrite(IN1, LOW);
    digitalWrite(IN2, HIGH);
    digitalWrite(IN3, LOW);
    digitalWrite(IN4, HIGH);

    nh.initNode();
    nh.subscribe(subCmdRight);
    nh.subscribe(subCmdLeft);
}
```

The `loop` function is as follows:

```
void loop()
{
    nh.spinOnce();
}
```

You can upload now the code to the board, and run the Arduino node from terminal:

```
| $ roscore  
| $ rosrun rosserial_python serial_node.py /dev/ttyACM0
```

In other terminal, you can publish manually some commands to the left and the right motor:

```
| $ rostopic pub /cmd_right_wheel std_msgs/Int16 "data: 190"  
| $ rostopic pub /cmd_left_wheel std_msgs/Int16 "data: -100"
```

The command signals are expected to be between -255 and 255, and motors should stop with a zero value.

Now that we have our robot motor connected to ROS, it is time to control them with a joystick. We can unify the velocity topics, using differential drive kinematics equations and ROS `geometry_msgs::Twist` topic. By this way, we can send a standard message to the odometry nodes that is independent of the type of robot that we are using. As we have prepared `c8_teleop_joy` node to send this type of topics, we will be able to control the wheels with the joystick.

We are going to create a new sketch name `robot_motors_with_twist.ino`. We are going to copy the previous sketch, and we will have to add a new include at the beginning of the sketch to use `geometry_msgs::Twist`:

```
| #include <geometry_msgs/Twist.h>  
| float L = 0.1; //distance between wheels
```

Then, we will include a new subscriber and its callback function:

```
void cmdVelCB( const geometry_msgs::Twist& twist)  
{  
    int gain = 4000;  
    float left_wheel_data = gain*(twist.linear.x -  
        twist.angular.z*L);  float right_wheel_data = gain*(twist.linear.x +  
        twist.angular.z*L);  
    {  
        analogWrite(ENA,abs(left_wheel_data));  
        digitalWrite(IN1, LOW);  
        digitalWrite(IN2, HIGH);  
    }  
    else  
    {  
        analogWrite(ENA,abs(left_wheel_data));  
        digitalWrite(IN1, HIGH);  
        digitalWrite(IN2, LOW);  
    }  
    if(right_wheel_data >= 0)  
    {  
        analogWrite(ENB,abs(left_wheel_data));  
        digitalWrite(IN3, LOW);  
        digitalWrite(IN4, HIGH);  
    }  
    else  
    {  
        analogWrite(ENB,abs(left_wheel_data));  
        digitalWrite(IN3, HIGH);  
        digitalWrite(IN4, LOW);  
    }  
}  
ros::Subscriber<geometry_msgs::Twist> subCmdVel("cmd_vel",  
cmdVelCB);
```

Now, you can upload the code to the Arduino board. In order to make easier to run all the nodes, we are going to create a launch file that manages the joystick, the robot model in the RViz visualizer, and the

Arduino node.

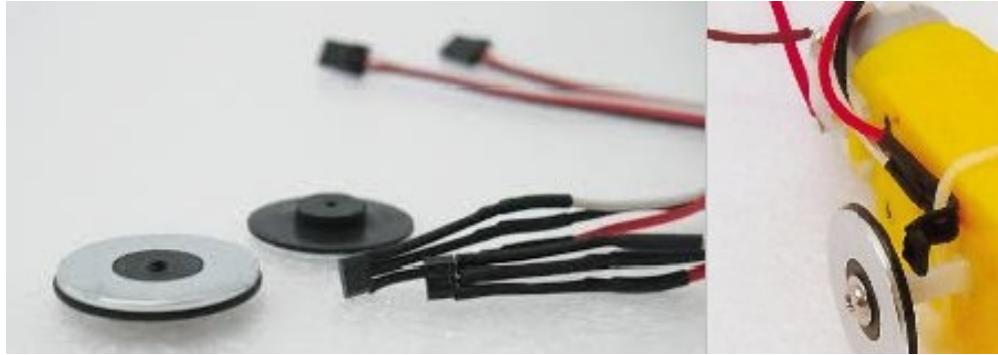
You can launch the example with the following command in terminals:

```
| $ roslaunch chapter8_tutorials chapter8_tutorials robot_model_with_motors.launch  
| $ rosrun rosserial_python serial_node.py /dev/ttyACM0
```

Now, you can control the robot with the joystick; also you can see in the RViz visualizer the robot model moving. If you are testing the code on an embedded ARM computer without display, you can comment the `rviz` node in the launch file.

Connecting encoders to your robot

Dagu encoders are composed of two parts for each wheel. A magnetic disk with eight poles: four north poles and four south poles and a Hall effect sensor. The magnet disk is placed in a shaft of a wheel and the Hall effect must face it at a maximum distance of 3 mm.



Dagu encoders: sensors and magnet, example of installation

Once you have placed the encoder on the wheel, you can connect it to the Arduino. The Hall effect sensor is directly soldered to three wires: black, ground, and white.

The black wire is the ground of the sensor and must be connected to Arduino ground; the red one is for power supply and should be connected to 5V. The white wire is the signal; the signal is binary and changes his values from high level to low level once the magnet disk turns and a different pole faces the sensor.

So, it is logical to think in connecting signal wire to an Arduino digital pin. But furthermore, encoders' signals can change very quickly depending on the revolutions of the motor and the number of poles/steps of your encoder. It is recommended to connect encoder signals to Arduino interruption pins to achieve a faster reading of the signal changes

For *Arduino/Genuino Uno*, interruption pins are digital pin 2 and digital pin 3. We will connect the encoders to this pin. For checking how the encoder works and learning how you can use Arduino interruptions, we will use the following code:

```
#include <std_msgs/Float32.h>
#include <TimerOne.h>
#define LOOP_TIME 200000
#define left_encoder_pin 3
#define right_encoder_pin 2
```

We declare new variables to count each tick of the encoders and create topics in order to publish the wheel velocities:

```
unsigned int counter_left=0;
unsigned int counter_right = 0;

ros::NodeHandle nh;
std_msgs::Float32 left_wheel_vel;
ros::Publisher left_wheel_vel_pub("/left_wheel_velocity",
&left_wheel_vel);
```

```

std_msgs::Float32 right_wheel_vel;
ros::Publisher right_wheel_vel_pub("/right_wheel_velocity",
&right_wheel_vel);

```

We have to create two functions to handle encoders' interruptions. Each time a change is detected in the right or left encoder pin, a count is increased:

```

void docount_left() // counts from the speed sensor
{
    counter_left++; // increase +1 the counter value
}

void docount_right() // counts from the speed sensor
{
    counter_right++; // increase +1 the counter value
}

```

A timer is used to publish the velocity of each wheel. We use the encoder counters, the wheel radius and the timer duration to calculate speed:
`speed: void timerIsr():`

```

{
    Timer1.detachInterrupt(); //stop the timer
    //Left Motor Speed
    left_wheel_vel.data = counter_left;
    left_wheel_vel_pub.publish(&left_wheel_vel);
    right_wheel_vel.data = counter_right;
    right_wheel_vel_pub.publish(&right_wheel_vel);
    counter_right=0;
    counter_left=0;
    Timer1.attachInterrupt( timerIsr ); //enable the timer
}

```

Finally, in the setup functions, the encoder pins are declared as `INPUT_PULLUP`. This means that Arduino will treat them as inputs and it will connect internally a pull-up resistor to these pins:

```

//Setup for encoders
pinMode(right_encoder_pin, INPUT_PULLUP);
pinMode(left_encoder_pin, INPUT_PULLUP);
Timer1.initialize(LOOP_TIME);
attachInterrupt(digitalPinToInterrupt(left_encoder_pin),
    docount_left, CHANGE); // increase counter when speed sensor pin
    goes HighattachInterrupt(digitalPinToInterrupt(right_encoder_pin),
    docount_right, CHANGE); // increase counter when speed sensor pin
    goes High

```

Also, the messages that will be published have been advertised in the setup:

```

nh.advertise(left_wheel_vel_pub);
nh.advertise(right_wheel_vel_pub);
Timer1.attachInterrupt( timerIsr ); // enable the timer

```

You can upload now the code to the board, and run the Arduino node:

```

$ roscore
$ rosrun rosserial_python serial_node.py /dev/ttyACM0

```

When you type the following in another terminal:

```

$ rostopic list

```

You will see:

```
/cmd_left_wheel  
/cmd_right_wheel  
/diagnostics  
/left_wheel_velocity  
/right_wheel_velocity  
/rosout  
/rosout_agg
```

You can now make an echo of the topic and turn the wheels or you can run `rqt_graph`. Also, you can check the frequency of each publications with `rostopic hz`:

```
subscribed to [/left_wheel_velocity]  
average rate: 5.008  
    min: 0.198s max: 0.201s std dev: 0.00127s window: 5  
average rate: 5.004  
    min: 0.197s max: 0.201s std dev: 0.00146s window: 10  
^Caverage rate: 5.006  
    min: 0.197s max: 0.201s std dev: 0.00156s window: 14  
luis@daneel:~$ rostopic hz /right_wheel_velocity  
subscribed to [/right_wheel_velocity]  
average rate: 5.007  
    min: 0.198s max: 0.201s std dev: 0.00095s window: 5  
average rate: 5.011  
    min: 0.198s max: 0.201s std dev: 0.00083s window: 10  
average rate: 5.009  
    min: 0.198s max: 0.201s std dev: 0.00078s window: 15  
^Caverage rate: 5.007  
    min: 0.198s max: 0.201s std dev: 0.00081s window: 16
```

OK, now we can unify these two topics in `geometry_msgs::Twist` as we have done with `/cmd_vel` previously. We have to create a new `geometry_msgs::Twist` to send the velocity of the platform and a topic publisher for this message:

```
geometry_msgs::Twist sensor_vel;  
ros::Publisher sensor_vel_pub("/sensor_velocity", &sensor_vel);
```

And in the `timerISR()` function, we calculate the linear and angular velocities using kinematics equations:

```
sensor_vel.linear.x = radius*(left_wheel_vel.data +  
    right_wheel_vel.data)/2;  
sensor_vel.linear.y = 0;  
sensor_vel.linear.z = 0;  
sensor_vel.angular.x = 0;  
sensor_vel.angular.y = 0;  
sensor_vel.angular.z = radius*(left_wheel_vel.data +  
    right_wheel_vel.data)/L;  
sensor_vel_pub.publish(&sensor_vel);
```

After that, we can modify the odometry node to subscribe to the real odometry data from our encoder sensors published in the topic `/sensor_vel`:

```
linear:  
  x: 0.046875  
  y: 0.0  
  z: 0.0  
angular:  
  x: 0.0  
  y: 0.0  
  z: 0.9375  
---  
linear:  
  x: 0.046875  
  y: 0.0  
  z: 0.0  
angular:  
  x: 0.0  
  y: 0.0  
  z: 0.9375  
---  
linear:  
  x: 0.046875  
  y: 0.0  
  z: 0.0  
angular:  
  x: 0.0  
  y: 0.0  
  z: 0.9375  
---
```

You can check the example with the following command:

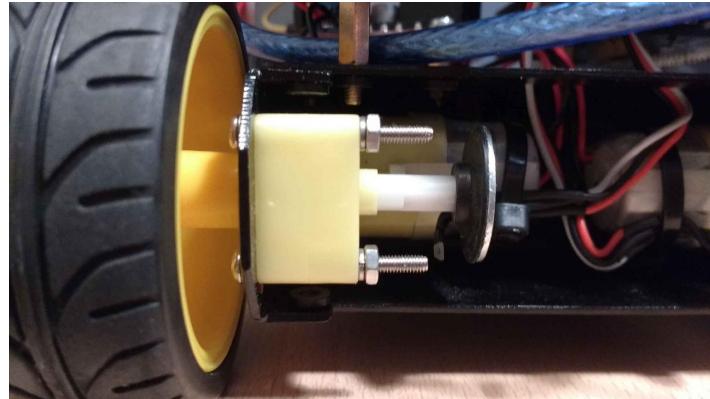
```
$ rosrun chapter8_tutorials chapter8_robot_encoders.launch  
$ rosrun rosserial_python serial_node.py /dev/ttyACM0
```

Controlling the wheel velocity

Now, we can command the wheels of our robot using a joystick, publishing manually `/cmd_vel` topics or using navigation algorithms. Also, we can calculate the odometry of the vehicle and estimate its position; thanks to the encoder data. But have you checked whether your `/sensor_velocity` data is close to `/cmd_vel` command?

If you check our `/cmd_vel` command and the data from `/sensor_velocity`, you will see that they are not equal. You can try to set different gains in your `/cmd_vel` command functions to adjust the velocity command with your real velocity, but it is not the best way to do that. This solution will depend on your power supply, the weight of the vehicle, or the floor surface where you are driving the robot among others.

It is time to go deeper and explore control algorithms. As we have a desired velocity for our robot, `/cmd_vel` and feedback from our encoder sensors, we can implement close-loop control algorithms to reach our `/cmd_vel` in our robot platform. For example, PID algorithms are commonly applied in these situations. So, try to do your own platform control algorithm in Arduino. If you have arrived here, you will be able to do it.



Wheel, motor, and encoders mounted in the platform

In the code repository, you will find an example of close-loop control algorithm with PID, just in case you want to try it without developing your own control algorithm.

Using a low-cost IMU - 9 degrees of freedom

"An inertial measurement unit, or IMU, is an electronic device that measures and reports on a craft's velocity, orientation, and gravitational forces, using a combination of accelerometers and gyroscopes, sometimes also magnetometers. IMUs are typically used to manoeuvre aircraft, including unmanned aerial vehicles (UAVs), among many others, and spacecraft, including satellites and landers." - Wikipedia

In this section, we will learn to use a low-cost sensor with 9 **Degree of Freedom (DoF)**. This sensor has an accelerometer (x3), a magnetometer (x3), a barometer (x1), and a gyroscope (x3). 9DoF Razor IMU and the 9DoF sensor stick are low-cost IMU that can be used in your robotics projects. The two boards have an HMCL5883 magnetometer, an ADXL345, and ITG3200 gyroscope. The old version has different magnetometer. The main difference between the two boards are that the Razor IMU contains an **ATMega328** microcontroller, so the pitch, roll, and yaw are calculated from raw data and transmitted by TTL. For connecting the Razor IMU, you will need a 3.3V TTL to USB converter. The sensor stick has only the three sensors, and it has I2C communication.

Note that the 9DoF Razor IMU is based on an ATMega328, the same microcontroller used by that Arduino UNO, so you can update the firmware or developing your own code easily using Arduino IDE. You can use your own Arduino to control the sensor stick using I2C.

The sensor explained in this section had an approximate cost of \$70. The low price of this sensor permits its usage in a lot of projects. Now, Sparkfun has upgraded it to a new 9DoF Razor IMU M0 with a cost around 50\$, you can check it in the link <https://www.sparkfun.com/products/14001>.

You can see the 9DoF Razor IMU sensor in the following image. It is small, and it has the following main components:

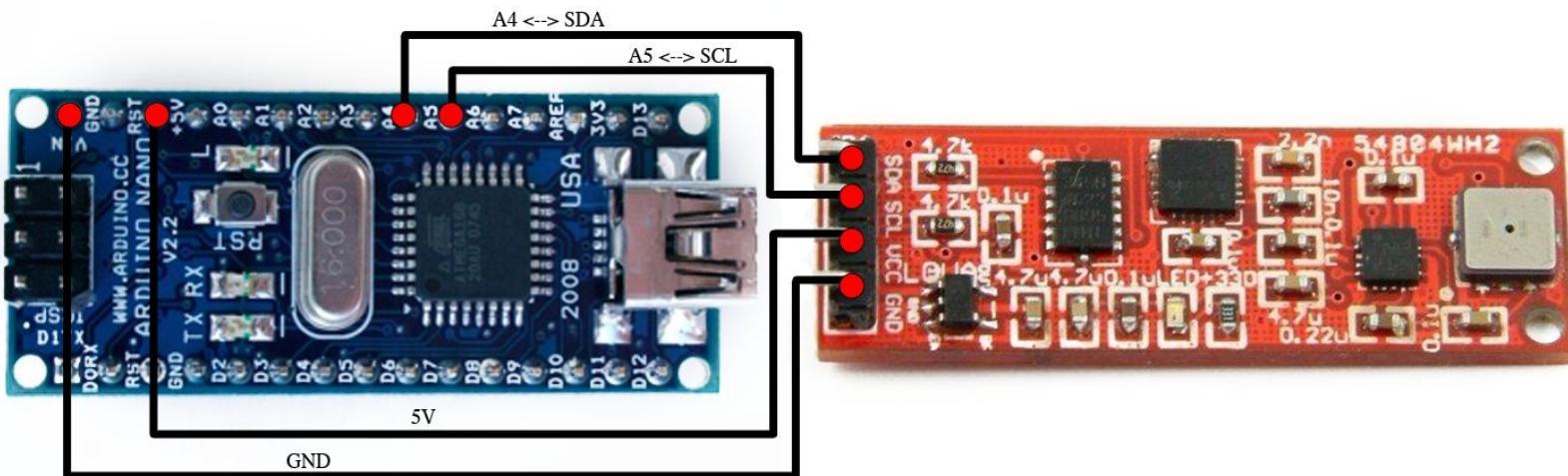


9DoF Razor IMU

This board has the following sensors:

- **ADXL345:** This is a three-axis accelerometer with a high resolution (13-bit) measurement of up to ± 16 g. This sensor is widely used in mobile device applications. It measures the static acceleration of gravity in tilt-sensing applications as well as the dynamic acceleration resulting from motion or shock.
- **HMC5883L:** This sensor is designed for low-field magnetic sensing with a digital interface for devices such as low-cost compasses and magnetometers.
- **BMP085:** This is a high-precision barometric pressure sensor used in advanced mobile applications. It offers superior performance with an absolute accuracy up to 0.03 hPa and has very low power consumption, 3 μ A.
- **L3G4200D:** This is a three-axis gyroscope with a very high resolution (16-bit) measurement of up to 2,000 **degrees per second (dps)**. This gyroscope measures how much the device is rotating around all three axes.

As we have said earlier, the sensor stick is controlled using the I2C protocol, and we will use Arduino to control it. In the following image, you can see the way to connect both the boards:



Arduino nano and sensor stick

For Arduino UNO, the pins connected are the same that for the Arduino Nano of the image. The only thing necessary to make it work is to connect the four wires. Connect **GND** and **VCC** from the sensor to **GND** and **5V** in Arduino.

The **Serial Data Line (SDL)** must be connected to the analog pin 4, and the **Serial Clock (SCK)** must be connected to the analog pin 5. If you connect these pins wrongly, Arduino will not be able to communicate with the sensor.

Installing Razor IMU ROS library

In order to install the Razor IMU library for ROS, we have to install visual python with the following command:

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

Then, we have to go to our workspace `/src` folder and clone from GitHub the `razor_imu_9dof` repository from *Kristof Robot* and compile it:

```
$ cd ~/dev/catkin_ws/src
$ git clone https://github.com/KristofRobot/razor_imu_9dof.git
$ cd ..
$ catkin_make
```

You need to have Arduino IDE installed; this is explained previously in the Arduino section of this chapter. Open your Arduino IDE-`/dev/catkin_ws/src/Razor_AHRS/Razor_AHRS.ino`. Now, you have to select your hardware options. Search in the code this section and uncomment the right line for you:

```
/********************* USER SETUP AREA! Set your options here! *****/
// HARDWARE OPTIONS
// Select your hardware here by uncommenting one line!
#ifndef HW__VERSION_CODE
#define HW__VERSION_CODE 10125 // SparkFun "9DOF Razor IMU"
version "SEN-10125" (HMC5843 magnetometer) #define HW__VERSION_CODE 10736 // SparkFun "9DOF Razor IMU"
version "SEN-10736" (HMC5883L magnetometer) //#define HW__VERSION_CODE 10183 // SparkFun "9DOF Sensor Stick"
version "SEN-10183" (HMC5843 magnetometer) //#define HW__VERSION_CODE 10321 // SparkFun "9DOF Sensor Stick"
version "SEN-10321" (HMC5843 magnetometer) //#define HW__VERSION_CODE 10724 // SparkFun "9DOF Sensor Stick"
version "SEN-10724" (HMC5883L magnetometer)
```

In my case, I'm using 9DoF Razor IMU version SEN-10736. Now, you can save and upload the code. Take care to choose the right processor in the Arduino IDE. For my razor IMU, an Arduino Pro or Pro Mini has to be chosen and an ATMega 3.3V 8 MHz must be selected as processor.

In the `razor_imu` package, you have to prepare a config file named `my_razor.yaml`. You can copy the default to `my_razor.yaml` configuration doing:

```
$ rosdep razor_imu_9dof/config
$ cp razor.yaml my_razor.yaml
```

You can open the file and check whether the configuration matches with yours. Now the most important thing is setting up your port correctly; check it whether you are using the sensor stick and your Arduino. In my case, it is by default:

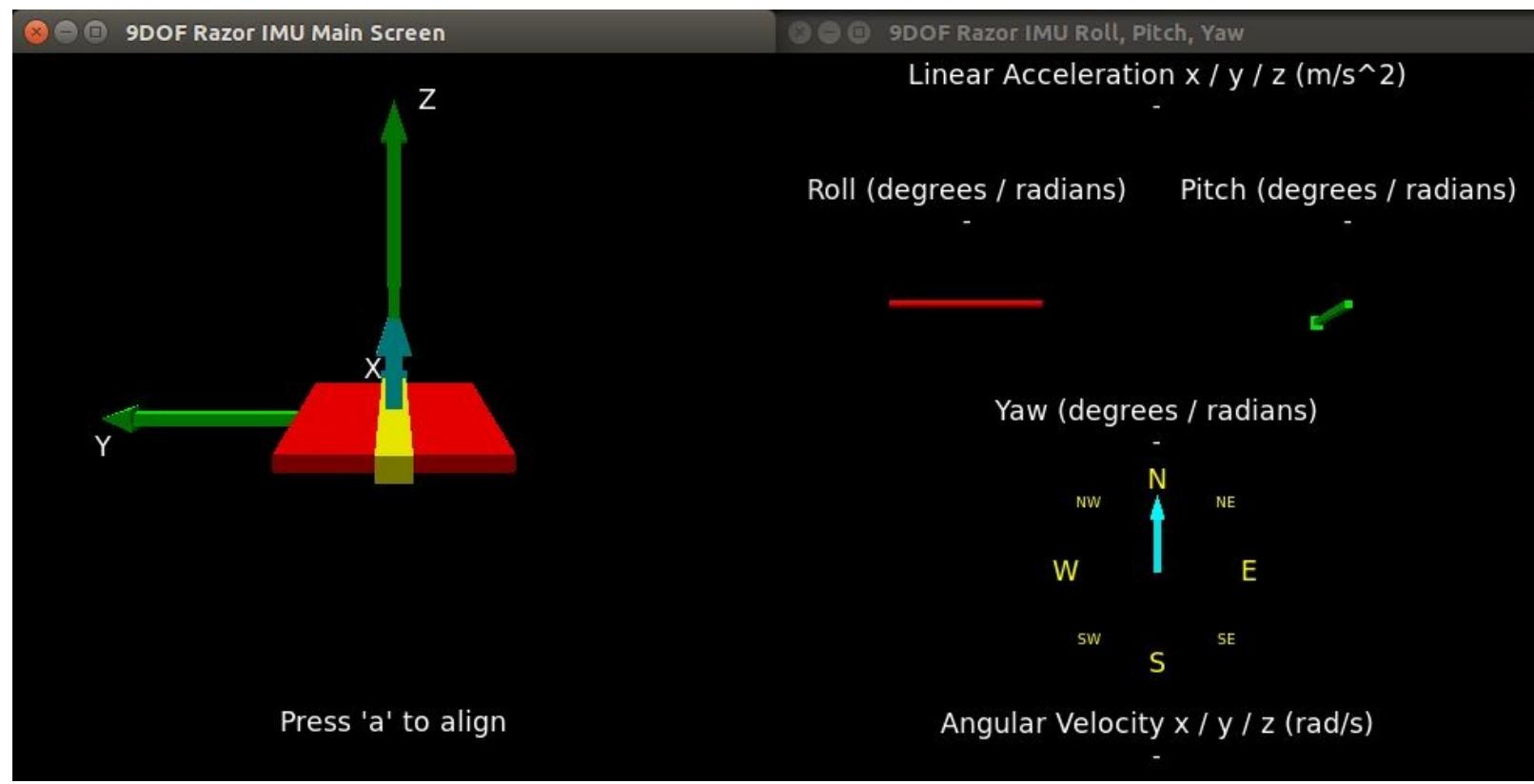
```
| port: /dev/ttyUSB0
```

Note that if you calibrate your IMU, you can include all the calibration parameter in this file to have a correct orientation measurement.

You can use now your `razor_imu` or stick with the following launch command:

```
$ roslaunch razor_imu_9dof razor-pub-and-display.launch
```

Two windows must appear, one with a 3D figure that represents the pose of the IMU and a 2D figure with the Roll, Pitch, and Yaw.



After some seconds, sensor `msgs::IMU` start to be transmitted and you will see how the parameters will change if you move your sensor.

SDOE Ratner IMU Main Screen

 SDCE Batzer IMU Roll, Pitch, Yaw



Press 'a' to align

Angular Velocity x / y / z (rad/s)
-0.0 / -0.09 / 0.01

```
212 #< /home/luis-devel/catkin_ws/src/razor_imu_9dof/launch/razor-pub-and-display.launch http://
213 #< magn_ellipsoid_center:[0.0000,0.0000,0.0000]
214 #< magn_ellipsoid_transform:[[0.0000000,0.0000000,0.0000000],[0.0000000,0.0000000,0.
215 #< magn_ellipsoid_transform:[0.0000000,0.0000000,0.0000000],[0.0000000,0.0000000,0.0000000]
216 #< gyro_offset_gyro_average_offset_x:0.00
217 #< gyro_offset_gyro_average_offset_y:0.00
218 #< gyro_offset_gyro_average_offset_z:0.00
219 #< gyro_offset_gyro_average_offset_x:0.00
220 #< gyro_offset_gyro_average_offset_y:0.00
221 #< gyro_offset_gyro_average_offset_z:0.00
222 [INFO] [WallTime: 1470778293.233547] Flushing first 200 IMU entries...
223 [INFO] [WallTime: 1470778297.350000] Publishing IMU data...
224
```

Razor IMU with different orientation

How does Razor send data in ROS?

If everything is fine, you can see the topic list using the `rostopic` command:

```
| $ rostopic list
```

The node will publish three topics. We will work with `/imu/data` in this section. First of all, we are going to see the type and the data sent by this topic. To see the type and the fields, use the following command lines:

```
| $ rostopic type /imu
| $ rosmsg show sensor_msgs/Imu
```

```
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
geometry_msgs/Quaternion orientation
  float64 x
  float64 y
  float64 z
  float64 w
float64[9] orientation_covariance
geometry_msgs/Vector3 angular_velocity
  float64 x
  float64 y
  float64 z
float64[9] angular_velocity_covariance
geometry_msgs/Vector3 linear_acceleration
  float64 x
  float64 y
  float64 z
float64[9] linear_acceleration_covariance
```

The `/imu` topic is `sensor_msgs/Imu`. The fields are used to indicate the orientation, acceleration, and velocity. In our example, we will use the orientation field.

Check a message to see a real example of the data sent. You can do it with the following command:

```
| $ rostopic echo /imu
```

You will see something similar to the following output:

```
header:
  seq: 43264
  stamp:
    secs: 1480621387
    nsecs: 926049947
  frame_id: base_imu_link
orientation:
  x: -0.664401936806
  y: 0.459286679427
  z: -0.562455021343
  w: 0.176833711254
orientation_covariance: [0.0025, 0.0, 0.0, 0.0, 0.0025, 0.0, 0.0, 0.0, 0.0025]
angular_velocity:
  x: -0.02
  y: -0.04
  z: -0.0
angular_velocity_covariance: [0.02, 0.0, 0.0, 0.0, 0.02, 0.0, 0.0, 0.0, 0.02]
linear_acceleration:
  x: 5.8836
  y: -7.60960921875
  z: -2.98087078125
linear_acceleration_covariance: [0.04, 0.0, 0.0, 0.0, 0.04, 0.0, 0.0, 0.0, 0.04]
---
```

If you observe the orientation field, you will see four variables instead of three, as you would probably expect. This is because in ROS, the spatial orientation is represented using quaternions. You can find a lot of literature on the Internet about this concise and unambiguous orientation representation.

Creating an ROS node to use data from the 9DoF sensor in our robot

Now that we have our low-cost IMU working in ROS, we are going to create a new node that subscribes to the `imu/data` topics from the Razor IMU. For example, we can see in our robot model the pitch, roll, and heading depending on the IMU data. So, let's create a new node based on our `c8_odom.cpp`. We will name this file `c8_odom_with_imu.cpp`.

Now, we are going to make some modifications. First, we will include `sensor_msgs/Imu` to be able to subscribe to IMU topics:

```
| #include <sensor_msgs/Imu.h>;
```

Also, we need a global variable to store IMU data:

```
| sensor_msgs::Imu imu;
```

Then, we are going to create a callback function to obtain IMU data:

```
| void imuCallback(const sensor_msgs::Imu &imu_msg)
| {
|     imu = imu_msg;
| }
```

In the `main` function, we need to declare a subscriber to `/imu_data` topic:

```
| ros::Subscriber imu_sub = n.subscribe("imu_data", 10,
| imuCallback);
```

And we have to assign each `odom` transform to the right IMU orientation data:

```
| odom_trans.transform.rotation.x = imu.orientation.x;
| odom_trans.transform.rotation.y = imu.orientation.y;
| odom_trans.transform.rotation.z = imu.orientation.z;
| odom_trans.transform.rotation.w = imu.orientation.w;
```

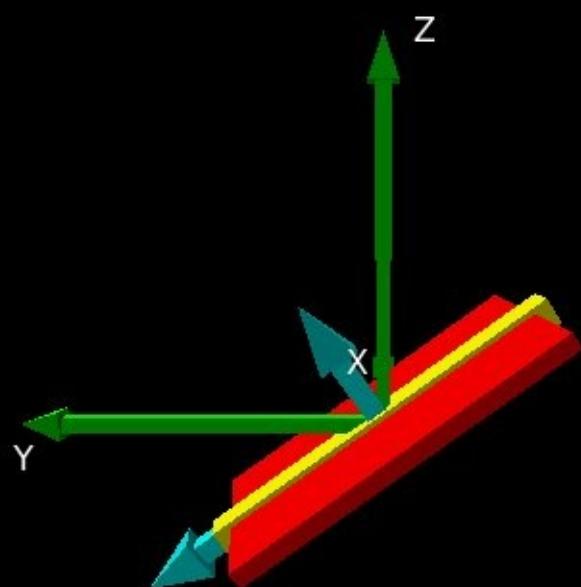
Also, we have to assign the pose of the robot to the orientation obtained by the IMU:

```
| odom.pose.pose.orientation.x = imu.orientation.x;
| odom.pose.pose.orientation.y = imu.orientation.y;
| odom.pose.pose.orientation.z = imu.orientation.z;
| odom.pose.pose.orientation.w = imu.orientation.w;
```

Once we have created this file and compile it, we are able to run this example:

```
| $ roslaunch razor_imu_9dof razor-pub-and-display.launch
| $ roslaunch chapter8_tutorials chapter8_robot_imu.launch
```

You will see the Razor IMU display and the robot model in the RViz visualizer. The robot model will have the pose and orientation of the Razor IMU.



Razor IMU display and robot model in Rviz

Using robot localization to fuse sensor data in your robot

Now, we have our robot and we have different sources that can be used to localize the robot such as the wheel encoders and the 9DoF Razor IMU. We are going to install a new package to combine the data from these sensors to improve our robot position estimation. The package `robot_localization` use an **Extended Kalman Filter (EKF)** to calculate the new estimation using data from multiple sensors.

To install this package, we will type the following command in a terminal:

```
$ sudo apt-get install ros-kinetic-robot-localization
```

Once we have installed this package, the next step is learning how to use it. So, we are going to explore inside the package, a launch file named `ekf_template.launch`:

```
$ rosed robot_localization ekf_template.launch
```

When the file is opened, we will see the following code:

```
<launch>
  <node pkg="robot_localization" type="ekf_localization_node"
    name="ekf_se" clear_params="true">      <rosparam command="load" file="$(find
      robot_localization)/params/ekf_template.yaml" />
    <!-- Placeholder for output topic remapping
    <remap from="odometry/filtered" to="" />
  -->
  </node>
</launch>
```

And we will see the code in order to launch the `ekf_localization_node` and load some parameters that are saved in `ekf_template.yaml`. The `yaml` file is similar to this format:

```
#Configuration for robot odometry EKF
#
frequency: 50

odom0: /odom
odom0_config: [false, false, false,
false, false, false,
true, true, true,
false, false, true,
false, false, false]
odom0_differential: false

imu0: /imu_data
imu0_config: [false, false, false,
false, false, true,
false, false, false,
false, false, true,
true, false, false]
imu0_differential: false

odom_frame: odom
base_link_frame: base_footprint
world_frame: odom
```

In this file, we define the name of our IMU topic, our odometry topic, also the `base_link_frame`. The EKF filter uses only data from the IMU and odometry topics that are selected by a 6 x 3 matrix filled with true or false.

We are going to save this `yaml` file in `chapter8_tutorials/config` with the name `robot_localization.yaml`, and we are going to create a new launch file in the launch folder of the chapter named `robot_localization.yaml`:

```
<launch>
  <node pkg="robot_localization" type="ekf_localization_node"
    name="ekf_localization">    <rosparam command="load" file="$(find
      chapter8_tutorials)/config/robot_localization.yaml" />
  </node>
</launch>
```

Now, we have finished the code of our robot and we can test all the parts:

```
$ roscore
$ rosrun rosserial_python serial_node.py /dev/ttyACM0
$ roslaunch razor_imu_9dof razor-pub-and-display.launch
$ roslaunch chapter8_tutorials chapter8_robot_encoders.launch
$ roslaunch chapter8_tutorials robot_localization
```

Using the IMU - Xsens MTi

In the following image, you can see the Xsens MTi, which is the sensor used in this section:



Xsens MtI

Xsens IMU is a typical inertial sensor that you can find in a robot. In this section, you will learn how to use it in ROS and how to use the topics published by the sensor.

You can use a lot of IMU devices with ROS such as the Razor IMU that we have used before. In this section, we will use the Xsens IMU, which is necessary to install the right drivers. But if you want to use MicroStrain 3DM-GX2 or Wiimote with Wii Motion Plus, you need to download the following drivers:



The drivers for the MicroStrain 3DM-GX2 IMU are available at http://www.ros.org/wiki/microstrain_3dmgx2_imu. The drivers for Wiimote with Wii Motion Plus are available at <http://www.ros.org/wiki/wiimote>.

To use our device, we are going to use `xsens_driver`. You can install it using the following command:

```
| $ sudo apt-get install ros-kinect-xsens-driver
```

Using the following commands, we also need to install two packages because the driver depends on them:

```
| $ rosstack profile  
| $ rospack profile
```

Now, we are going to start IMU and see how it works. In a shell, launch the following command:

```
| $ rosrun xsens_driver xsens_driver.launch
```

This driver detects the USB port and the baud rate directly without any changes.

How does Xsens send data in ROS?

If everything is fine, you can see the topic list by using the `rostopic` command:

```
$ rostopic list
```

The node will publish three topics. We will work with `/imu/data` in this section. First of all, we are going to see the type and the data sent by this topic. To see the type and the fields, use the following command lines:

```
$ rostopic type /imu/data
$ rostopic type /imu/data | rosmsg show
```

The `/imu/data` topic is `sensor_msgs/Imu`. The fields are used to indicate the orientation, acceleration, and velocity. In our example, we will use the orientation field. Check a message to see a real example of the data sent. You can do it with the following command:

```
$ rostopic echo /imu/data
```

You will see something similar to the following output:

```
...
header:
seq: 288
stamp:
secs: 1330631562
nsecs: 789304161
frame_id: xsens_mti_imu
orientation:
x: 0.00401890464127
y: -0.00402884092182
z: 0.679586052895
w: 0.73357373476
...
```

If you observe the orientation field, you will see four variables instead of three, as you would probably expect. This is because in ROS, the spatial orientation is represented using quaternions. You can find a lot of literature on the Internet about this concise and nonambiguous orientation representation.

We can observe the IMU orientation in the `rviz` run and add the `imu` display type:

```
$ rosrun rviz rviz
```

Using a GPS system

The **Global Positioning System (GPS)** is a space-based satellite system that provides information on the position and time for any weather and any place on the face of the earth and its vicinity. You must have an unobstructed direct path with four GPS satellites to obtain valid data.

The data received from the GPS conforms to the standards of communication set up by **National Maritime Electronics Association (NMEA)** and follows a protocol with different types of sentences. In them, we can find all the information about the position of the receiver.



To read more about all the types of NMEA messages, you can visit <http://www.gpsinformation.org/dale/nmea.htm>.

One of the most interesting pieces of information about GPS is contained in GGA sentences. They provide the current fix data with the 3D location of the GPS. An example of this sentence and an explanation of each field is given here:

```
$GPGLL,123519,4807.038,N,01131.000,E,1,08,0.9,545.4,M,46.9,M,,*47
Where:
GGA Global Positioning System Fix Data
123519 Fix taken at 12:35:19 UTC
4807.038,N Latitude 48 deg 07.038' N
01131.000,E Longitude 11 deg 31.000' E
1 Fix quality: 0 = invalid
1 = GPS fix (SPS)
2 = DGPS fix
3 = PPS fix
4 = Real Time Kinematic
5 = Float RTK
6 = estimated (dead reckoning) (2.3 feature)
7 = Manual input mode
8 = Simulation mode
08 Number of satellites being tracked
0.9 Horizontal dilution of position
545.4,M Altitude, Meters, above mean sea level
46.9,M Height of geoid (mean sea level) above WGS84
ellipsoid
(empty field) time in seconds since last DGPS update
(empty field) DGPS station ID number
*47 the checksum data, always begins with *
```

Depending on the GPS receiver, we can find different performances and precisions. We have a simple GPS at a low cost that is commonly used in different applications, such as UAV. They have an error that can be in the range of a few meters. Also, we can find expensive GPS devices that can be configured as differential GPS or can work in the **Real Time Kinematics (RTK)** mode, where a second GPS at a known location sends corrections to the first GPS. This GPS can achieve great results in terms of precision, with location errors less than 10 cm.

In general, GPS uses serial protocols to transmit the data received to a computer or a microcontroller, such as Arduino. We can find devices that use TTL or RS232, and they are easy to connect to the computer with a USB adapter. In this section, we will use a low-cost NEO-Bloc 6 M and a really accurate system, such as GR-3 Topcon in the RTK mode. We will see that, with the same drivers, we can obtain the

latitude, longitude, and altitude from both devices:



Em406a and Topcon GPS

In order to control a GPS sensor with ROS, we will install the NMEA GPS driver package by using the following command line (don't forget to run the `rosstack` and `rospack` profiles after that):

```
$ sudo apt-get install ros-kinetic-nmea-gps-driver  
$ rosstack profile & rospack profile
```

To execute the GPS driver, we will run the `nmea_gps_driver.py` file. To do that, we have to indicate two arguments: the port that is connected to the GPS and the baud rate:

```
$ rosrun nmea_gps_driver nmea_gps_driver.py _port:=/dev/ttyUSB0 _baud:=4800
```

In the case of the EM-406a GPS, the default baud rate is 4800 Hz as indicated in the preceding command line. For Topcon GR-3, the baud rate is higher; it's about 1,15,200 Hz. If we want to use it with ROS, we will modify the `_baud` argument, as shown in the following command:

```
$ rosrun nmea_gps_driver nmea_gps_driver.py _port:=/dev/ttyUSB0 _baud:=115200
```

How GPS sends messages

If everything is OK, we will see a topic named `/fix` in the topic list by typing this:

```
$ rostopic list
```

To know which kind of data we will use, we typed the `rostopic` command. The NMEA GPS driver uses the `sensor_msgs/NavSatFix` message to send the GPS status information:

```
$ rostopic type /fix
sensor_msgs/NavSatFix
```

The `/fix` topic is `sensor_msgs/NavSatFix`. The fields are used to indicate the latitude, longitude, altitude, status, quality of the service, and the covariance matrix. In our example, we will use the latitude and the longitude to project them to a 2D Cartesian coordinate system named **Universal Transverse Mercator (UTM)**.

Check a message to see a real example of the data sent. You can do it with the following command:

```
$ rostopic echo /fix
```

You will see something that looks similar to the following output:

```
...
header:
seq: 3
stamp:
secs: 1404993925
nsecs: 255094051
frame_id: /gps
status:
status: 0
service: 1
latitude: 28.0800916667
longitude: -15.451595
altitude: 315.3
position_covariance: [3.24, 0.0, 0.0, 0.0, 3.24, 0.0, 0.0, 0.0, 12.96]
position_covariance_type: 1
---
```

Creating an example project to use GPS

In this example, we are going to project the latitude and the longitude of GPS to a 2D Cartesian space. For this, we will use a function written by *Chuck Gantz* that converts latitudes and longitudes into UTM coordinates. The node will subscribe to the `/fix` topic where GPS data is sent. You can find the code in `chapter8_tutorials` in the `c8_fixtoUTM.cpp` file:

```
#include <ros/ros.h>
#include <tf/transform_broadcaster.h>
#include <nav_msgs/Odometry.h>
#include <stdio.h>
#include <iostream>
#include <sensor_msgs/NavSatFix.h>
geometry_msgs::Point global_position;
ros::Publisher position_pub;
void gpsCallBack(const sensor_msgs::NavSatFixConstPtr& gps)
{
    double northing, easting;
    char zone;
    LLtoUTM(gps->latitude, gps->longitude, northing, easting ,
             &zone);
    global_position.x = easting;
    global_position.y = northing;
    global_position.z = gps->altitude;
}
int main(int argc, char** argv){
    ros::init(argc, argv, "fixtoUTM");
    ros::NodeHandle n;
    ros::Subscriber gps_sub = n.subscribe("fix",10, gpsCallBack);
    position_pub = n.advertise<geometry_msgs::Point>
        ("global_position", 1);
    ros::Rate loop_rate(10);
    while(n.ok())
    {
        ros::spinOnce();
        loop_rate.sleep();
    }
}
```

First, you should declare the `NavSatFix` message using `#include <sensor_msgs/NavSatFix.h>`.

This way, we can subscribe to the `/fix` topic in the `ros::Subscriber gps_sub = n.subscribe("fix",10, gpsCallBack)` main function.

All the action happens in the `gpsCallBack()` function. We will use the `LLtoUTM()` function to make the conversion from latitudes and longitudes to the UTM space. We will publish a `geometry_msg/Point` topic named `/global_position` with the UTM northing and easting coordinates and the altitude from the GPS.

To try this code, after running the GPS driver, you can use the following command:

```
$ rosrun chapter8_tutorials c8_fixtoUTM
```

You can use GPS data to improve your robot localization using *Kalman* filters to fuse odometry and IMU with `NavSatFix` data.

Using a laser rangefinder - Hokuyo URG-04lx

In mobile robotics, it is very important to know where the obstacles are, the outline of a room, and so on. Robots use maps to navigate and move across unknown spaces. The sensor used for these purposes is LIDAR. This sensor is used to measure distances between the robot and objects.

In this section, you will learn how to use a low-cost version of LIDAR that is widely used in robotics. This sensor is the Hokuyo URG-04lx rangefinder. You can obtain more information about it at <http://www.hokuyo-aut.jp/>. The Hokuyo rangefinder is a device used to navigate and build maps in real time:



Hokuyo URG-04lx

The Hokuyo URG-04lx model is a low-cost rangefinder commonly used in robotics. It has a very good resolution and is very easy to use. To start with, we are going to install the drivers for the laser:

```
$ sudo apt-get install ros-kinetic-hokuyo-node  
$ rosstack profile && rospack profile
```

Once installed, we are going to check whether everything is OK. Connect your laser and check whether the system can detect it and whether it is configured correctly:

```
$ ls -l /dev/ttyACM0
```

When the laser is connected, the system sees it, so the result of the preceding command line is the following output:

```
crw-rw---- 1 root dialout 166, 0 Jan 13 11:09 /dev/ttyACM0
```

In our case, we need to reconfigure the laser device to give ROS the access to use it; that is, we need to

give the appropriate permissions:

```
| $ sudo chmod a+r /dev/ttyACM0
```

Check the reconfiguration with the following command line:

```
| $ ls -l /dev/ttyACM0  
crw-rw-rw- 1 root dialout 166, 0 Jan 13 11:09 /dev/ttyACM0
```

Once everything is OK, we are going to switch on the laser. Start `roscore` in one shell, and in another shell, execute the following command:

```
| $ rosrun hokuyo_node hokuyo_node
```

If everything is fine, you will see the following output:

```
| [ INFO] [1358076340.184643618]: Connected to device with ID: H1000484
```

Understanding how the laser sends data in ROS

To check whether the node is sending data, use `rostopic`, as shown here:

```
$ rostopic list
```

You will see the following topics as the output:

```
/diagnostics  
/hokuyo_node/parameter_descriptions  
/hokuyo_node/parameter_updates  
/rosout  
/rosout_agg  
/scan
```

The `/scan` topic is the topic where the node is publishing. The type of data used by the node is shown here:

```
$ rostopic type /scan
```

You will then see the message type used to send information about the laser:

```
sensor_msgs/LaserScan
```

You can see the structure of the message using the following command:

```
$ rosmsg show sensor_msgs/LaserScan
```

To learn a little bit more about how the laser works and what data it is sending, we are going to use the `rostopic` command to see a real message:

```
$ rostopic echo /scan
```

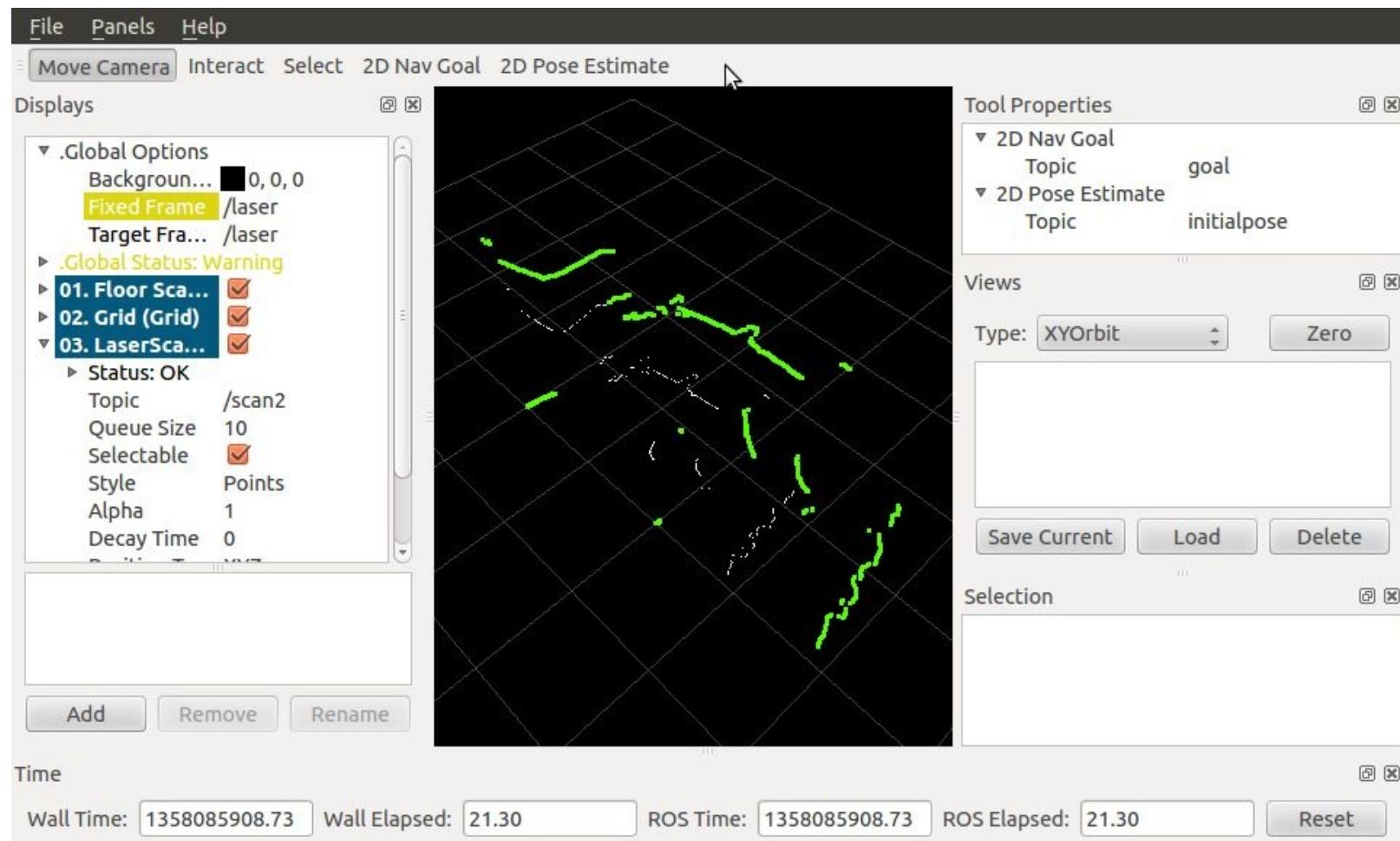
Then, you will see the following message sent by the laser:

```
---  
header:  
  seq: 3895  
  stamp:  
    secs: 1358076731  
    nsecs: 284896750  
    frame_id: laser  
...  
  ranges: [1.111999985559082, 1.111999985559082, 1.1109999418258667, ...]  
  intensities: []  
---
```

This data is difficult to understand for humans. If you want to see the data in a more friendly and graphical way, it is possible to do so using `rviz`. Type the following command line in a shell to launch `rviz` with the correct configuration file:

```
$ rosrun rviz rviz -d 'rospack find chapter8_tutorials'/config/laser.rviz
```

The following screenshot shows a graphical representation of the message:



You will see the contour on the screen. If you move the laser sensor, you will see the contour changing.

Accessing the laser data and modifying it

Now, we are going to make a node get the laser data, do something with it, and publish the new data. Perhaps, this will be useful at a later date, and with this example, you will learn how to do it.

Copy the following code snippet to the `c8_laserscan.cpp` file in your `/chapter8_tutorials/src` directory:

```
#include <ros/ros.h>
#include "std_msgs/String.h"
#include <sensor_msgs/LaserScan.h>

#include<stdio.h>
using namespace std;
class Scan2{
public:
Scan2();
private:
ros::NodeHandle n;
ros::Publisher scan_pub;
ros::Subscriber scan_sub;
void scanCallBack(const sensor_msgs::LaserScan::ConstPtr&
scan2);
};

Scan2::Scan2()
{
    scan_pub = n.advertise<sensor_msgs::LaserScan>("/scan2",1);
    scan_sub = n.subscribe<sensor_msgs::LaserScan>("/scan",1,
    &Scan2::scanCallBack, this);
}

void Scan2::scanCallBack(const sensor_msgs::LaserScan::ConstPtr&
scan2)
{
    int ranges = scan2->ranges.size();
    //populate the LaserScan message
    sensor_msgs::LaserScan scan;
    scan.header.stamp = scan2->header.stamp;
    scan.header.frame_id = scan2->header.frame_id;
    scan.angle_min = scan2->angle_min;
    scan.angle_max = scan2->angle_max;
    scan.angle_increment = scan2->angle_increment;
    scan.time_increment = scan2->time_increment;
    scan.range_min = 0.0;
    scan.range_max = 100.0;
    scan.ranges.resize(ranges);
    for(int i = 0; i < ranges; ++i)
    {
        scan.ranges[i] = scan2->ranges[i] + 1;
    }
    scan_pub.publish(scan);
}
int main(int argc, char** argv)
{
    ros::init(argc, argv, "laser_scan_publisher");
    Scan2 scan2;
    ros::spin();
}
```

We are going to break the code and see what it is doing.

In the `main` function, we initialize the node with the name `example2_laser_scan_publisher` and create an instance of the class that we have created in the file.

In the constructor, we will create two topics: one of them will subscribe to the other topic, which is the

original data from the laser. The second topic will publish the newly modified data from the laser.

This example is very simple; we are only going to add one unit to the data received from the laser topic and publish it again. We do that in the `scanCallBack()` function. Take the input message and copy all the fields to another variable. Then, take the field where the data is stored and add the one unit. Once the new value is stored, publish the new topic:

```
void Scan2::scanCallBack(const sensor_msgs::LaserScan::ConstPtr&
 scan2)
{
    ...
    sensor_msgs::LaserScan scan;
    scan.header.stamp = scan2->header.stamp;
    ...
    ...
    scan.range_max = 100.0;
    scan.ranges.resize(ranges);

    for(int i = 0; i < ranges; ++i){
        scan.ranges[i] = scan2->ranges[i] + 1;
    }

    scan_pub.publish(scan);
}
```

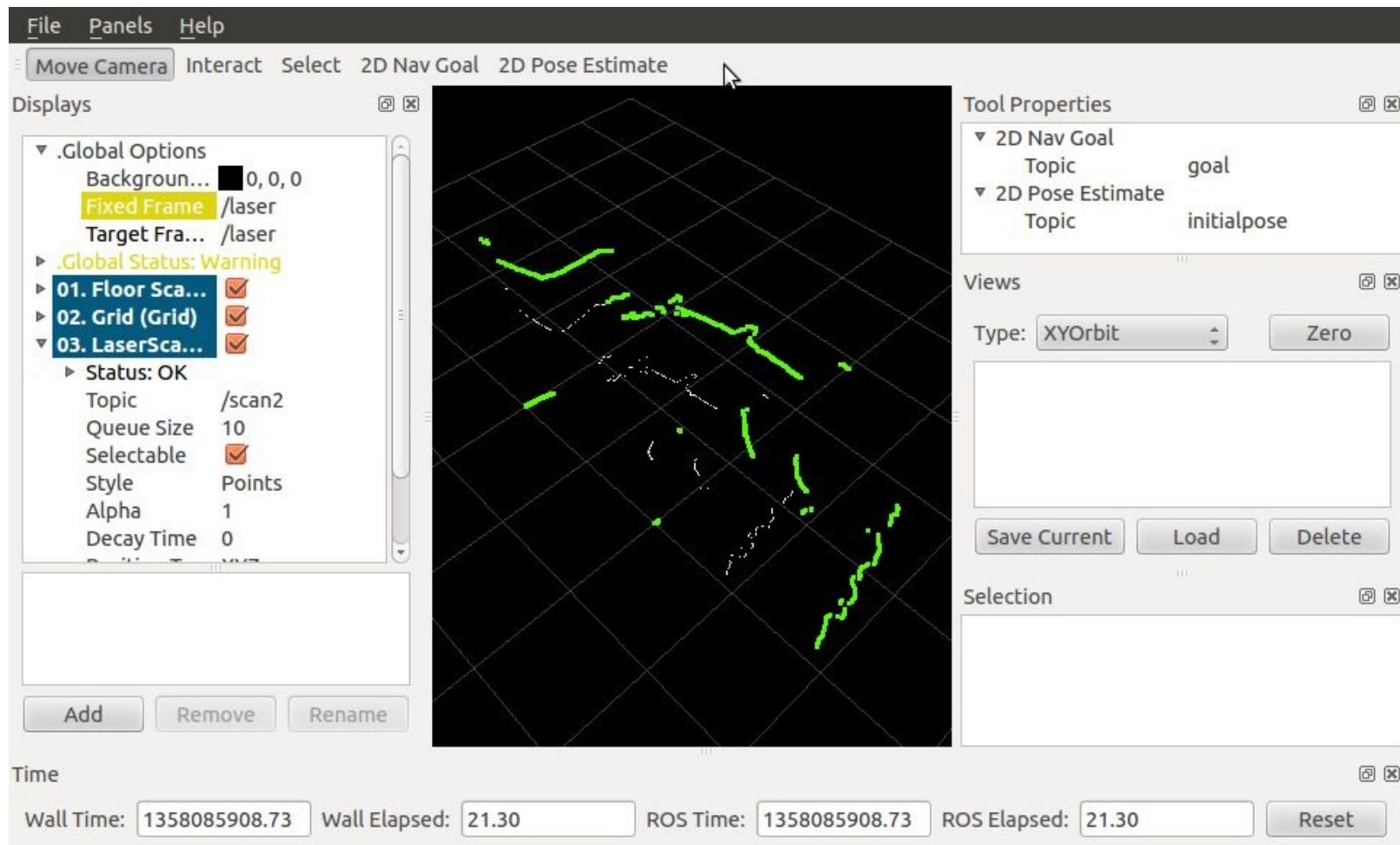
Creating a launch file

To launch everything, we are going to create a launch file, `chapter8_laserscan.launch`:

```
<launch>
  <node pkg="hokuyo_node" type="hokuyo_node" name="hokuyo_node"/>
  <node pkg="rviz" type="rviz" name="rviz"
    args="-d $(find chapter8_tutorials)/config/laser.rviz"/>

  <node pkg="chapter8_tutorials" type="c8_laserscan"
    name="c8_laserscan" />
</launch>
```

Now, if you launch the `chapter8_laserscan.launch` file, three nodes will start: `hokuyo_node`, `rviz`, and `c8_laserscan`. You will see the RViz visualizer screen with the two-laser contour. The green contour is the new data, as shown in the following screenshot:



Using the Kinect sensor to view objects in 3D

The Kinect sensor is a flat, black box that sits on a small platform when placed on a table or shelf near the television you're using with your Xbox 360. This device has the following three sensors that we can use for vision and robotics tasks:

- A color VGA video camera to see the world in color
- A depth sensor, which is an infrared projector and a monochrome CMOS sensor working together, to see objects in 3D
- A multiarray microphone that is used to isolate the voices of the players from the noise in the room.



KINECT™
for  XBOX 360.

Kinect

In ROS, we are going to use two of these sensors: the RGB camera and the depth sensor. In the latest version of ROS, you can even use three.

Before we start using it, we need to install the packages and drivers. Use the following command lines to install them:

```
$ sudo apt-get install ros-kinetic-openni-camera ros-kinetic-kinetic-openni-launch  
$ rosstack profile && rospack profile
```

Once the packages and drivers are installed, plug in the Kinect sensor, and we will run the nodes to start using it. In a shell, start `roscore`. In another shell, run the following command lines:

```
$ rosrun openni_camera openni_node  
$ roslaunch openni_launch openni.launch
```

If everything goes well, you will not see any error messages.

How does Kinect send data from the sensors, and how do we see it?

Now, we are going to see what we can do with these nodes. List the topics that you have created using this command:

```
$ rostopic list
```

Then, you will see a lot of topics, but the most important ones for us are the following:

```
...  
/camera/rgb/image_color  
/camera/rgb/image_mono  
/camera/rgb/image_raw  
/camera/rgb/image_rect  
/camera/rgb/image_rect_color  
...
```

We will see a lot of topics created by nodes. If you want to see one of the sensors, for example, the RGB camera, you can use the `/camera/rgb/image_color` topic. To see the image from the sensor, we are going to use the `image_view` package. Type the following command in a shell:

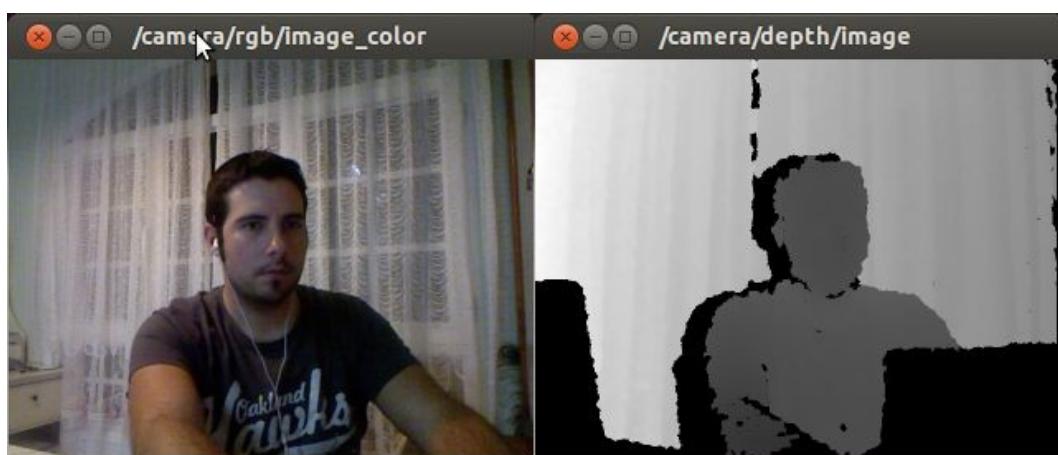
```
$ rosrun image_view image_view image:=/camera/rgb/image_color
```

Note that we need to rename (remap) the image topic to `/camera/rgb/image_color` using the parameter's `image`. If everything is fine, a new window appears that shows the image from Kinect.

If you want to see the depth sensor, you can do so just by changing the topic in the last command line:

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

You will then see an image similar to the following screenshot:



RGB and depth image from Kinect

Another important topic is the one that sends the point cloud data. This kind of data is a 3D representation of the depth image. You can find this data in `/camera/depth/points`, `/camera/depth_registered/points` and other topics.

We are going to see the type of message this is. To do this, use `rostopic type`. To see the fields of a message, we can use `rostopic type /topic_name | rosmsg show`. In this case, we are going to use the `/camera/depth/points` topic:

```
| $ rostopic type /camera/depth/points | rosmsg show
```

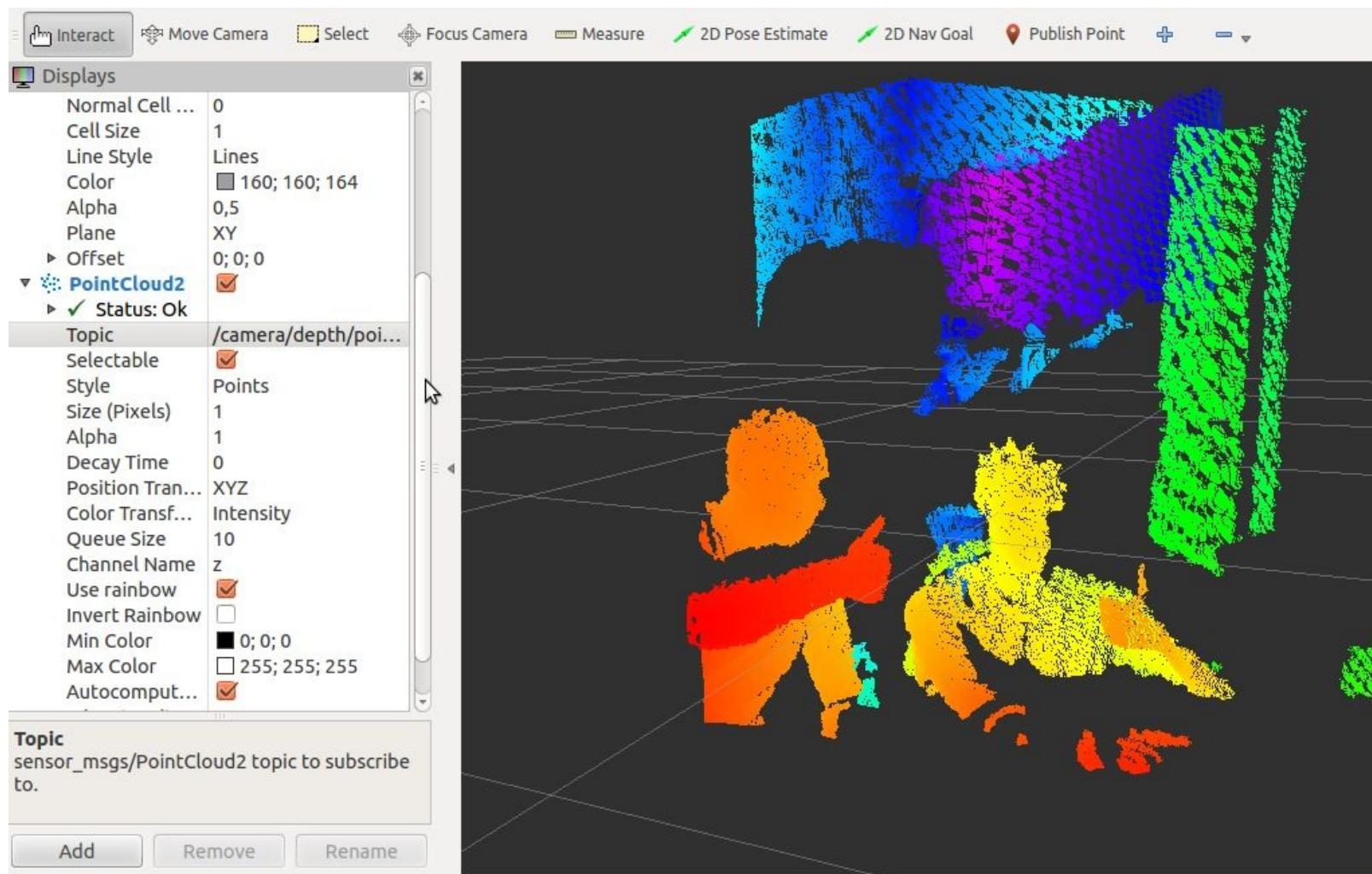
 *To see the official specification of the message, visit http://ros.org/doc/api/sensor_msgs/html/msg/PointCloud2.html.*

If you want to visualize this type of data, run `rviz` in a new shell and add a new `PointCloud2` data visualization, as shown here:

```
| $ rosrun rviz rviz
```

Click on Add, order topics by display type, and select `PointCloud2`. Once you have added a `PointCloud2` display type, you have to select the name of the `camera/depth/points` topic.

On your computer, you will see a 3D image in real time; if you move in front of the sensor, you will see yourself moving in 3D, as you can see in the following screenshot:



3D point cloud data from Kinect

Creating an example to use Kinect

Now, we are going to implement a program to generate a node that filters the point cloud from the Kinect sensor. This node will apply a filter to reduce the number of points in the original data. It will make a down sampling of the data.

Create a new file, `c8_kinect.cpp`, in your `chapter8_tutorials/src` directory and type in the following code snippet:

```
#include <ros/ros.h>
#include <sensor_msgs/PointCloud2.h>
// PCL specific includes
#include <pcl_conversions/pcl_conversions.h>
#include <pcl/point_cloud.h>
#include <pcl/point_types.h>
#include <pcl/filters/voxel_grid.h>

#include <pcl/io/pcd_io.h>

ros::Publisher pub;
void cloud_cb (const pcl::PCLPointCloud2ConstPtr& input)
{
    pcl::PCLPointCloud2 cloud_filtered;
    pcl::VoxelGrid<pcl::PCLPointCloud2> sor;
    sor.setInputCloud (input);
    sor.setLeafSize (0.01, 0.01, 0.01);
    sor.filter (cloud_filtered);
    // Publish the data
    pub.publish (cloud_filtered);
}

int main (int argc, char** argv)
{
    // Initialize ROS
    ros::init (argc, argv, "c8_kinect");
    ros::NodeHandle nh;
    // Create a ROS subscriber for the input point cloud
    ros::Subscriber sub = nh.subscribe ("/camera/depth/points", 1,
        cloud_cb);
    // Create a ROS publisher for the output point cloud
    pub = nh.advertise<sensor_msgs::PointCloud2> ("output", 1);
    // Spin
    ros::spin ();
}
```

 You can see it at http://pointclouds.org/documentation/tutorials/voxel_grid.php#voxelgrid.

This sample is based on the tutorial of Point Cloud Library (PCL).

All the work is done in the `cb()` function. This function is called when a message arrives. We create a `sor` variable with the `VoxelGrid` type, and the range of the grid is changed in `sor.setLeafSize()`. These values will change the grid used for the filter. If you increment the value, you will obtain less resolution and fewer points on the point cloud:

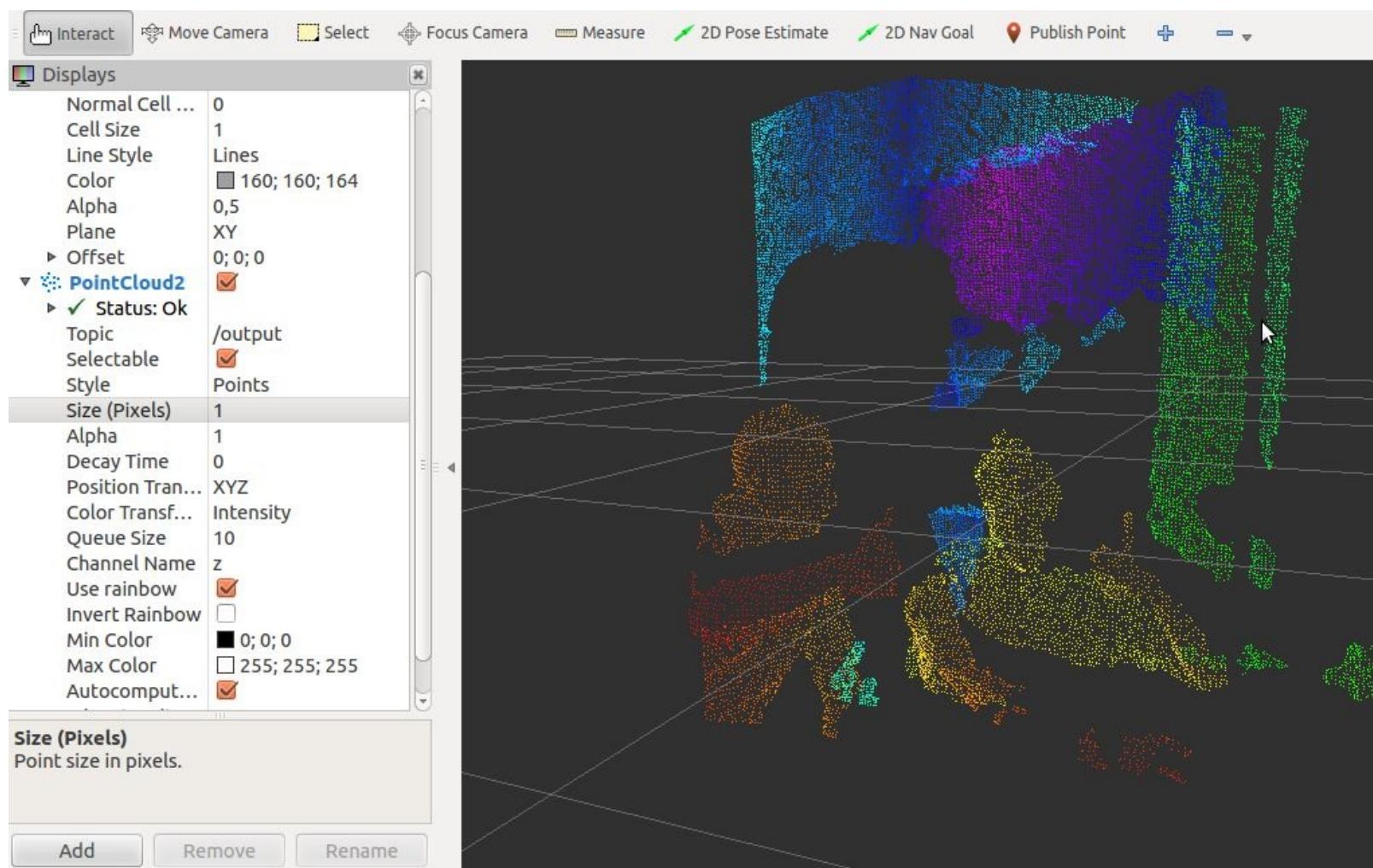
```
cloud_cb (const sensor_msgs::PointCloud2ConstPtr& input)
{
    ...
    pcl::VoxelGrid<sensor_msgs::PointCloud2> sor;
    ...
}
```

```

        sor.setLeafSize(0.01f, 0.01f, 0.01f);
    ...
}

```

If we open `rviz` now with the new node running, we will see the new point cloud in the window, and you will directly notice that the resolution is less than that of the original data, as shown in the following screenshot:



3D point cloud data after downsampling

On `rviz`, you can see the number of points that a message has. For original data, we can see that the number of points is 2,19,075. With the new point cloud, we obtain 16,981 points. As you can see, it is a huge reduction of data.

At <http://pointclouds.org/>, you will find more filters and tutorials on how you can use this kind of data.

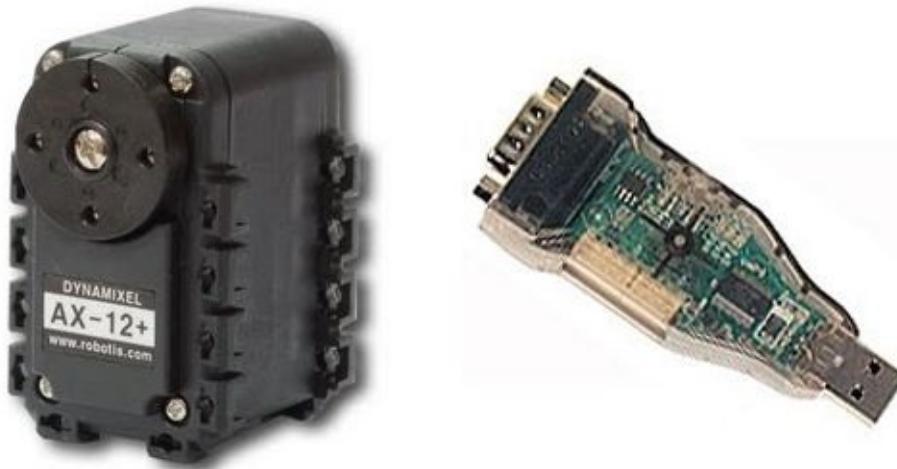
Using servomotors - Dynamixel

In mobile robots, servomotors are widely used. This kind of actuator is used to move sensors, wheels, and robotic arms. A low-cost solution is to use RC servomotors. It provides a movement range of 180 degrees and a high torque for the existing servomotors.

The servomotor that we will explain in this section is a new type of servomotor designed and used for robotics. This is the **Dynamixel servomotor**.

Dynamixel is a lineup, high-performance, networked actuator for robots developed by *ROBOTIS*, a Korean manufacturer. ROBOTIS is also the developer and manufacturer of OLLO, Bioloid, and DARwIn-OP DXL. These robots are used by numerous companies, universities, and hobbyists due to their versatile expansion capability, powerful feedback functions, position, speed, internal temperature, input voltage, and their simple daisy chain topology for simplified wiring connections.

In the following image, you can see Dynamixel AX-12 and the USB interface. Both are used in this example.



Dynamixel AX-12+ motor

First, we are going to install the necessary packages and drivers. Type the following command line in a shell:

```
$ sudo apt-get install ros-kinetic-dynamixel-motor  
$ rosstack profile && rospack profile
```

Once the necessary packages and drivers are installed, connect the dongle to the computer and check whether it is detected. Normally, it will create a new port with the name `ttyUSBX` inside your `/dev/` folder. If you see this port, everything is OK, and now we can let the nodes play a little with the servomotor.

In a shell, start `roscore`, and in another shell, type the following command line:

```
$ rosrun dynamixel_tutorials controller_manager.launch
```

If the motors are connected, you will see the motors detected by the driver. In our case, a motor with the ID 6 is detected and configured:

```
process[dynamixel_manager-1]: started with pid [3966]
[INFO] [WallTime: 1359377042.681841] pan_tilt_port: Pinging motor IDs 1 through 25...
[INFO] [WallTime: 1359377044.846779] pan_tilt_port: Found 1 motors - 1 AX-12 [6], initialization complete.
```

How does Dynamixel send and receive commands for the movements?

Once you have launched the `controller_manager.launch` file, you will see a list of topics. Remember to use the following command line to see these topics:

```
$ rostopic list
```

These topics will show the state of the motors configured, as follows:

```
/diagnostics  
/motor_states/pan_tilt_port  
/rosout  
/rosout_agg
```

If you see `/motor_states/pan_tilt_port` with the `rostopic echo` command, you will see the state of all the motors, which, in our case, is only the motor with the ID 6; however, we cannot move the motors with these topics, so we need to run the next launch file to do it.

This launch file will create the necessary topics to move the motors, as follows:

```
$ rosrun dynamixel_tutorials controller_spawner.launch
```

The topic list will have two new topics added to the list. One of the new topics will be used to move the servomotor, as follows:

```
/diagnostics  
/motor_states/pan_tilt_port  
/rosout  
/rosout_agg  
/tilt_controller/command  
/tilt_controller/state
```

To move the motor, we are going to use the `/tilt_controller/command` that will publish a topic with the `rostopic pub` command. First, you need to see the fields of the topic and the type. To do this, use the following command lines:

```
$ rostopic type /tilt_controller/command  
std_msgs/Float64
```

As you can see, it is a `Float64` variable. This variable is used to move the motor to a position measured in radians. So, to publish a topic, use the following commands:

```
$ rostopic pub /tilt_controller/command std_msgs/Float64 -- 0.5
```

Once the command is executed, you will see the motor moving, and it will stop at 0.5 radians or

28.6478898 degrees.

Creating an example to use the servomotor

Now, we are going to show you how you can move the motor using a node. Create a new file, `c8_dynamixel.cpp`, in your `/c_tutorials/src` directory with the following code snippet:

```
#include<ros/ros.h>
#include<std_msgs/Float64.h>
#include<stdio.h>

using namespace std;

class Dynamixel{
private:
ros::NodeHandle n;
ros::Publisher pub_n;
public:
Dynamixel();
int moveMotor(double position);
};

Dynamixel::Dynamixel(){
pub_n = n.advertise<std_msgs::Float64>
("/tilt_controller/command",1);
}
int Dynamixel::moveMotor(double position)
{
std_msgs::Float64 aux;
aux.data = position;
pub_n.publish(aux);
return 1;
}

int main(int argc,char** argv)
{
ros::init(argc, argv, "c8_dynamixel");
Dynamixel motors;

float counter = -180;
ros::Rate loop_rate(100);
while(ros::ok())
{
if(counter < 180)
{
motors.moveMotor(counter*3.14/180);
counter++;
}else{
counter = -180;
}
loop_rate.sleep();
}
}
```

This node will move the motor continuously from `-180` to `180` degrees. It is a simple example, but you can use it to make complex movements or control more motors. We assume that you understand the code and that it is not necessary to explain it. Note that you are publishing data to the `/tilt_controller/command` topic; this is the name of the motor.

Summary

The use of sensors and actuators in robotics is very important since this is the only way to interact with the real world. In this chapter, you learned how to use, configure, and investigate further how certain common sensors and actuators work, which are used by a number of people in the world of robotics. We are sure that if you wish to use another type of sensor, you will find information on the Internet and in the ROS documentation about how to use it without problems.

In our opinion, Arduino is a very interesting device because you can build your own robots, add more devices and cheap sensors to your computer with it, and use them within the ROS framework easily and transparently. Arduino has a large community, and you can find information on many sensors, which cover the spectrum of applications you can imagine.

Finally, we must mention that the range laser will be a very useful sensor for the navigation algorithms as you saw in the simulation chapters. The reason is that it is a mandatory device to implement the navigation stack, which relies on the range readings it provides at a high frequency and with good precision.

Computer Vision

ROS provides basic support for **Computer Vision**. First, drivers are available for different cameras and protocols, especially for FireWire (IEEE1394a or IEEE1394b) cameras. An image pipeline helps with the camera calibration process, distortion rectification, color decoding, and other low-level operations. For more complex tasks, you can use **OpenCV** and the `cv_bridge` and `image_transport` libraries to interface with it and subscribe and publish images on topics. Finally, there are several packages that implement algorithms for object recognition, augmented reality, visual odometry, and so on.

Although FireWire cameras are best integrated in ROS, it is not difficult to support other protocols, such as USB and Gigabit Ethernet. Since USB cameras are usually less expensive and easier to find, in this chapter we discuss several available options, and we will also provide a driver that integrates seamlessly in the image pipeline, using the OpenCV video capture API.

The camera calibration and the result integration in the image pipeline will be explained in detail. ROS provides GUIs to help with the camera calibration process using a calibration pattern. Furthermore, we will cover stereo cameras and explain how we can manage rigs of two or more cameras, with more complex setups than a binocular camera. Stereo vision will also let us obtain depth information from the world, up to a certain extent and depending on certain conditions. Hence, we will also see how to inspect that information as point clouds and how to improve its quality to the best possible extent for our camera's quality and its setup.

We will also explain the **ROS image pipeline**, which simplifies the process of converting the RAW images acquired by the camera into monochrome (grayscale) and color images; this sometimes requires you to *debayer* the RAW images if they are codified as a Bayer pattern. If the camera has been calibrated, the calibration information is used to rectify the images, that is, to correct the distortion.

For stereo images, since we have the baseline the left and right cameras, we can compute the disparity image, which allows us to obtain depth information and a 3D point cloud once it has been fine-tuned; here, we will also give you tuning advice, as this can be quite difficult for low-quality cameras that sometimes require good calibration results beforehand. Finally, by using OpenCV inside ROS, even though it's only version 2.x (version 3.x is not yet supported), we have the ability to implement a wide range of Computer Vision and machine learning algorithms, or we can even run some algorithms or examples already present in this library. However, we will not cover the OpenCV API, which is outside the scope of this section. We advise the reader to check the online documentation (<http://docs.opencv.org>) or any book about OpenCV and Computer Vision. We will also simply show you how you can use OpenCV in your nodes, with examples of feature detection, descriptor extraction, and matching to compute the homography between two images. Additionally, this chapter will finish with a tutorial that will show you how to set up and run a visual odometry implementation integrated into ROS, the `viso2_ros` wrapper of the `libviso2` visual odometry library, using a stereo pair built with two cheap webcams attached to a supporting bar. Other visual odometry libraries will be mentioned, for example, `fovvis`, along with some advice on how to start working with them and how to improve the results with RGBD sensors (such as Kinect), sensor fusion, and additional information on monocular vision.

ROS camera drivers support

The different camera drivers available and the different ways to use cameras on ROS are explained in the following sections. In essence, they distinguish between FireWire and USB cameras.

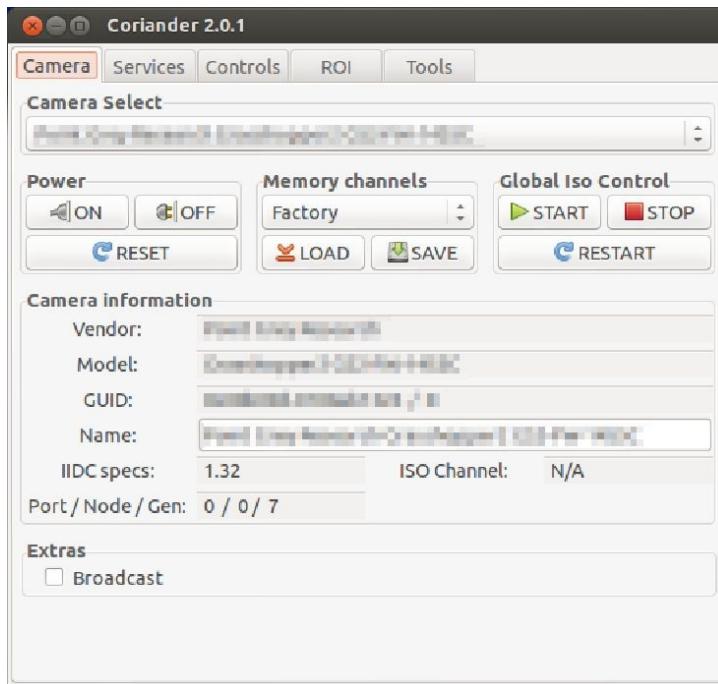
The first few steps that we must perform are connecting the camera to the computer, running the driver, and seeing the images it acquires in ROS. Before we get into ROS, it is always a good idea to use external tools to check that the camera is actually recognized by our system, which, in our case, is an Ubuntu distribution. We will start with FireWire cameras since they are better supported in ROS, and later we will look at USB cameras.

FireWire IEEE1394 cameras

Connect your camera to the computer, which should have a FireWire IEEE1394a or IEEE1394b slot. Then, in Ubuntu, you only need `coriander` to check that the camera is recognized and working. If it is not already installed, just install `coriander`. Then, run it (in old Ubuntu distributions, you may have to run it as `sudo`):

```
| $ coriander
```

It will automatically detect all your FireWire cameras, as shown in the following screenshot:



The great thing about `coriander` is that it also allows us to view the image and configure the camera. Indeed, our advice is to use the `coriander` package's camera configuration interface and then take those values into ROS, as we will see later. The advantage of this approach is that `coriander` gives us the dimensional values of some parameters. In ROS, there are certain parameters that sometimes fail to set, that is, gamma, and they may need to be set beforehand in `coriander` as a workaround.

Now that we know that the camera is working, we can close `coriander` and run the ROS FireWire camera driver (with `roscore` running). The camera driver package can be installed with:

```
| $ sudo apt-get install ros-kinetic-camera1394
```

If it is not available yet on ROS Kinetic, build it from source on a workspace. It can be obtained from <https://github.com/ros-drivers/camera1394>; simply clone the repository into a workspace and build it.

The camera driver is run with:

```
| $ rosrun camera1394 camera1394_node
```

Simply run `roscore` and the previous command. It will start the first camera on the bus, but note that you can

select the camera by its GUID, which you can see in the `coriander` package's GUI.

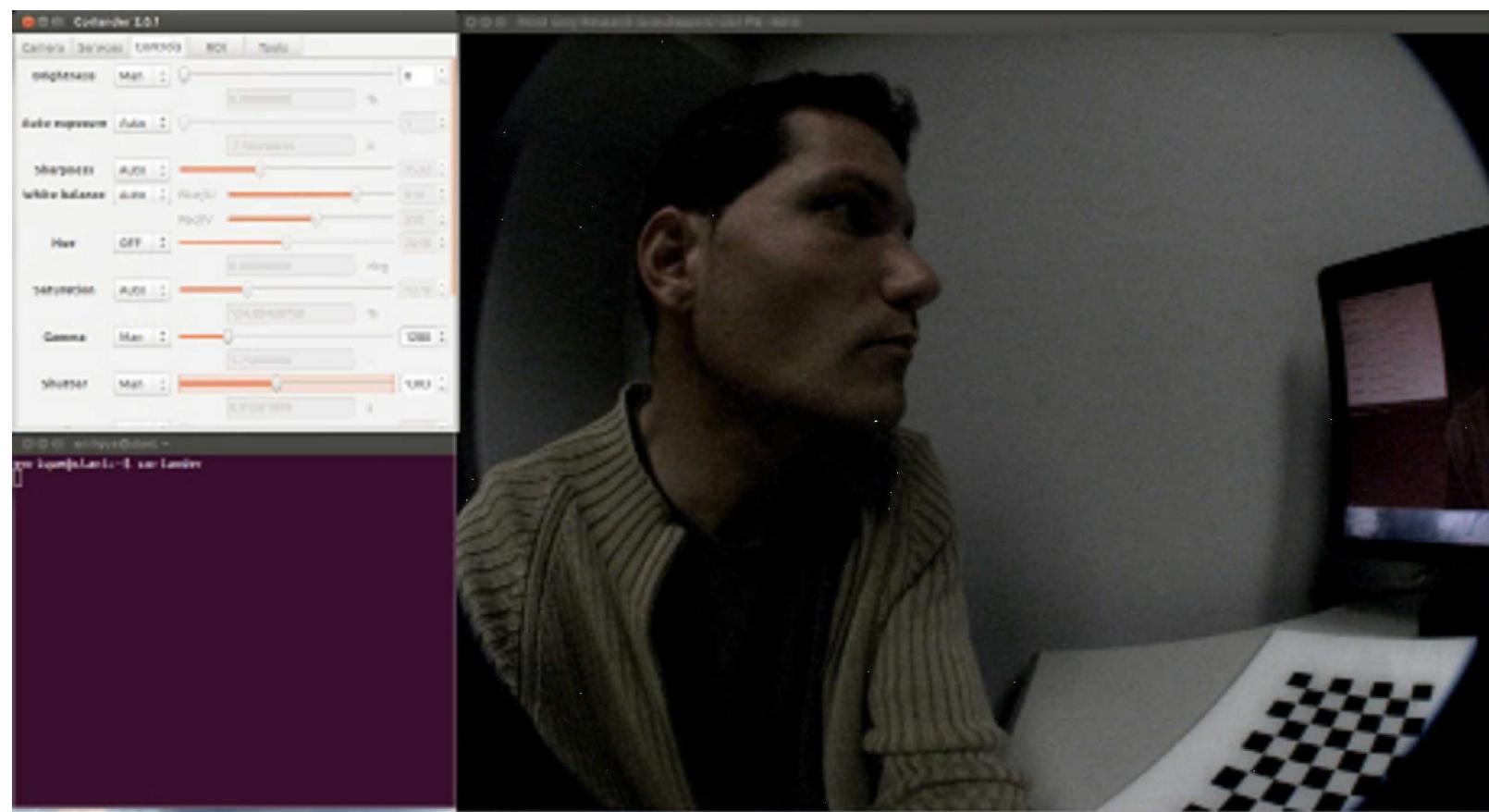
The FireWire camera's supported parameters are listed and assigned sensible values in the `camera1394/config/firewire_camera/format7_mode0.yaml` file, as shown in the following code:

```
guid: 00b09d0100ab1324 # (defaults to first camera on bus)
iso_speed: 800 # IEEE1394b
video_mode: format7_mode0 # 1384x1036 @ 30fpsbayer pattern
# Note that frame_rate is overwritten by frame_rate_feature; some
useful values:
# 21fps (480)
frame_rate: 21 # max fps (Hz)
auto_frame_rate_feature: 3 # Manual (3)
frame_rate_feature: 480
format7_color_coding: raw8 # for bayer
bayer_pattern: rggb
bayer_method: HQ
auto_brightness: 3 # Manual (3)
brightness: 0
auto_exposure: 3 # Manual (3)
exposure: 350
auto_gain: 3 # Manual (3)
gain: 700
# We cannot set gamma manually in ROS, so we switch it off
auto_gamma: 0 # Off (0)
#gamma: 1024 # gamma 1
auto_saturation: 3 # Manual (3)
saturation: 1000
auto_sharpness: 3 # Manual (3)
sharpness: 1000
auto_shutter: 3 # Manual (3)
#shutter: 1000 # = 10ms
shutter: 1512 # = 20ms (1/50Hz), max. in30fps
auto_white_balance: 3 # Manual (3)
white_balance_BU: 820
white_balance_RV: 520
frame_id: firewire_camera
camera_info_url:
  package://chapter5_tutorials/calibration/firewire_camera/
  calibration_firewire_camera.yaml
```

The values must be tuned by watching the images acquired, for example in `coriander`, and setting the values that give better images. The `guid` parameter is used to select the camera, which is a unique value. You should set the shutter speed to a frequency equal to, or a multiple of, the electric light you have in the room to avoid flickering. If outside in sunlight, you only have to worry about setting a value that gives you an appropriate amount of light. You can use a high gain, but it will introduce noise. However, in general, it is better to have salt-and-pepper noise such as that of a low shutter speed (to receive most light), because with a low shutter speed, we have motion blur and most algorithms perform badly with it. As you can see, the configuration depends on the lighting conditions of the environment, and you may have to adapt the configuration to them.

That is quite easy using `coriander` or the `rqt_reconfigure` interface (see the following screenshots and the upcoming code, for instance):

```
| $ rosrun rqt_reconfigure rqt_reconfigure /camera
| $ coriander
```



In order to better understand how to properly set the parameters of the camera to obtain high-quality images, which are also algorithm-friendly, you are encouraged to find out more about the basic concepts of photography, such as the exposure triangle, which is a combination of shutter speed, ISO, and aperture:



Here, the camera's namespace is `/camera`. Then, we can change all the parameters that are specified in the `camera1394` dynamic reconfigure .cfg file, as shown in [Chapter 3, Visualization and Debugging Tools](#). Here,

for your convenience, you can create a launch file, which is also in `launch/firewire_camera.launch`:

```
&lt;launch>
  &lt;!-- Arguments -->
  &lt;!-- Show video output (both RAW and rectified) -->
  &lt;arg name="view" default="false" />
  &lt;!-- Camera params (config) -->
  &lt;arg name="params" default="$(find chapter5_tutorials)/
    config/firewire_camera/format7_mode0.yaml" />
  &lt;!-- Camera driver -->
  &lt;node pkg="camera1394" type="camera1394_node" name="camera1394_node">
    &lt;rosparam file="$(arg params)" />
  &lt;/node>
  &lt;!-- Camera image processing (color + rectification) -->
  &lt;node ns="camera" pkg="image_proc" type="image_proc"
    name="image_proc" />
  &lt;!-- Show video output -->
  &lt;group if="$(arg view)">
    &lt;!-- Image viewer (non-rectified image) -->
    &lt;node pkg="image_view" type="image_view"
      name="non_rectified_image">
      &lt;remap from="image" to="camera/image_color" />
    &lt;/node>
    &lt;!-- Image viewer (rectified image) -->
    &lt;node pkg="image_view" type="image_view"
      name="rectified_image">
      &lt;remap from="image" to="camera/image_rect_color" />
    &lt;/node>
  &lt;/group>
&lt;/launch>
```

The `camera1394` driver is started with the parameters shown so far. It also runs the image pipeline that we will see in the sequel in order to obtain the color-rectified images using the **Debayer algorithm** and the calibration parameters (once the camera has been calibrated). We have a conditional group to visualize the color and color-rectified images using `image_view` (or `rqt_image_view`).

In sum, in order to run a FireWire camera in ROS and view the images, once you have set its GUID in the parameters file, simply run the following command:

```
| $ rosrun chapter5_tutorials firewire_camera.launch view:=true
```

Then, you can also configure it dynamically with `rqt_reconfigure`.

USB cameras

Now we are going to do the same thing with USB cameras. The only problem is that, surprisingly, they are not inherently supported by ROS. First of all, once you have connected the camera to the computer, test it with a chat or video meeting program, for example, Skype or Cheese. The camera resource should appear in `/dev/video?`, where ? should be a number starting with 0 (that may be your internal webcam if you are using a laptop).

There are two main options that deserve to be mentioned as possible USB camera drivers for ROS. First, we have `usb_cam`. To install it, use the following command:

```
| $ sudo apt-get install ros-kinetic-usb-cam
```

Then, run the following command:

```
| $ roslaunch chapter5_tutorials usb_cam.launch view:=true
```

It simply does `rosrun usb_cam usb_cam_node` and shows the camera images with `image_view` (or `rqt_image_view`), so you should see something similar to the following screenshot. It has the RAW image of the USB camera, which is already in color:



Similarly, another good option is `gscam`, which is installed as follows:

```
| $ sudo apt-get install ros-kinetic-gscam
```

Then, run the following command:

```
| $ roslaunch chapter5_tutorials gscam.launch view:=true
```

As for `usb_cam`, this launch file performs a `rosrun gscam` and also sets the camera's parameters. It also visualizes the camera's images with `image_view` (or `rqt_image_view`), as shown in the following screenshot:



The parameters required by `gscam` are as follows (see `config/gscam/logitech.yaml`):

```
gscam_config: v4l2src device=/dev/video0 !video/x-rawrgb,  
framerate=30/1 ! ffmpegcolorspace  
frame_id: gscam  
camera_info_url:  
  package://chapter5_tutorials/calibration/gscam/  
  calibration_gscam.yaml
```

The `gscam_config` parameter invokes the `v4l2src` command with the appropriate arguments to run the camera. The rest of the parameters will be useful once the camera is calibrated and used in the ROS image pipeline.

Making your own USB camera driver with OpenCV

Although we have the two previous options, this section comes with its own USB camera driver implemented on top of OpenCV using the `cv::VideoCapture` class. It runs the camera and also allows you to change some of its parameters as long as they are supported by the camera's firmware. It also allows us to set the calibration information in the same way as with the FireWire cameras. With `usb_cam`, this is not possible because the `cameraInfo` message is not available. With regards to `gscam`, we have more control; we can change the camera configuration and see how to publish the camera's images and information in ROS. In order to implement a camera driver using OpenCV, we have two options on how we read images from the camera. First, we can poll with a target **Frames Per Second (FPS)**; secondly, we can set a timer for the period of such FPS and, in the timer callback, we perform the actual reading. Depending on the FPS, one solution may be better than the other in terms of CPU consumption. Note that polling does not block anything, as the OpenCV reading function waits until an image is ready; meanwhile, other processes take the CPU. In general, for fast FPS, it is better to use polling so that we do not incur the time penalty of using the timer and its callback. For low FPS, the timer should be similar to polling and the code should be cleaner. We invite the reader to compare both implementations in the `src/camera_polling.cpp` and `src/camera_timer.cpp` files. For the sake of space, here we will only show you the timer-based approach. Indeed, the final driver in `src/camera.cpp` uses a timer. Note that the final driver also includes camera information management, which we will see in the sequel.

In the package, we must set the dependency with OpenCV, the ROS image message libraries, and related. They are the following packages:

```
<depend package="sensor_msgs"/>
<depend package="opencv2"/>
<depend package="cv_bridge"/>
<depend package="image_transport"/>
```

Consequently, in `src/camera_timer.cpp`, we have the following headers:

```
#include <ros/ros.h>
#include <image_transport/image_transport.h>
#include <cv_bridge/cv_bridge.h>
#include <sensor_msgs/image_encodings.h>
#include <opencv2/highgui/highgui.hpp>
```

The `image_transport` API allows the publishing of images using several transport formats seamlessly, which can be compressed images, with different codecs, based on the plugins installed in the ROS system, for example, `compressed` and `theora`. The `cv_bridge` is used to convert OpenCV images to ROS image messages, for which we may need the image encoding of `sensor_msgs` in the case of grayscale/color conversion. Finally, we need the `highgui` API of OpenCV (`opencv2`) in order to use `cv::VideoCapture`.

Here, we will explain the main parts of the code in `src/camera_timer.cpp`, which has a class that implements the camera driver. Its attributes are as follows:

```
ros::NodeHandle nh;
```

```

image_transport::ImageTransport it;
image_transport::Publisher pub_image_raw;

cv::VideoCapture camera;
cv::Mat image;
cv_bridge::CvImagePtr frame;

ros::Timer timer;

int camera_index;
int fps;

```

As usual, we need the node handle. Then, we need `ImageTransport`, which is used to send the images in all available formats in a seamless way. In the code, we only need to use `Publisher` (only one), but note that it must be a specialization of the `image_transport` library.

Then, we have the OpenCV stuff to capture images/frames. In the case of the frame, we directly use the `cv_bridge` frame, which is `CvImagePtr`, because we can access the `image` field it has.

Finally, we have the timer and the basic camera parameters for the driver to work. This is the most basic driver possible. These parameters are the camera index, that is, the number for the `/dev/video?` device, for example `0` for `/dev/video0`; the camera index is then passed to `cv::VideoCapture`. The `fps` parameter sets the camera FPS (if possible) and the timer. Here, we use an `int` value, but it will be a double in the final version, `src/camera.cpp`.

The driver uses the class constructor for the setup or initialization of the node, the camera, and the timer:

```

nh.param<int>( "camera_index", camera_index, DEFAULT_CAMERA_INDEX
);

if ( not camera.isOpened() )
{
    ROS_ERROR_STREAM( "Failed to open camera device!" );
    ros::shutdown();
}

nh.param<int>( "fps", fps, DEFAULT_FPS );
ros::Duration period = ros::Duration( 1. / fps );

pub_image_raw = it.advertise( "image_raw", 1 );

frame = boost::make_shared<cv_bridge::CvImage>();
frame->encoding = sensor_msgs::image_encodings::BGR8;

timer = nh.createTimer( period, &CameraDriver::capture, this );

```

First, we open the camera and abort it if it does not open. Note that we must do this in the attribute constructor, shown as follows, where `camera_index` is passed by the parameter:

```
| camera(camera_index )
```

Then, we read the `fps` parameter and compute the timer `period`, which is used to create the timer and set the capture callback at the end. We advertise the image publisher using the image transport API, for `image_raw` (RAW images), and initialize the `frame` variable.

The `capture` callback method reads and publishes images as follows:

```

camera>> frame->image;
if( not frame->image.empty() )
```

```

} {
    frame->header.stamp = ros::Time::now();
    pub_image_raw.publish( frame->toImageMsg() );
}

```

The preceding method captures the images, checks whether a frame was actually captured, and in that case sets the timestamp and publishes the image, which is converted to a ROS image.

You can run this node with the following command:

```
| $ rosrun chapter9_tutorials camera_timer _camera_index:=0 _fps:=15
```

This will open the `/dev/video0` camera at 15 FPS.

Then, you can use `image_viewer rqt_image_view` to see the images. Similarly, for the polling implementation, you have the following command:

```
| $ roslaunch chapter5_tutorials camera_polling.launch
  camera_index:=0 fps:=15 view:=true
```

With the previous command, you will see the `/camera/image_raw` topic images.

For the timer implementation, we also have the `camera.launch` file, which runs the final version and provides more options, which we will see throughout this entire chapter. The main contributions of the final version support for dynamic reconfiguration parameters and the provision of camera information, which includes camera calibration. We are going to show you how to do this in brief, and we advise that you look at the source code for a more detailed understanding.

As with the FireWire cameras, we can give support for the dynamic reconfiguration of the camera's parameters. However, most USB cameras do not support changing certain parameters. What we do is expose all OpenCV supported parameters and warn in case of error (or disable a few of them). The configuration file is in `cfg/camera.cfg`; check it for details. It supports the following parameters:

- `camera_index`: This parameter is used to select the `/dev/video?` device
- `frame_width` and `frame_height`: These parameters give the image resolution
- `fps`: This parameter sets the camera FPS
- `fourcc`: This parameter specifies the camera pixel format in the *FOURCC* format (<http://www.fourcc.org>); the file format is typically *YUYV* or *MJPEG*, but they fail to change in most USB cameras with OpenCV
- `brightness`, `contrast`, `saturation`, and `hue`: These parameters set the camera's properties; in digital cameras, this is done by software during the acquisition process in the sensor or simply on the resulting image
- `gain`: This parameter sets the gain of the **Analog-to-Digital Converter (ADC)** of the sensor; it introduces salt-and-pepper noise into the image but increases the lightness in dark environments
- `exposure`: This parameter sets the lightness of the images, usually by adapting the gain and shutter speed (in low-cost cameras, this is simply the integration time of the light that enters the sensor)
- `frame_id`: This parameter is the camera frame and is useful if we use it for navigation, as we will see in the *Using visual odometry with viso2* section
- `camera_info_url`: This parameter provides the path to the camera's information, which is basically its

calibration

Then, in the following line of code in the driver, we use a dynamic reconfigure server:

```
| #include <dynamic_reconfigure/server.h>
```

We set a callback in the constructor:

```
| server.setCallback( boost::bind( &CameraDriver::reconfig, this,
| _1, _2 ) );
```

The `setCallback` constructor reconfigures the camera. We even allow it to change the camera and stop the current one when the `camera_index` changes. Then, we use the OpenCV `cv::VideoCapture` class to set the camera's properties, which are part of the parameters shown in the preceding line. As an example, in the case of `frame_width`, we use the following commands:

```
| newconfig.frame_width = setProperty( camera,
| CV_CAP_PROP_FRAME_WIDTH, newconfig.frame_width );
```

This relies on a private method named `setProperty`, which calls the `set` method of `cv::VideoCapture` and controls the cases in which it fails to print a ROS warning message. Note that the FPS has changed in the timer itself and cannot usually be modified in the camera. Finally, it is important to note that all this reconfiguration is done within a locked mutex to avoid acquiring any images while reconfiguring the driver.

In order to set the camera's information, ROS has a `camera_info_manager` library that helps us to do so, as shown in the following line:

```
| #include <camera_info_manager/camera_info_manager.h>
```

We use the library to obtain the `CameraInfo` message. In the `capture` callback of the timer, we use `image_transport::CameraPublisher`, and not only for images. The code is as follows:

```
| camera>> frame->image;
| if( not frame->image.empty() )
| {
|   frame->header.stamp = ros::Time::now();
|
|   *camera_info = camera_info_manager.getCameraInfo();
|   camera_info->header = frame->header;
|
|   camera_pub.publish( frame->toImageMsg(), camera_info );
| }
```

This is run within the mutex mentioned previously for the reconfiguration method. We do the same as for the first version of the driver but also retrieve the camera information from the manager, which is set with the node handler, the camera name, and `camera_info_url` in the reconfiguration method (which is always called once on loading). Then, we publish both the image/frame (ROS image) and the `CameraImage` messages.

In order to use this driver, use the following command:

```
| $ roslaunch chapter5_tutorials camera.launch view:=true
```

The command will use the `config/camera/webcam.yaml` parameter as default, which sets all the dynamic reconfiguration parameters seen so far.

You can check that the camera is working with `rostopic list` and `rostopic hz/camera/image_raw`; you can also check with `image_view` or `rqt_image_view`.

With the implementation of this driver, we have used all the resources available in ROS to work with cameras, images, and Computer Vision. In the following sections, for the sake of clarity, we will explain each of them separately.

ROS images

ROS provides the `sensor_msgs::Image` message to send images between nodes. However, we usually need a data type or object to manipulate the images in order to do some useful work. The most common library for that is OpenCV, so ROS offers a bridge class to transform ROS images back and forth from OpenCV.

If we have an OpenCV image, that is, `cv::Mat` image, we need the `cv_bridge` library to convert it into a ROS image message and publish it. We have the option to share or copy the image with `cvshare` or `cvcopy`, respectively. However, if possible, it is easier to use the OpenCV image field inside the `CvImage` class provided by `cv_bridge`. That is exactly what we do in the camera driver as a pointer:

```
| cv_bridge::CvImagePtr frame;
```

Being a pointer, we initialize it in the following way:

```
| frame = boost::make_shared<cv_bridge::CvImage>();
```

If we know the image encoding beforehand, we can use the following code:

```
| frame->encoding = sensor_msgs::image_encodings::BGR8;
```

Later, we set the OpenCV image at some point, for example, capturing it from a camera:

```
| camera>> frame->image;
```

It is also common to set the timestamp of the message at this point:

```
| frame->header.stamp = ros::Time::now();
```

Now we only have to publish it. To do so, we need a publisher and it must use the `image_transport` API of ROS. This is shown in the following section.

Publishing images with ImageTransport

We can publish single images with `ros::Publisher`, but it is better to use an `image_transport` publisher. It can publish images with their corresponding camera information. That is exactly what we did for the camera driver previously. The `image_transport` API is useful for providing different transport formats in a seamless way. The images you publish actually appear in several topics. Apart from the basic, uncompressed one, you will see a compressed one or even more. The number of supported transports depends on the plugins installed in your system; you will usually have the `compressed` and `theora` transports.

You can see this with `rostopic info`. In order to install all the plugins, use the following command:

```
| $ sudo apt-get install ros-kinetic-image-transport-plugins
```

In your code, you need the node handle to create the image transport and then the publisher. In this example, we will use a simple image publisher; please check the final USB camera driver for the `CameraPublisher` usage:

```
| ros::NodeHandle nh;
| image_transport::ImageTransport it;
| image_transport::Publisher pub_image_raw;
```

The node handle and the image transport are constructed with (in the attribute constructors of a class) the following code:

```
| nh( "~" ),
| it( nh )
```

For an `image_raw` topic, the publisher is created this way within the node namespace:

```
| pub_image_raw = it.advertise( "image_raw", 1 );
```

Hence, now the frame shown in the previous section can be published with the following code:

```
| pub_image_raw.publish( frame->toImageMsg() );
```

OpenCV in ROS

ROS provides a very simple integration with OpenCV, the most widely used open source Computer Vision library. However, it does not ship a specific Debian for ROS Kinetic because that would force users to use a single version. Instead of that, it allows you to use the latest OpenCV library on the system and provides additional integration tools to use OpenCV in ROS. In the following sections, we will explain how to install OpenCV and those additional tools.

Installing OpenCV 3.0

In ROS Kinetic we can start using OpenCV 3.0, in contrast to previous versions where some packages had some dependencies on OpenCV 2.x or compatibility issues with 3.0.

The installation follows the standard workflow of installing Ubuntu packages, so you only have to do the following:

```
| $ sudo apt-get install libopencv-dev
```

Alternatively, you can also install a ROS package that installs the library too:

```
| $ sudo apt-get install ros-kinetic-opencv3
```

Using OpenCV in ROS

ROS uses the standalone OpenCV library installed on your system. However, you must specify a build and run dependency with an `opencv2` package in the `package.xml` file:

```
| &lt;build_depend>opencv2&lt;/build_depend>
| &lt;run_depend>opencv2&lt;/run_depend>
```

In `CMakeLists.xml`, we have to insert the following lines:

```
| find_package(OpenCV)
| include_directories(${catkin_INCLUDE_DIRS} ${OpenCV_INCLUDE_DIRS})
```

Then, for each library or executable that uses OpenCV, we must add `${OpenCV_LIBS}` to `target_link_libraries` (see `CMakeLists.txt` provided for the `chapter5_tutorials` package).

In our node `.cpp` file, we include any of the OpenCV libraries we need. For example, for `highgui.hpp`, we use the following line:

```
| #include &lt;opencv2/highgui/highgui.hpp>
```

Now you can use any of the OpenCV API classes, functions, and so on in your code, as usual. Simply use its `cv` namespace and follow any OpenCV tutorial if you are starting with it. Note that this section is not about OpenCV-just how we can do Computer Vision inside ROS.

Visualizing the camera input images with rqt_image_view

In [Chapter 3, Visualization and Debugging Tools](#), we explained how to visualize any image published in the ROS framework by using the `image_view` node of the `image_view` package or `rqt_image_view`. The following code encapsulates this discussion:

```
| $ rosrun image_view image_view image:=/camera/image_raw
```

What is important here is the fact that by using the image transport, we can select different topics for viewing images using compressed formats if required. Also, in the case of stereo vision, as we will see later, we can use `rqt rviz` to see the point cloud obtained with the disparity image.

Camera calibration

Most cameras, especially wide-angle ones, exhibit large distortions. We can model such distortions as radial or tangential and compute the coefficients of that model using calibration algorithms. The camera calibration algorithms also obtain a calibration matrix that contains the focal distance and principle point of the lens and, hence, provide a way to measure distances in the world using the images acquired. In the case of stereo vision, it is also possible to retrieve depth information, that is, the distance of the pixels to the camera, as we will see later. Consequently, we have 3D information of the world up to a certain extent.

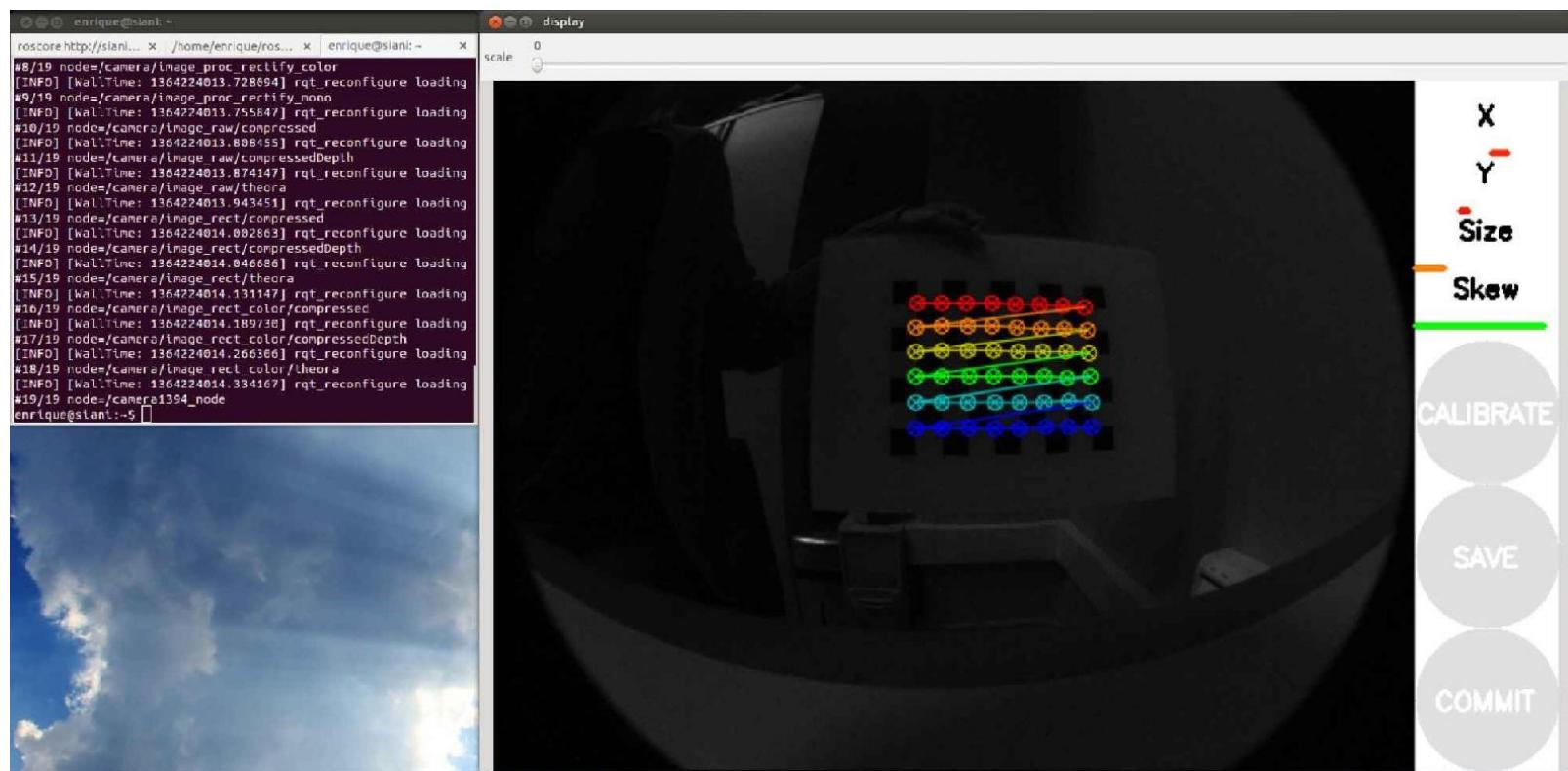
The calibration is done by showing several views of a known image called a **calibration pattern**, which is typically a chessboard/checkerboard. It can also be an array of circles or an asymmetric pattern of circles; note that circles are seen as ellipses by the camera for skew views. A detection algorithm obtains the inner corner point of the cells in the chessboard and uses them to estimate the camera's intrinsic and extrinsic parameters. In brief, the extrinsic parameters are the pose of the camera or, in other words, the pose of the pattern with respect to the camera if we left the camera in a fixed position. What we want are the intrinsic parameters because they do not change, can be used later for the camera in any pose, allow the measuring of distances in the images, and allow correcting the image distortion, that is, rectifying the image.

How to calibrate a camera

With our camera driver running, we can use the calibration tool of ROS to calibrate it. It is important that the camera driver provides `cameraInfo` messages and has the `camera_info_set` service, which allows you to set the path to the calibration results file. Later, this calibration information is loaded by the image pipeline when using the camera. One camera driver that satisfies these prerequisites is the `camera1394` driver for the FireWire cameras. In order to calibrate your FireWire camera, use the following command:

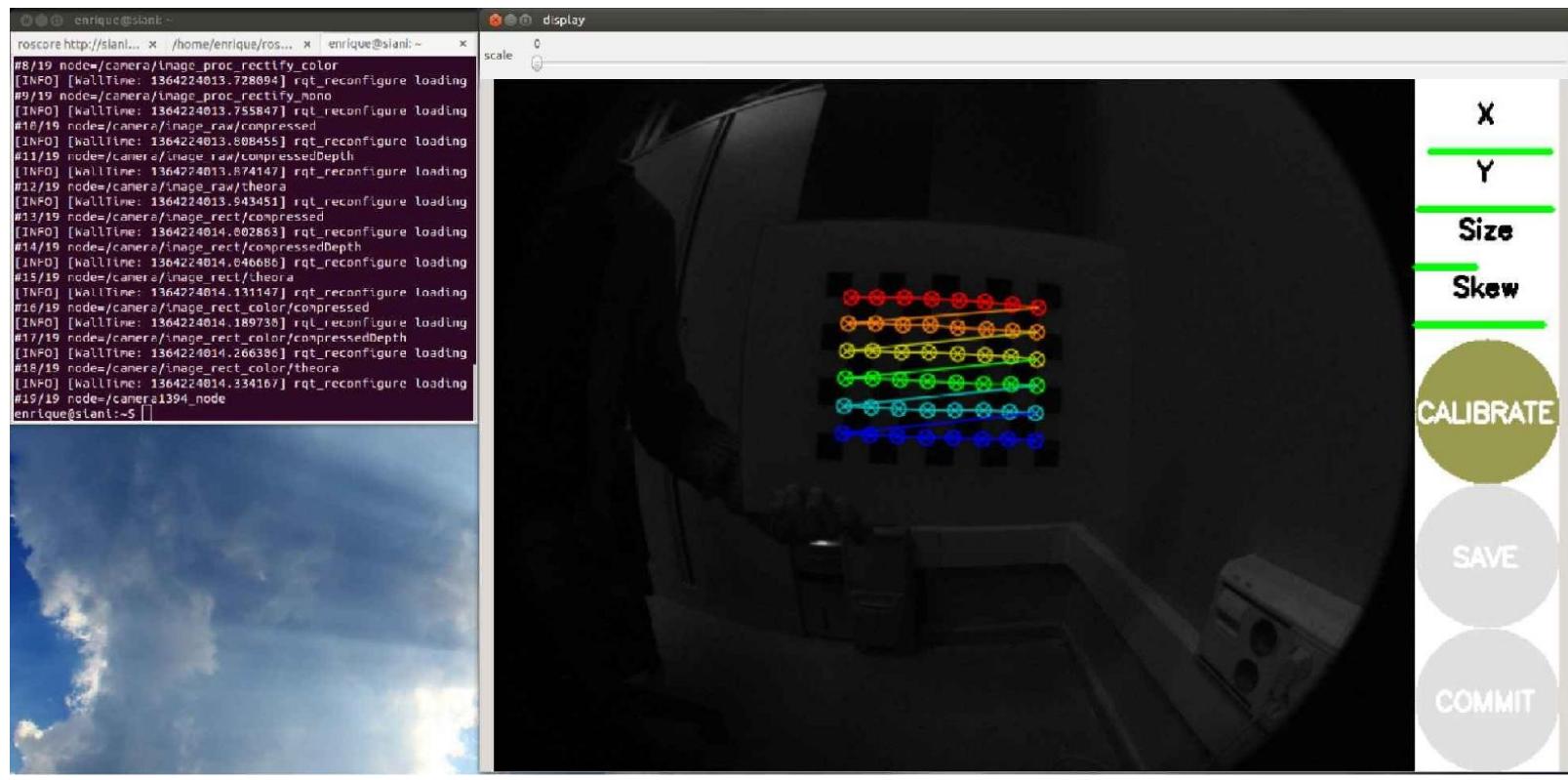
```
| $ roslaunch chapter5_tutorials calibration_firewire_camera_chessboard.launch
```

This will open a GUI that automatically selects the views of our calibration pattern and provides bars to inform users how each axis is covered by the views retrieved. It comprises the *x* and *y* axes, meaning how close the pattern has been shown to each extreme of these axes in the image plane, that is, the horizontal and vertical axes, respectively. Then, the scale goes from close to far (up to the distance at which the detection works). Finally, skew requires that views of the pattern tilt in both the *x* and *y* axes. The following three buttons these bars are disabled by default, as shown in the following screenshot:



You will see the points detected overlaid over the pattern every time the detector finds them. The views are automatically selected to cover a representative number of different views, so you must show views to make the bars become green from one side to the other, following the instructions given in the following section. In theory, two views are enough, but in practice around ten are usually needed. In fact, this interface captures even more (30 to 40). You should avoid fast movements because blurry images are bad for detection. Once the tool has enough views, it will allow you to calibrate, that is, to start the optimizer which, given the points detected in the calibration pattern views, solves the system of the pinhole camera model.

This is shown in the following screenshot:



Then, you can save the calibration data and commit the calibration results to the camera, that is, it uses the `camera_info_set` service to commit the calibration to the camera, so later it is detected automatically by the ROS image pipeline.

The launch file provided for the calibration simply uses `cameracalibrator.py` of the `camera_calibration` package:

```
<node pkg="camera_calibration" type="cameracalibrator.py"
  name="cameracalibrator" args="--size 8x6 --square 0.030"
  output="screen">
  <remap from="image" to="camera/image_colour" />
  <remap from="camera" to="camera" />
</node>
```

The calibration tool only needs the pattern's characteristics (the number of squares and their size, `--size 8x6` and `--square 0.030` in this case), the `image` topic, and the `camera` namespace.

The launch file also runs the image pipeline, but it is not required. In fact, instead of the `image_color` topic, we could have used the `image_raw` one.

Once you have saved the calibration (Save button), a file is created in your `/tmp` directory. It contains the calibration pattern views used for the calibration. You can find it at `/tmp/calibrationdata.tar.gz`; the ones used for calibration in the section can be found in the `calibration` directory and the `firewire_camera` subfolder for the FireWire camera. Similarly, on the terminal (`stdout` output), you will see information regarding the views taken and the calibration results. The ones obtained for the section are in the same folder as the calibration data. The calibration results can also be consulted in the `ost.txt` file inside the `calibrationdata.tar.gz` ZIP file. Remember that after the commit, the calibration file is updated with the calibration matrix and the coefficients of the distortion model. A good way to do so consists of creating a dummy calibration file before the calibration. In our package, that file is in `calibration/firewire_camera/calibration_firewire_camera.yaml`, which is referenced by the parameters file:

```
camera_info_url:  
package://chapter5_tutorials/calibration/firewire_camera/calibrati  
on_firewire_camera.yaml
```

Now, we can use our camera again with the image pipeline, and the rectified images will have the distortion corrected as a clear sign that the camera is calibrated correctly. Since ROS uses the Zhang calibration method implemented in OpenCV, our advice is that you consult its documentation at http://docs.opencv.org/doc/tutorials/calib3d/camera_calibration/camera_calibration.html.

Finally, you can also play with different calibration patterns using the following launch files for circles and asymmetric circles (https://raw.githubusercontent.com/opencv/opencv/05b15943d6a42c99e5f921b7dbaa8323f3c042c6/doc/acircles_pattern.png), prepared for FireWire cameras, as an example:

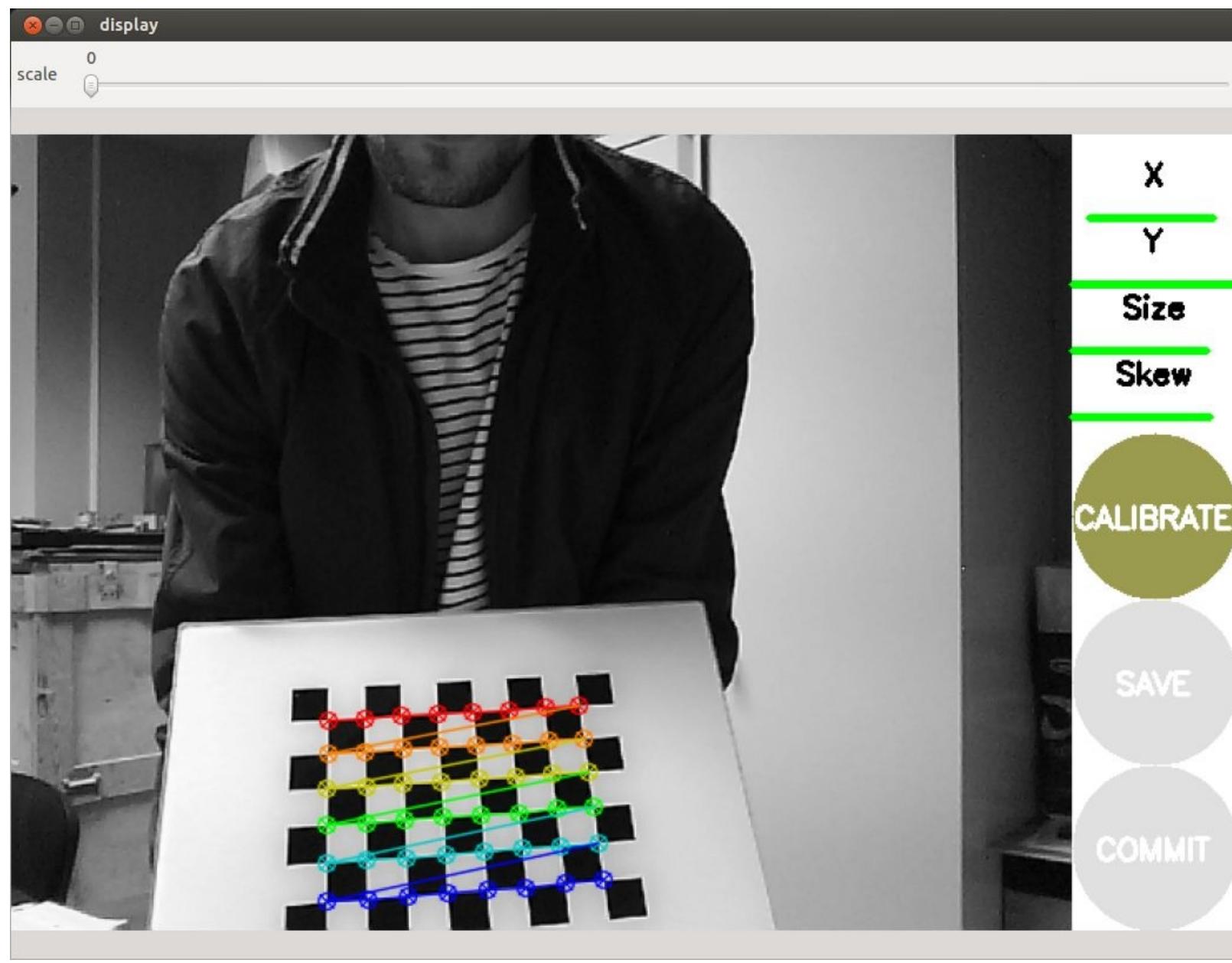
```
roslaunch chapter5_tutorials calibration_firewire_camera_circles.lau  
nchroslaunch chapter5_tutorials calibration_firewire_camera_acircles.la  
unch
```

You can also use multiple chessboard patterns for a single calibration using patterns of different sizes. However, we think it is enough to use a single chessboard pattern printed with good quality. Indeed, for the USB camera driver, we only use that.

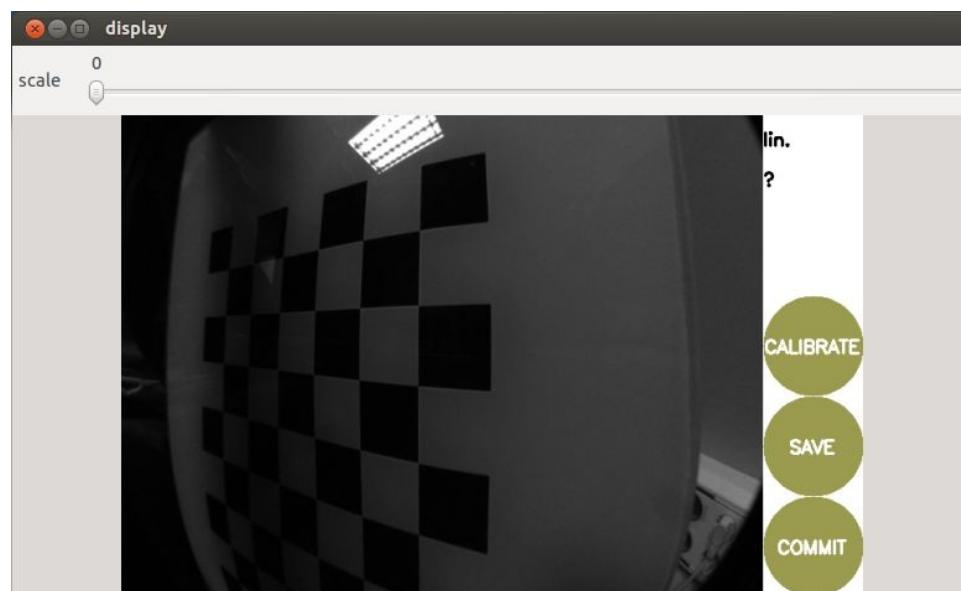
In the case of the USB camera driver, we have a more powerful launch file that integrates the camera calibration node; there is also a standalone one for FireWire cameras. In order to calibrate your camera, use the following action:

```
| $ rosrun chapter5_tutorials camera.launch calibrate:=true
```

In the following screenshots, you will see the steps of the calibration process in the GUI, identical to the case of FireWire cameras. That means we have an operating `camera_info_set` service:



After pressing the CALIBRATE button, the calibration optimization algorithm will take a while to find the best intrinsic and extrinsic parameters. Once it is done, SAVE and COMMIT will be enabled. The following screenshot shows this:



Stereo calibration

The next step consists of working with stereo cameras. One option is to run two monocular camera nodes, but in general, it is better to consider the whole stereo pair as a single sensor because the images must be synchronized. In ROS, there is no driver for FireWire stereo cameras, but we can use an extension to stereo using the following command line:

```
| $ git clone git://github.com/srv/camera1394stereo.git
```

However, FireWire stereo pairs are quite expensive. For this reason, we provide a stereo camera driver for USB cameras. We use the Logitech C120 USB webcam, which is very cheap. It is also noisy, but we will see that we can do great things with it after calibration. It is important that in the stereo pair, the cameras are similar, but you can try with different cameras as well. Our setup for the cameras is shown in the images. You only need the two cameras on the same plane, pointing in parallel directions.

We have a baseline of approximately 12 cm, which will also be computed in the stereo calibration process. As you can see in the following screenshot, you only need a rod to attach the cameras to, with zip ties:



Now, connect the cameras to your USB slots. It is good practice to connect the left-hand side camera first and then the right-hand side one. This way, they are assigned to the `/dev/video0` and `/dev/video1` devices, or `1` and `2` if `0` was already taken. Alternatively, you can create a `udev` rule.

Then, you can test each camera individually, as we would for a single camera. Some tools you will find useful are the `video4linux` control panels for cameras:

```
| $ sudo apt-get install v4l-utils qv4l2
```

You might experience the following problem:

```
| In case of problems with stereo:  
| libv4l2: error turning on stream: No space left on device
```

This happens because you have to connect each camera to a different USB controller; note that certain USB slots are managed by the same controller, and hence it cannot deal with the bandwidth of more than a single camera. If you only have a USB controller, there are other options you can try. First, try to use a compressed pixel format, such as MJPEG in both cameras. You can check whether or not it is supported by your by using the following command:

```
| $ v4l2-ctl -d /dev/video2 --list-formats
```

The command will generate something similar to the following output:

```
ioctl: VIDIOC_ENUM_FMT
Index : 0
Type : Video Capture
Pixel Format: 'YUYV'
Name : YUV 4:2:2 (YUYV)

Index : 1
Type : Video Capture
Pixel Format: 'MJPG' (compressed)
Name : MJPG
ioctl: VIDIOC_ENUM_FMT
Index : 0
Type : Video Capture
Pixel Format: 'YUYV'
Name : YUV 4:2:2 (YUYV)

Index : 1
Type : Video Capture
Pixel Format: 'MJPG' (compressed)
```

If MJPEG is supported, we can use more than one camera in the same USB controller; otherwise, with uncompressed pixel formats, we must use different USB controllers or reduce the resolution to 320 x 240 or lower. Similarly, with the GUI of `qv4l2`, you can check this and test your camera. You can also check whether it is possible to set the desired pixel format. In fact, this does not work for our USB cameras using the OpenCV set method, so we use a USB slot managed by a different USB controller.

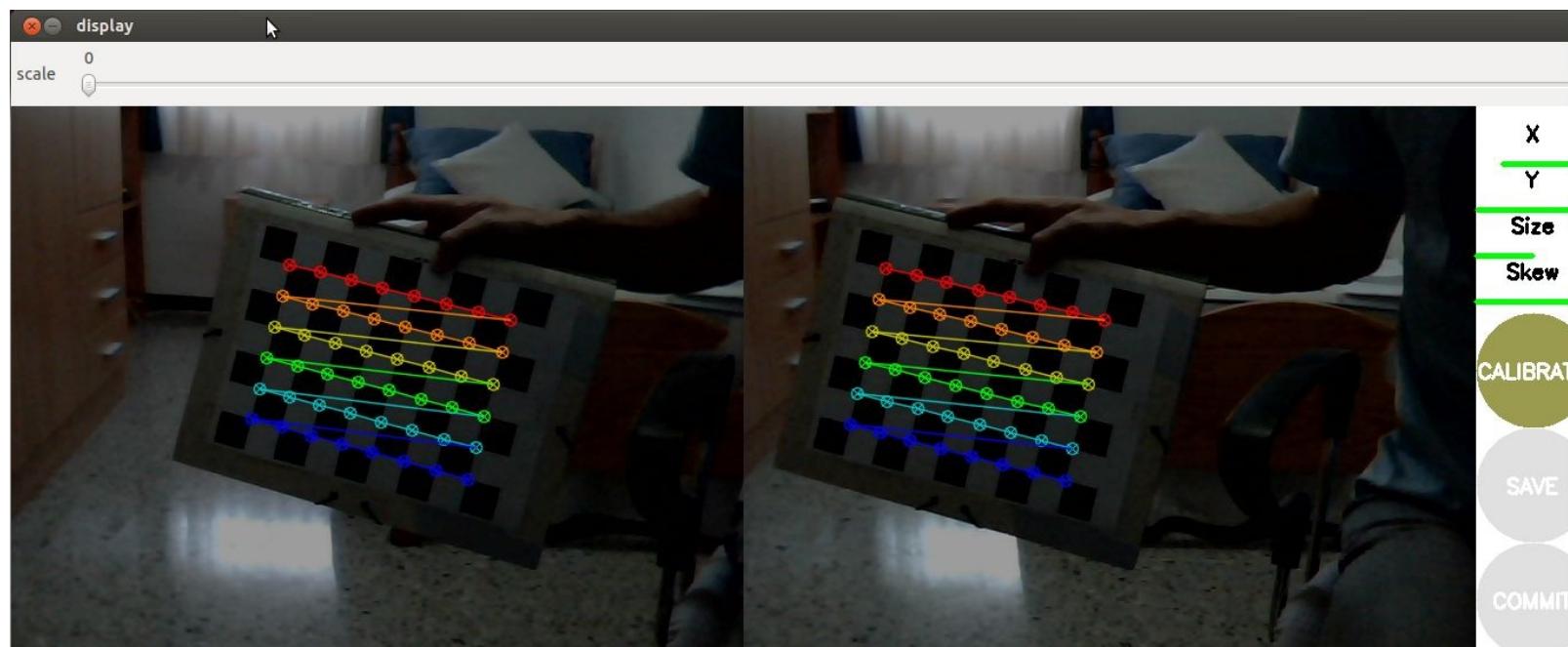
The USB stereo camera driver that comes with this section is based on the USB camera discussed so far. Basically, the driver extends the camera to support camera publishers, which send the left-hand side and right-hand side images and the camera information as well. You can run it and view the images by using the following command:

```
| $ roslaunch chapter5_tutorials camera_stereo.launch view:=true
```

It also shows the disparity image of the left-hand side and right-hand side cameras, which will be useful once the cameras are calibrated as it is used by the ROS image pipeline. In order to calibrate the cameras, use the following command:

```
| $ roslaunch chapter5_tutorials camera_stereo.launch calibrate:=true
```

You will see a GUI for monocular cameras similar to the following screenshot:



At the time the preceding image was taken, we had shown enough views to start the calibration. Note that the calibration pattern must be detected by both cameras simultaneously to be included in the calibration optimization step. Depending on the setup, this may be quite tricky, so you should put the pattern at an appropriate distance from the camera. You will see the setup used for the calibration of this section in the following image:



The calibration is done by the same `cameracalibrator.py` node used for monocular cameras. We pass the left-hand side and right-hand side cameras and images, so the tool knows that we are going to perform stereo calibration. The following is the node in the launch file:

```
&lt;node ns="$(arg camera)" name="cameracalibrator"
pkg="camera_calibration" type="cameracalibrator.py"
args="--size 8x6 --square 0.030" output="screen">
  &lt;remap from="left" to="left/image_raw"/>
  &lt;remap from="right" to="right/image_raw"/>
  &lt;remap from="left_camera" to="left"/>
  &lt;remap from="right_camera" to="right"/>
&lt;/node>
```

The result of the calibration is the same as for monocular cameras, but in this case, we have two calibration files, one for each camera. In accordance with the parameters file in `config/camera_stereo/logitech_c120.yaml`, we have the following code:

```
camera_info_url_left:
package://chapter5_tutorials/calibration/camera_stereo/
${NAME}.yaml
camera_info_url_right:
package://chapter5_tutorials/calibration/camera_stereo/
${NAME}.yaml
```

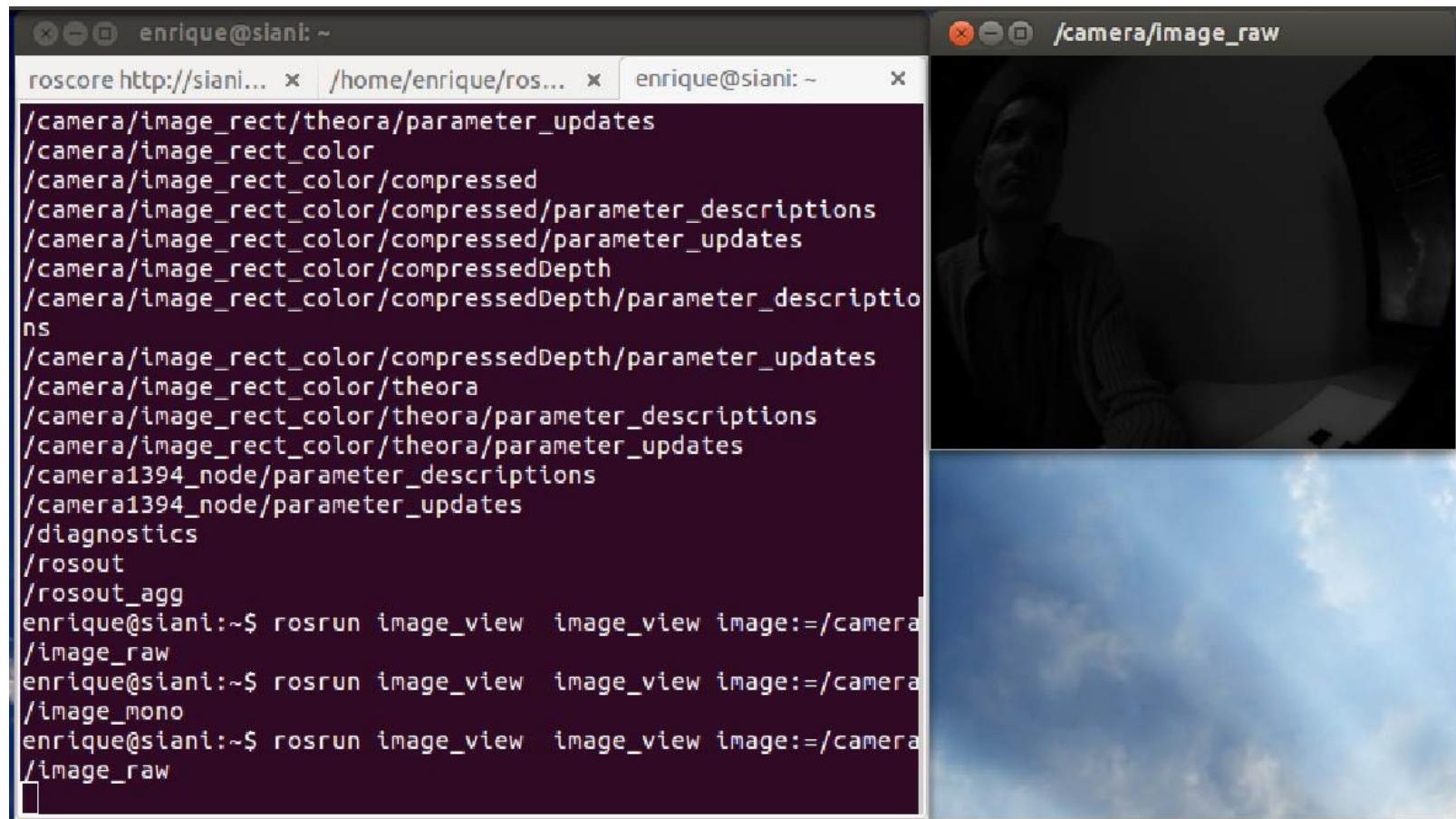
`${NAME}` is the name of the camera, which resolved to `logitech_c120_left` and `logitech_c120_right` for the left-hand side and right-hand side cameras, respectively. After the commit of the calibration, those files are updated with the calibration of each camera. They contain the calibration matrix, the distortion model coefficients, and the rectification and projection matrix, which includes the baseline, that is, the separation between each camera in the `x` axis of the image plane. In the parameters file, you can also see

values for the camera properties that have been set for indoor environments with artificial light; the camera model used has some auto correction, so sometimes the images may be bad, but these values seem to work well in most cases.

The ROS image pipeline

The ROS image pipeline is run with the `image_proc` package. It provides all the conversion utilities for obtaining monochrome and color images from the RAW images acquired from the camera. In the case of FireWire cameras, which may use a Bayer pattern to code the images (actually in the sensor itself), it *debayers* them to obtain the color images. Once you have calibrated the camera, the image pipeline takes the `cameraInfo` messages, which contain that information, and rectifies the images. Here, rectification means to un-distort the images, so it takes the coefficients of the distortion model to correct the radial and tangential distortion.

As a result, you will see more topics for your camera in its namespace. In the following screenshots, you can see the `image_raw`, `image_mono`, and `image_colour` topics, which display the RAW, monochrome, and color images, respectively:



The mono color, in this case, is equivalent to the following raw image. Note that for good cameras the raw image usually employs a Bayer pattern for the cells on the camera sensor, so it will not be equivalent to the mono color one.

```
enrique@siani: ~ roscore http://siani... x /home/enrique/ros... x enrique@siani: ~ x
/camera/image_rect_color/compressed
/camera/image_rect_color/compressed/parameter_descriptions
/camera/image_rect_color/compressedDepth/parameter_updates
/camera/image_rect_color/compressedDepth
/camera/image_rect_color/compressedDepth/parameter_descriptions
/camera/image_rect_color/compressedDepth/parameter_updates
/camera/image_rect_color/theora
/camera/image_rect_color/theora/parameter_descriptions
/camera/image_rect_color/theora/parameter_updates
/camera1394_node/parameter_descriptions
/camera1394_node/parameter_updates
/diagnostics
/rosout
/rosout_agg
enrique@siani:~$ rosrun image_view image_view image:=/camera
/image_raw
enrique@siani:~$ rosrun image_view image_view image:=/camera
/image_mono
enrique@siani:~$ rosrun image_view image_view image:=/camera
/image_raw
enrique@siani:~$ rosrun image_view image_view image:=/camera
/image_mono
```



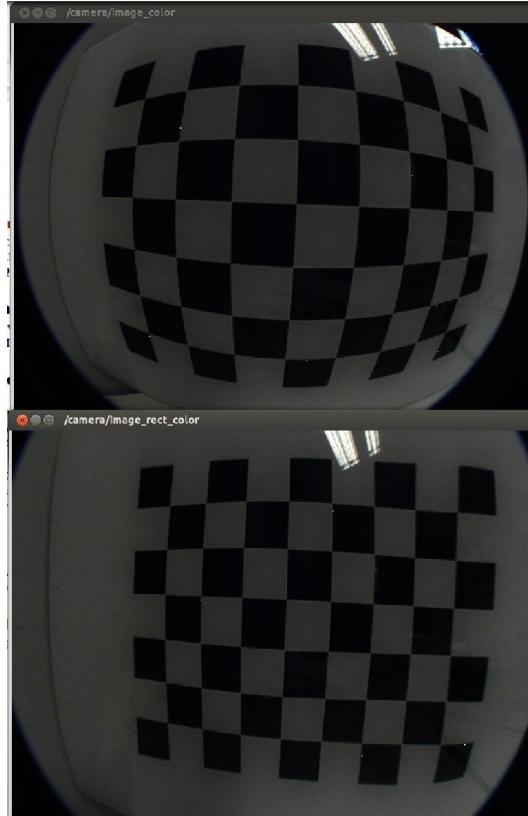
Finally, the following color image is shown, which is only different from the mono color one if the camera actually supports colors; otherwise, it will be mono color too:

```
enrique@siani: ~ roscore http://siani... x /home/enrique/ros... x enrique@siani: ~ x
/camera/image_rect_color/compressed/parameter_updates
/camera/image_rect_color/compressedDepth
/camera/image_rect_color/compressedDepth/parameter_descriptions
/camera/image_rect_color/compressedDepth/parameter_updates
/camera/image_rect_color/theora
/camera/image_rect_color/theora/parameter_descriptions
/camera/image_rect_color/theora/parameter_updates
/camera1394_node/parameter_descriptions
/camera1394_node/parameter_updates
/diagnostics
/rosout
/rosout_agg
enrique@siani:~$ rosrun image_view image_view image:=/camera
/image_raw
enrique@siani:~$ rosrun image_view image_view image:=/camera
/image_mono
enrique@siani:~$ rosrun image_view image_view image:=/camera
/image_raw
enrique@siani:~$ rosrun image_view image_view image:=/camera
/image_mono
enrique@siani:~$ rosrun image_view image_view image:=/camera
/image_color
```



The rectified images are provided in monochrome and color in the `image_rect` and `image_rect_color` topics. In

the following image, we compare the uncalibrated, distorted RAW images with the rectified ones. You can see the correction because the pattern shown in the screenshots has straight lines only in the rectified images, particularly in areas far from the center (principle point) of the image (sensor):



You can see all the topics available with `rostopic list` or `rqt_graph`, which include the `image_transport` topics as well.

You can view the `image_raw` topic of a monocular camera directly with the following command:

```
| $ rosrun chapter5_tutorials camera.launch view:=true
```

It can be changed to see other topics, but for these cameras, the RAW images are already in color. However, in order to see the rectified images, use `image_rect_color` with `image_view` or `rqt_image_view` or change the launch file. The `image_proc` node is used to make all these topics available. The following code shows this:

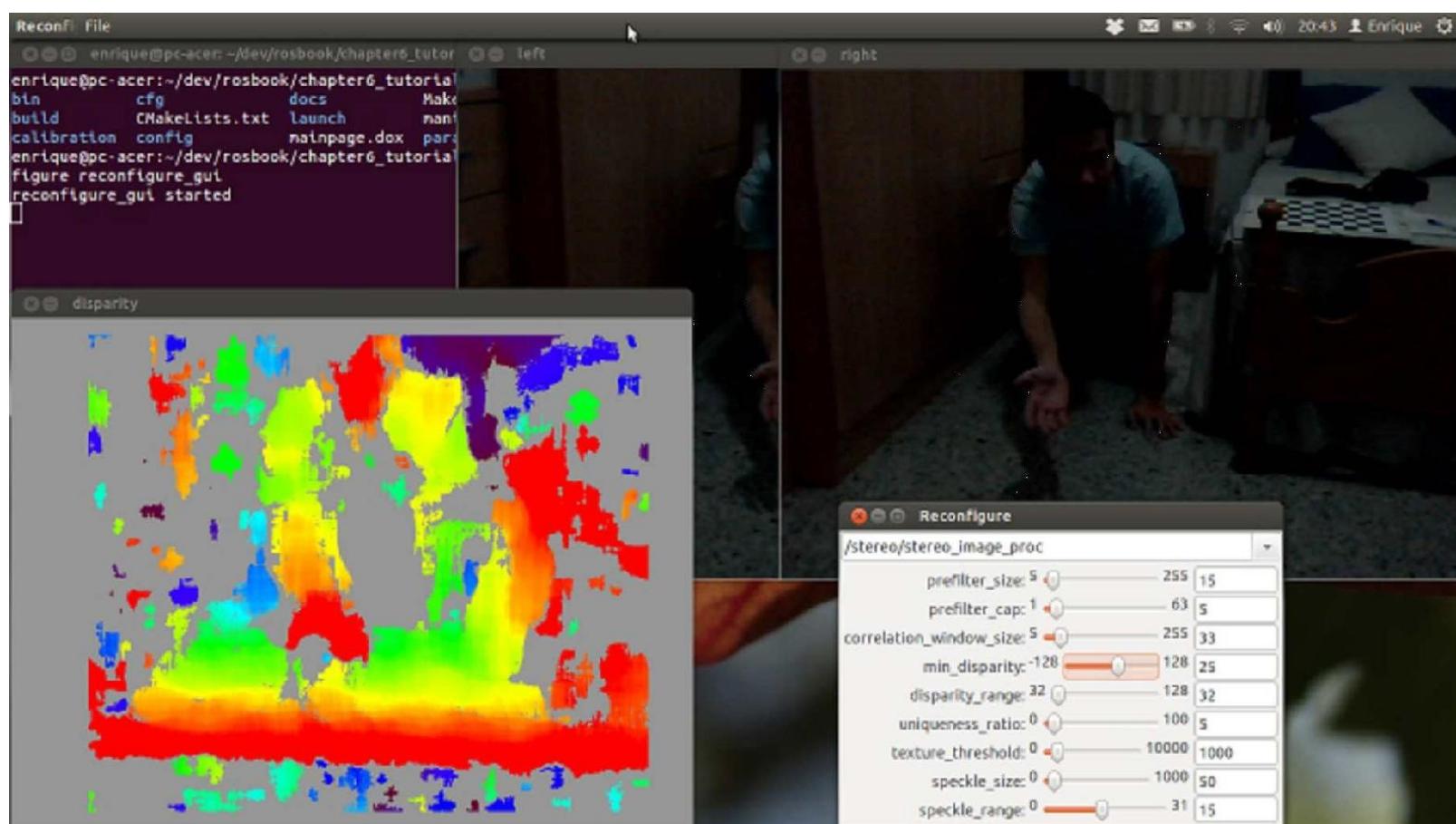
```
| <node ns="$(arg camera)" pkg="image_proc" type="image_proc"  
|   name="image_proc"/>
```

Image pipeline for stereo cameras

In the case of stereo cameras, we have the same for the left-hand side and right-hand side cameras. However, there are specific visualization tools for them because we can use the left-hand side and right-hand side images to compute and see the disparity image. An algorithm uses stereo calibration and the texture of both images to estimate the depth of each pixel, which is the disparity image. To obtain good results, we must tune the algorithm that computes such an image. In the following screenshot, we will see the left-hand side, right-hand side, and disparity images as well as `rqt_reconfigure` for `stereo_image_proc`, which is the node that builds the image pipeline for stereo images; in the launch file, we only need the following lines:

```
&lt;node ns="$(arg camera)" pkg="stereo_image_proc"  
        type="stereo_image_proc"  
        name="stereo_image_proc" output="screen">  
  &lt;rosparam file="$(argparams_disparity)" />  
&lt;/node>
```

It requires the disparity parameters, which can be set with `rqt_reconfigure`, as shown in the following screenshot, and saved with `rosparam dump/stereo/stereo_image_proc`:



We have good values for the environment used in this section's demonstration in the `config/camera_stereo/disparity.yaml` parameters file. This is shown in the following code:

```
{correlation_window_size: 33, disparity_range: 32, min_disparity:  
  25, prefilter_cap: 5, prefilter_size: 15, speckle_range: 15, speckle_size: 50,  
  texture_threshold: 1000,  
  uniqueness_ratio: 5.0}
```

However, these parameters depend on a lot on the calibration quality and the environment. You should adjust it to your experiment. It takes time and it is quite tricky, but you can follow the next guidelines. You start by setting a `disparity_range` value that makes enough blobs appear. You also have to set `min_disparity`, so you can see areas covering the whole range of depths (from red to blue/purple). Then, you can fine-tune the result, setting `speckle_size` to remove small, noisy blobs. You can also modify `uniqueness_ratio` and `texture_threshold` to have larger blobs. The `correlation_window_size` is also important as it affects the detection of initial blobs.

If it becomes difficult to obtain good results, you might have to recalibrate or use better cameras for your environment and lighting conditions. You can also try it in another environment or with more light. It is important that you have texture in the environment; for example, against a flat, white wall, you cannot find any disparity. Also, depending on the baseline, you cannot retrieve depth information very close to the camera. For stereo navigation, it is better to have a large baseline, say 12 cm or more. We use it here because we are trying visual odometry later. With this setup, we only have depth information one meter away from the cameras. With a smaller baseline, we can obtain depth information from closer objects. This is bad for navigation because we lose resolution far away, but it is good for perception and grasping.

As far as calibration problems go, you can check your calibration results with the `cameracheck.py` node, which is integrated in both the monocular and stereo camera launch files:

```
$ roslaunch chapter5_tutorials camera.launch view:=true check:=true  
$ roslaunch chapter5_tutorials camera_stereo.launch view:=true  
check:=true
```

For the monocular camera, our calibration yields this RMS error (see more in `calibration/camera/cameracheck-stdout.log`):

```
Linearity RMS Error: 1.319 Pixels ReprojectionRMS Error: 1.278  
PixelsLinearity RMS Error: 1.542 Pixels ReprojectionRMS Error: 1.368  
PixelsLinearity RMS Error: 1.437 Pixels ReprojectionRMS Error: 1.112  
PixelsLinearity RMS Error: 1.455 Pixels ReprojectionRMS Error: 1.035  
PixelsLinearity RMS Error: 2.210 Pixels ReprojectionRMS Error: 1.584  
PixelsLinearity RMS Error: 2.604 Pixels ReprojectionRMS Error: 2.286  
PixelsLinearity RMS Error: 0.611 Pixels ReprojectionRMS Error: 0.349  
Pixels
```

For the stereo camera, we have `epipolar error` and the estimation of the cell size of the calibration pattern (see more in `calibration/camera_stereo/cameracheck-stdout.log`):

```
epipolar error: 0.738753 pixels dimension: 0.033301 m  
epipolar error: 1.145886 pixels dimension: 0.033356 m  
epipolar error: 1.810118 pixels dimension: 0.033636 m  
epipolar error: 2.071419 pixels dimension: 0.033772 m  
epipolar error: 2.193602 pixels dimension: 0.033635 m  
epipolar error: 2.822543 pixels dimension: 0.033535 m
```

To obtain these results, you only have to show the calibration pattern to the camera; that is the reason we also pass `view:=true` to the launch files. An RMS error greater than 2 pixels is quite large; we have something around it, but remember that these are very low-cost cameras. Something below a 1 pixel error is desirable. For the stereo pair, the `epipolar error` should also be lower than 1 pixel; in our case, it is still quite large (usually greater than 3 pixels), but we still can do many things. Indeed, the disparity image is just a representation of the depth of each pixel shown with the `stereo_view` node. We also have a 3D point cloud that can be visualized and texturized with `rviz`. We will see this in the following demonstrations on

visual odometry.

ROS packages useful for Computer Vision tasks

The great advantage of doing Computer Vision in ROS is the fact that we do not have to re-invent the wheel. A lot of third-party software is available, and we can also connect our vision stuff to the real robots or perform simulations. Here, we are going to enumerate interesting Computer Vision tools for the most common visual tasks, but we will only explain one of them in detail, including all the steps to set it up.

We will do this for visual odometry, but other packages are also easy to install and it is also easy to start playing with them; simply follow the tutorials or manuals in the links provided here:

- **Visual Servoing:** Also known as **vision-based robot control**, this is a technique that uses feedback information obtained from a vision sensor to control the motion of a robot, typically an arm used for grasping. In ROS, we have a wrapper of the **Visual Servoing Platform (ViSP)** software (<http://www.iris.a.fr/lagadic/visp/visp.html> and http://www.ros.org/wiki/vision_visp). ViSP is a complete cross-platform library that allows you to prototype and develop applications in visual tracking and visual serving. The ROS wrapper provides a tracker that can be run with the `visp_tracker` (moving edge tracker) node as well as `visp_auto_tracker` (a model-based tracker). It also helps to calibrate the camera and perform hand-to-eye calibration, which is crucial for visual serving in grasping tasks.
- **Augmented Reality:** An **Augmented Reality (AR)** application involves overlaying virtual imagery in the real world. A well-known library for this purpose is ARToolkit (<https://www.hitl.washington.edu/artoolkit/>). The main problem in this application is tracking the user's viewpoint to draw the virtual imagery on the viewpoint where the user is looking in the real world. **ARToolkit** video tracking libraries calculate the real camera position and orientation relative to physical markers in real time. In ROS, we have a wrapper named `ar_pose` (http://www.ros.org/wiki/ar_pose). It allows us to track single or multiple markers where we can render our virtual imagery (for example, a 3D model).
- **Perception and object recognition:** Most basic perception and object recognition is possible with the OpenCV libraries. However, there are several packages that provide an object recognition pipeline, such as the `object_recognition` stack, which provides `tabletop_object_detector` to detect objects on a table; for example, a more general solution provided by **Object Recognition Kitchen (ORK)** can be found at http://wg-perception.github.io/object_recognition_core. It is also worth mentioning a tool called **RoboEarth** (<http://www.roboearth.org>), which allows you to detect and build 3D models of physical objects and store them in a global database accessible for any robot (or human) worldwide. The models stored can be 2D or 3D and can be used to recognize similar objects and their viewpoint, that is, to identify what the camera/robot is watching. The RoboEarth project is integrated into ROS, and many tutorials are provided to have a running system (<http://www.ros.org/wiki/roboearth>), although it does not support officially the latest versions of ROS.
- **Visual odometry:** A visual odometer is an algorithm that uses the images of the environment to track features and estimate the robot's movement, assuming a static environment. It can solve the six DoF poses of the robot with a monocular or stereo system, but it may require additional information in the monocular case. There are two main libraries for visual odometry: `libviso2` (<http://www.cvlabs.net/software/libviso2.html>) and `libfovise` (http://www.ros.org/wiki/fovise_ros), both of which have wrappers for ROS. The wrappers just expose these libraries to ROS. They are the `viso2` and `fovise` packages, respectively. In

the following section, we will see how we can do visual odometry with our homemade stereo camera using the `viso2_ros` node of `viso2`. We also show how to install them, which at the moment needs to be done from source because these packages do not support ROS Kinetic officially; however, there is no other alternative for them integrated in ROS. The `libviso2` library allows us to do monocular and stereo visual odometry. However, for monocular odometry, we also need the pitch and heading for the ground plane estimation. You can try the monocular case with one camera and an IMU , but you will always have better results with a good stereo pair, correctly calibrated, as seen so far in this chapter. Finally, `libfovis` does not allow the monocular case, but it does support RGBD cameras, such as the Kinect sensor . In regards the stereo case, it is possible to try both libraries and see which one works better in your case. Here, we will give you a step-by-step tutorial on how to install and run `viso2` in ROS and `fovis` with Kinect.

Visual odometry

Visual odometry is the name used for algorithms that use vision to estimate the relative displacement of a mobile robot or sensor. Odometry accumulates the consecutive relative displacement to give a global estimation of the robot or sensor pose in the environment in respect to the initial pose, but bear in mind that it accumulates drift because the error on each relative displacement accumulates and grows without bounds. To solve this problem, a global localization or loop closure detection algorithm is needed. This is one of the components of a Visual SLAM system, but it also needs visual odometry to create a reference guess for the pose estimation.

Using visual odometry with viso2

In order to use `viso2`, go to your `catkin` workspace (`~/catkin_ws`) and use the following commands:

```
| $ cdsrc  
| $ wstoolinit  
| $ wstool set viso2 --git git://github.com/srv/viso2.git  
| $ wstool update
```

Now, to build it, run the following command:

```
| $ cd ..  
| $ catkin_make
```

Once it is built, we set up our environment by using the following command:

```
| $ sourcedevel/setup.bash
```

Now we can run `viso2_ros` nodes, such as `stereo_odometer`, which is the one we are going to use here. But before that, we need to publish the frame transformation between our camera and the robot or its base link. The stereo camera driver is already prepared for that, but we will explain how it is done in the following sections.

Camera pose calibration

In order to set the transformation between the different frames in our robot system, we must publish the `tf` message of such transforms. The most appropriate and generic way to do so consists of using the `camera_pose` stack; we will use the latest version from this repository, which can be found at https://github.com/john-ren-forks/camera_pose. This stack offers a series of launch files that calibrate the camera poses with respect to each other. It comes with launch files for two, three, or more cameras. In our case, we only have two cameras (stereo), so we proceed this way. First, we extend `camera_stereo.launch` with the `calibrate_pose` argument that calls `calibration_tf_publisher.launch` from `camera_pose`:

```
&lt;include file="$(find  
camera_pose_calibration)/blocks/calibration_tf_publisher.launch"> &lt;arg name="cache_file"  
value="/tmp/camera_pose_calibration_cache.bag"/>  
&lt;/include>
```

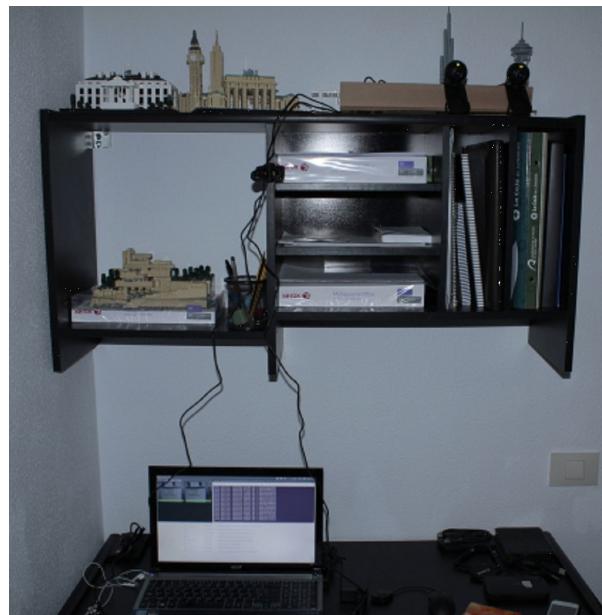
Now, run the following command:

```
| $ roslaunch chapter5_tutorials camera_stereo.launch calibrate_pose:=true
```

The `calibration_tf_publisher` will publish the frame transforms (`tf`) as soon as the calibration has been done correctly. The calibration is similar to the one we have seen so far but uses the specific tools from `camera_pose`, which are run using the following command:

```
| $ roslaunch camera_pose_calibration calibrate_2_ camera.launch camera1_ns:=/stereo/left  
camera2_ns:=/stereo/right checker_rows:=6 checker_cols:=8 checker_size:=0.03
```

With this call, we can use the same calibration pattern we used with our previous calibration tools. However, this requires the images to be static; some bars can move from one side to another of the image and turn green when the images in all the cameras have been static for a sufficient period of time. This is shown in the following screenshot:

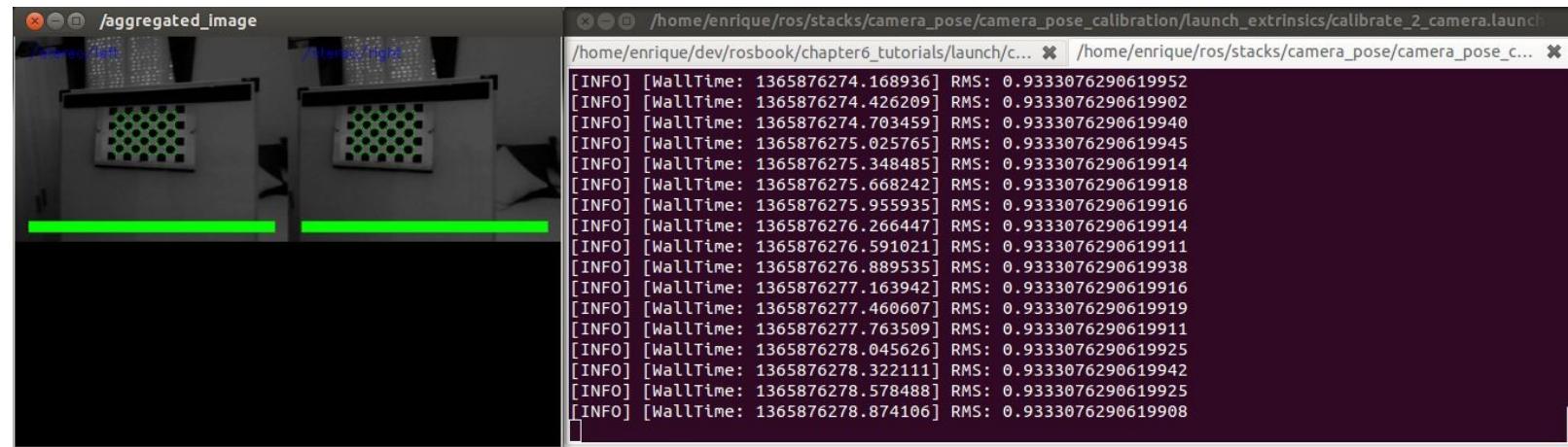


With our noisy cameras, we will need support for the calibration pattern, such as a tripod or a panel, as

shown in the following image:



Then, we can calibrate, as shown in the following screenshot:



This creates `tf` from the left-hand side camera to the right-hand side camera. However, although this is the most appropriate way to perform the camera pose calibration, we are going to use a simpler approach that is enough for a stereo pair. This is also required by `viso2`, as it needs the frame of the whole stereo pair as a single unit/sensor; internally, it uses the stereo calibration results of `cameracalibrator.py` to retrieve the baseline.

We have a launch file that uses `static_transform_publisher` for the camera link to the base link (robot base) and another from the camera link to the optical camera link because the optical one requires rotation; recall that the camera frame has the z axis pointing forward from the camera's optical lens, while the other frames (world, navigation, or odometry) have the z axis pointing up. This launch file is in `launch/frames/stereo_frames.launch`:

```
&lt;launch>
  &lt;arg name="camera" default="stereo" />
  &lt;arg name="baseline/2" value="0.06"/>
  &lt;arg name="optical_translation" value="0 -$(arg baseline/2) 0"/>
  &lt;arg name="pi/2" value="1.5707963267948966"/>
  &lt;arg name="optical_rotation" value="-$(arg pi/2) 0 -$(arg
    pi/2)" /> &lt;node pkg="tf" type="static_transform_publisher" name="$(arg
    camera)_link"
  args="0 0 0.1 0 0 0 /base_link /$(arg camera) 100"/>
  &lt;node pkg="tf" type="static_transform_publisher" name="$(arg
    camera)_optical_link" args="$(arg optical_translation) $(arg optical_rotation) /$(arg
    camera) /$(arg camera)_optical 100"/>
&lt;/launch>
```

This launch file is included in our stereo camera launch file and publishes these static frame transforms. Hence, we only have to run the following command to get the launch file to publish them:

```
| $ roslaunch chapter5_tutorials camera_stereo.launch tf:=true
```

Then, you can check whether they are being published in `rqt_rviz` with the `tf` display, as we will see in the following running `viso2`; you can also use `rqt_tf_tree` for this (see [Chapter 3, Visualization and Debugging Tools](#)).

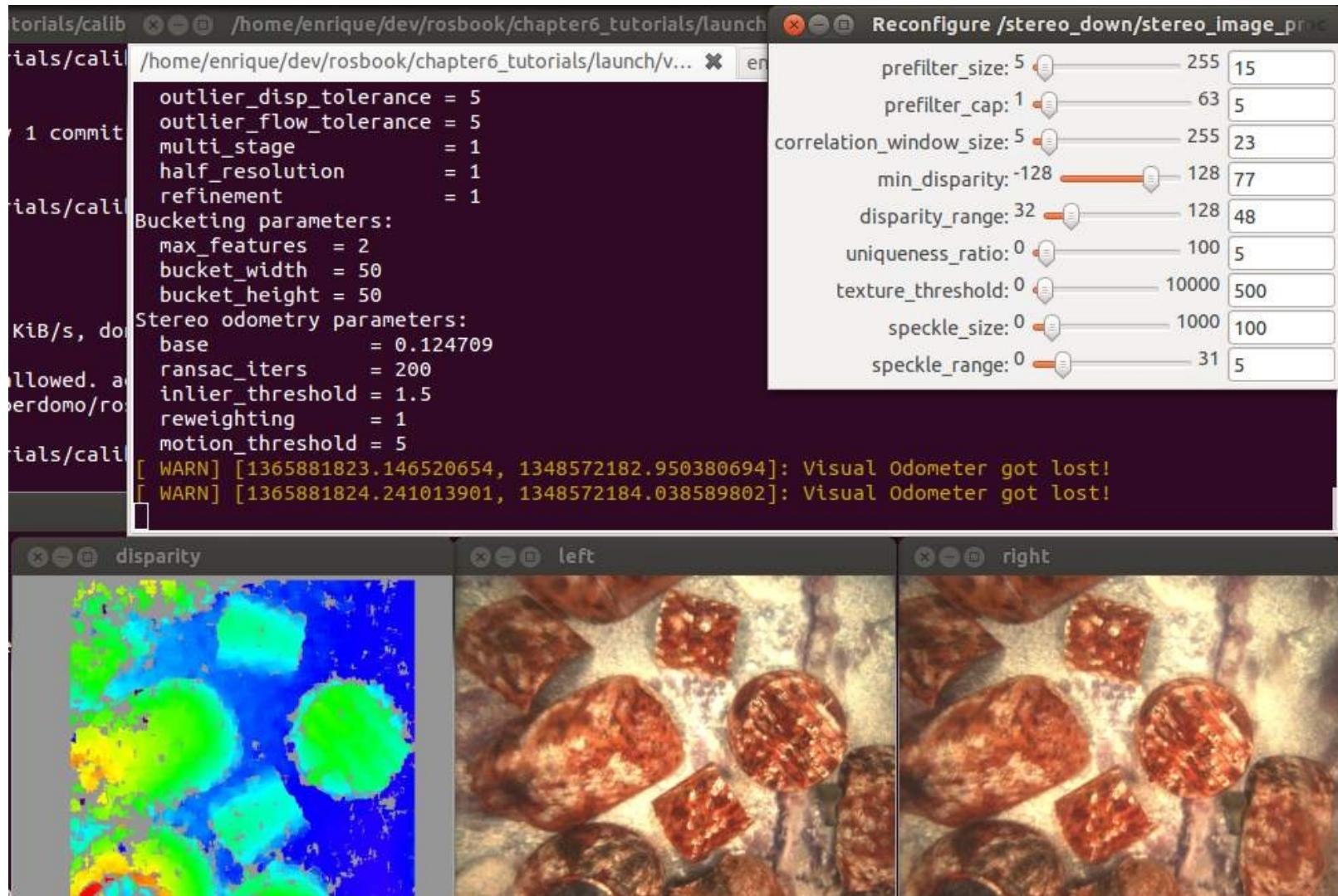
Running the viso2 online demo

At this point, we are ready to run the visual odometry algorithm: our stereo pair cameras are calibrated, their frame has the appropriate name for `viso2` (ends with `_optical`), and `tf` for the camera and optical frames has been published. However, before using our own stereo pair, we are going to test `viso2` with the bag files provided in http://srv.uib.es/public/viso2_ros/sample_bagfiles/; just run

`bag/viso2_demo/download_amphoras_pool_bag_files.sh` to obtain all the bag files (this totals about 4 GB). Then, we have a launch file for both the monocular and stereo odometers in the `launch/visual_odometry` folder. In order to run the stereo demo, we have a launch file on top that plays the bag files and also allows you to inspect and visualize its contents. For instance, to calibrate the disparity image algorithm, run the following command:

```
| $ roslaunch chapter5_tutorials viso2_demo.launch config_disparity:=true view:=true
```

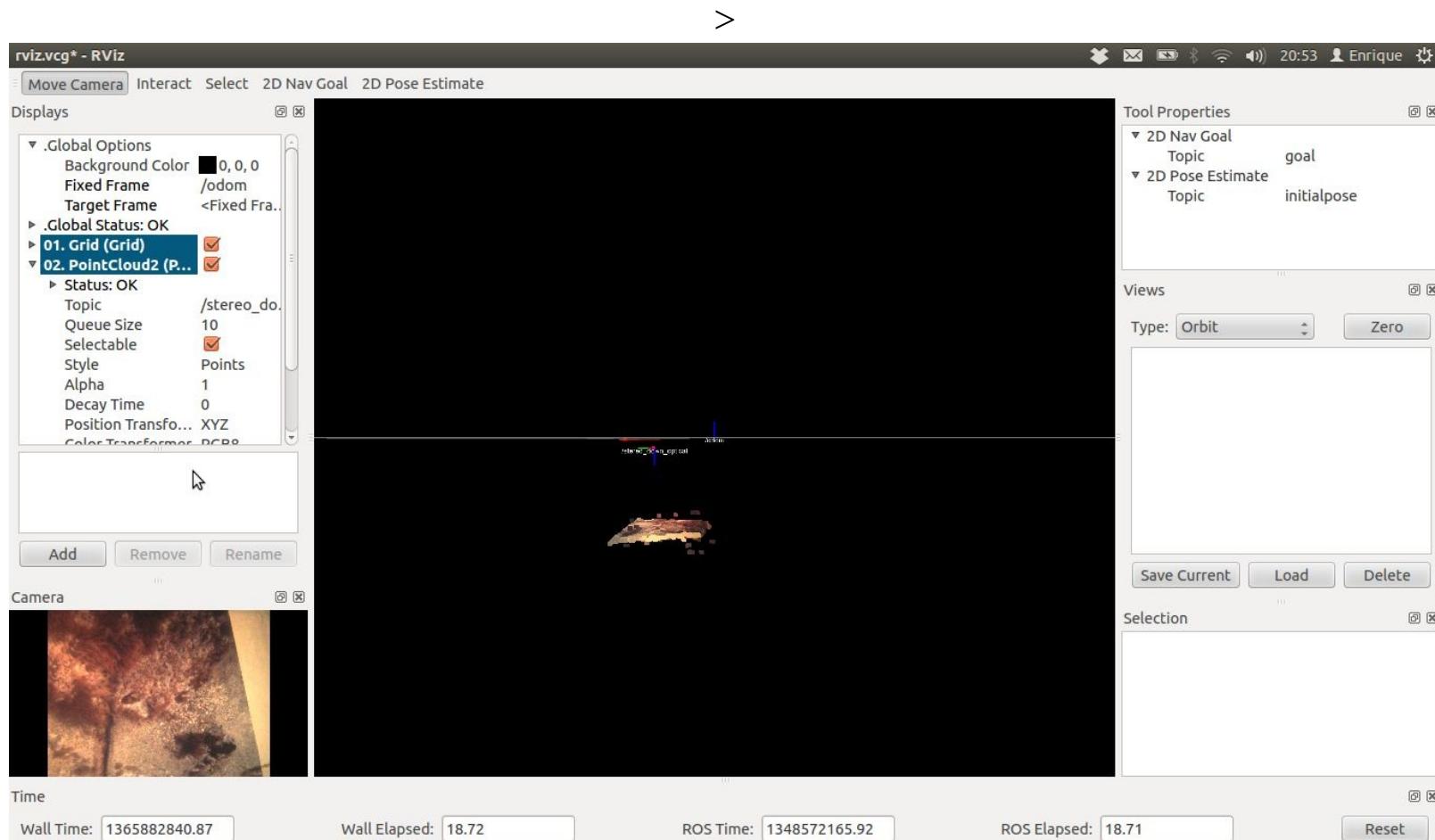
You will see that the left-hand side, right-hand side, disparity images, and the `rqt_reconfigure` interface configures the disparity algorithm. You need to perform this tuning because the bag files only have the RAW images. We have found good parameters that are in `config/viso2_demo/disparity.yaml`. In the following screenshot, you can see the results obtained using them, where you can clearly appreciate the depth of the rocks in the stereo images:



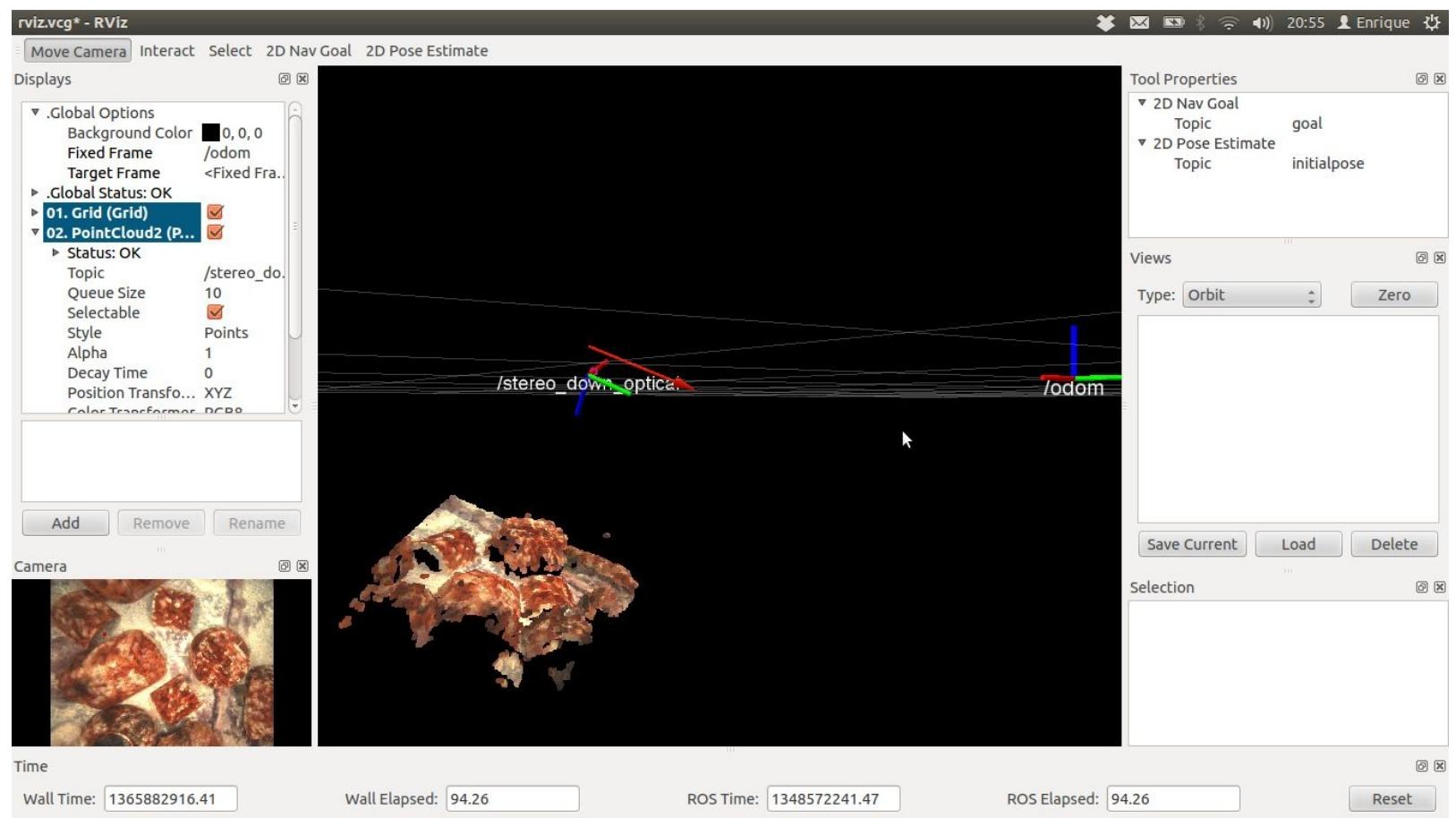
In order to run the stereo odometry and see the result in `rqt_rviz`, run the following command:

```
$ roslaunch chapter5_tutorials viso2_demo.launch odometry:=true  
rviz:=true
```

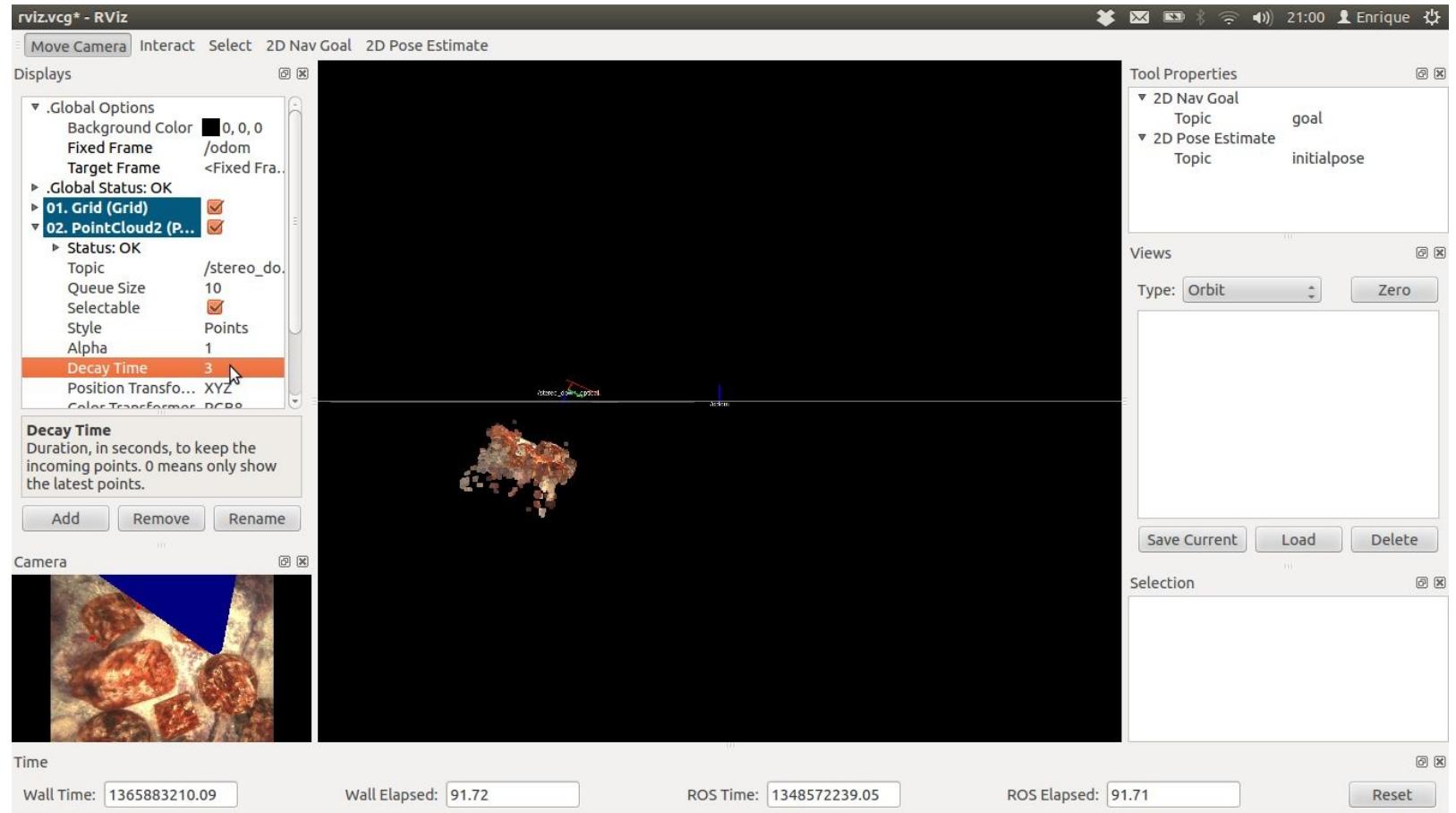
Note that we have provided an adequate configuration for `rqt_rviz` in `config/viso2_demo/rviz.rviz`, which is automatically loaded by the launch file. The following sequence of images shows different instances of the textrized 3D point cloud and the `/odom` and `/stereo_optical` frames that show the camera pose estimate from the stereo odometer. The third image has a decay time of three seconds for the point cloud, so we can see how the points overlay over time. This way, with good images and odometry, we can even see a map drawn in `rqt_rviz`, but this is quite difficult and generally needs a SLAM algorithm (see chapter 7, *Using Sensors and Actuators with ROS*). All of this is encapsulated in the following screenshots:



As new frames come, the algorithm is able to create a 3D reconstruction, as shown in the following screenshot, where we can see the heights of the rocks on the seabed:



If we set a Decay Time of three seconds, the different point clouds from consecutive frames will be shown together, so we can see a map of the bottom of the sea. Remember that the map will contain some errors because of the drift of the visual odometry algorithm:

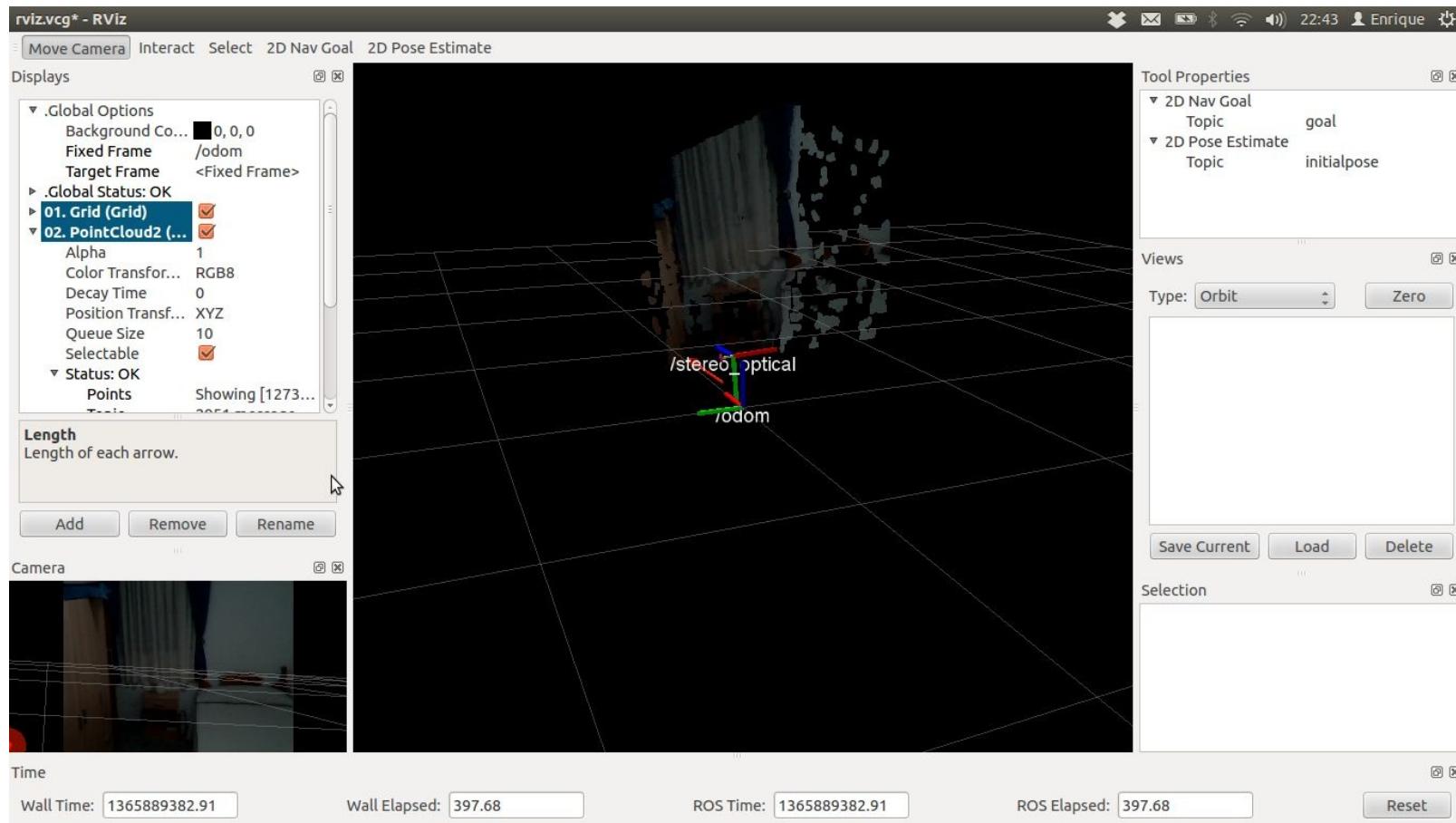


Performing visual odometry with viso2 with a stereo camera

Finally, we can do what `viso2_demo` does with our own stereo pair. We only have to run the following command to run the stereo odometry and see the results in `rqt_rviz` (note that the `tf` tree is published by default):

```
$ roslaunch chapter5_tutorials camera_stereo.launch odometry:=true  
rviz:=true
```

The following image shows an example of the visual odometry system running for our low-cost stereo camera. If you move the camera, you should see the `/odom` frame moving. If the calibration is bad or the cameras are very noisy, the odometer may get lost, which is indicated by a warning message on the terminal. In that case, you should look for better cameras or recalibrate them to see whether better results can be obtained. You might also want to have to look for better parameters for the disparity algorithm:



Performing visual odometry with an RGBD camera

Now we are going to see how to perform visual odometry using RGBD cameras using `fovis`.

Installing `fovis`

Since `fovis` is not provided as a Debian package, you must build it in your `catkin` workspace (use the same workspace as you used for `chapter5_tutorials`). Proceed with the following commands within any workspace:

```
| $ cdsrc  
| $ git clone https://github.com/srv/libfovis.git  
| $ git clone https://github.com/srv/fovis.git  
| $ cd ..  
| $ catkin_make
```

This clones two repositories that allow us to have the `fovis` software integrated in ROS. Note that the original code is hosted on this Google Code Project at <http://fovis.github.io/>.

Once this has been built successfully, set up the environment for this workspace before using the software:

```
| $ sourcedevel/setup.bash
```

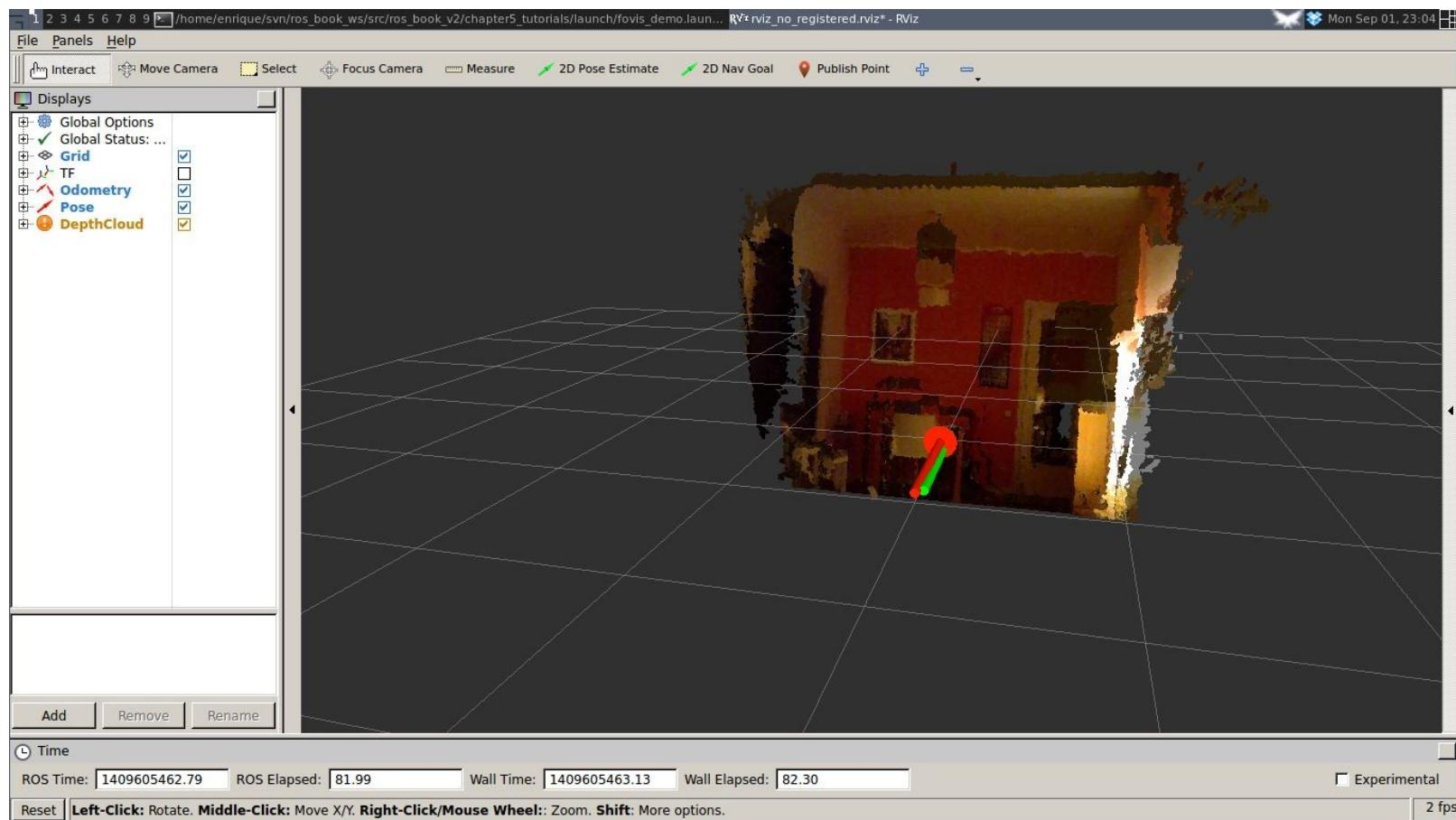
Using `fovis` with the Kinect RGBD camera

At this point, we are going to run `fovis` for the Kinect RGBD camera. This means that we are going to have 3D information to compute the visual odometry, so better results are expected than when we use stereo vision or a monocular camera (as with `viso2`).

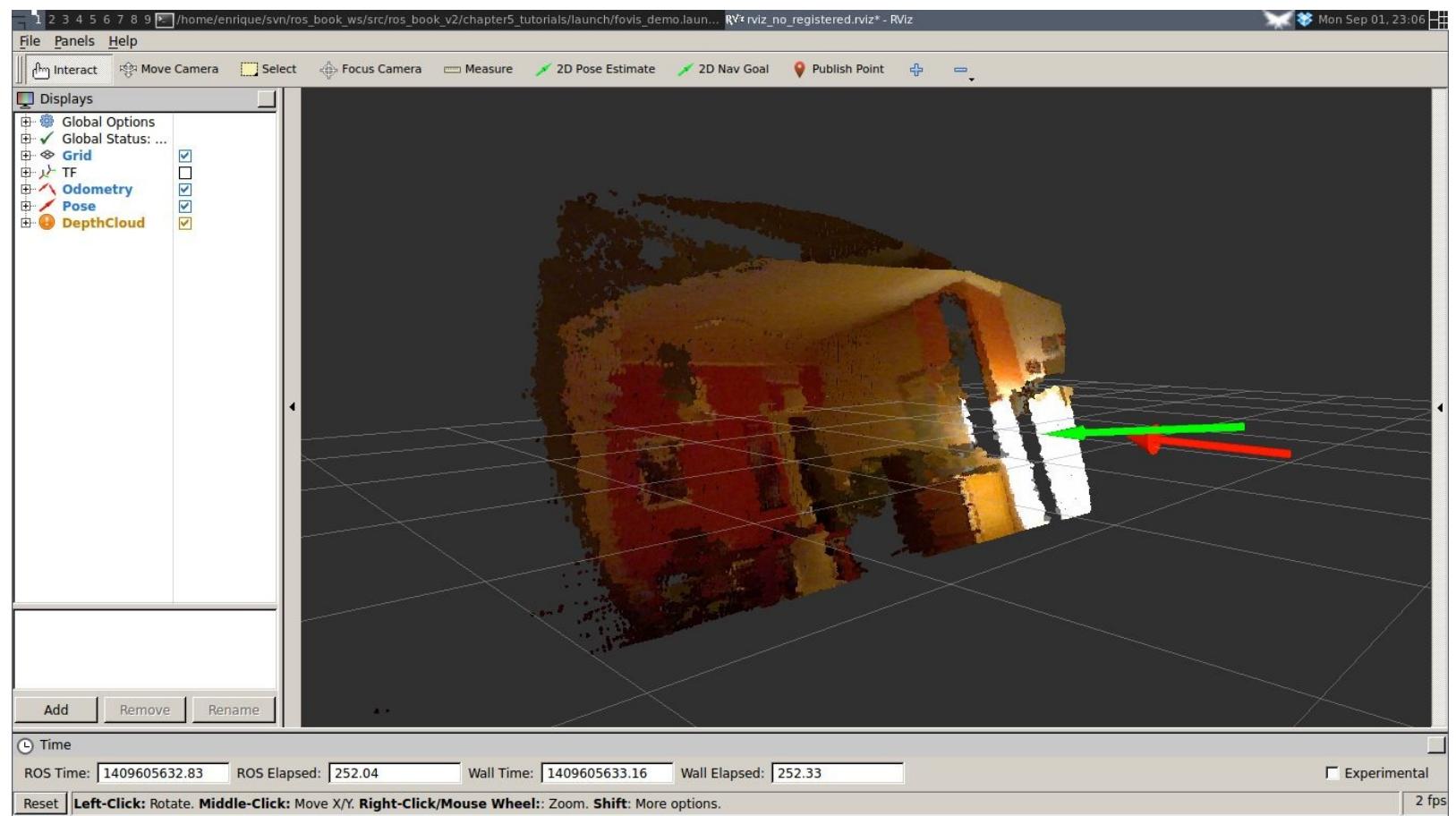
We simply have to launch the Kinect RGBD camera driver and `fovis`. For convenience, we provide a single `launch` file that runs both:

```
| $ roslaunch chapter5_tutorials fovis_demo.launch
```

Move the camera around and you should be able to have a good odometry estimation of the trajectory followed by the camera. The next figure shows this on `rviz` in the initial state before moving the camera. You can see the RGBD point cloud and two arrows showing the odometry and the current position of the camera in the following screenshot:



After moving the camera, you should see the arrows showing the camera pose (as shown in the following screenshot). Take into account that you have to move it quite slowly as the software needs time to compute the odometry, depending on the computer you are using:



By default, the `fovis_demo.launch` file uses the `no_registered` depth information. This means that the depth image is not registered or transformed into the RGB camera frame. Although it is better to register it, this drops the frame rate dramatically from the raw throughput of 30 Hz provided by the Kinect sensor to something around 2.5 Hz depending on your computing resources.

You can use `throttle` on the RGB camera frames to use the registered version. This is automatically done by the `launch` file provided. You can select between the following modes: `no_registered` (default), `hw_registered`, and `sw_registered`. Note that, in principle, the Kinect sensor does not support the hardware registration mode (`hw_registered`), which is expected to be the fastest one. Therefore, you can try the software registration mode (`sw_registered`), for which we throttle the RGB camera messages to 2.5 Hz; you can change this in `fovis_sw_registered.launch`, as shown here:

```
$ rosrun chapter5_tutorials fovis_demo.launch
mode:=sw_registered
```

Computing the homography of two images

The homography matrix is a 3×3 matrix that provides transformation up to scale from a given image and a new one, which must be coplanar. In `src/homography.cpp`, there is an extensive example that takes the first image acquired by the camera and then computes the homography for every new frame in respect to the first image. In order to run the example, take something planar, such as a book cover, and run the following command:

```
| $ roslaunch chapter5_tutorials homography.launch
```

This runs the camera driver that should grab frames from your camera (webcam), detect features (`SURF` by default), extract descriptors for each of them, and match them with the ones extracted from the first image using Flann-based matching with a cross-check filter. Once the program has the matches, the homography matrix \mathbf{H} is computed.

With \mathbf{H} , we can warp the new frame to obtain the original one, as shown in the following screenshot (matches on the top, warped image using \mathbf{H} , which is shown in plain text in the terminal):



Summary

In this chapter, we have given you an overview of the Computer Vision tools provided by ROS. We started by showing how we can connect and run several types of cameras, particularly FireWire and USB ones. The basic functionality to change their parameters was presented, so now you can adjust certain parameters to obtain images of good quality. Additionally, we provided a complete USB camera driver example.

Then, we showed how you can calibrate the camera. The importance of calibration is the ability to correct the distortion of wide-angle cameras, particularly cheap ones. The calibration matrix also allows you to perform many Computer Vision tasks, such as visual odometry and perception.

We showed how you can work with stereo vision in ROS and how you can set up an easy solution with two inexpensive webcams. We also explained the image pipeline, several APIs that work with Computer Vision in ROS, such as `cv_bridge` and `image_transport`, and the integration of OpenCV within ROS packages.

Finally, we enumerated useful tasks and topics in Computer Vision that are supported by tools developed in ROS. In particular, we illustrated the example of visual odometry using the `viso2` and `fovis` libraries. We showed you an example of data recorded with a high-quality camera and also with the inexpensive stereo pair we proposed. Finally, feature detection, descriptor extraction, and matching was shown in order to illustrate how you can obtain homography between two images. All in all, after reading and running the code in this chapter, you will have seen the basics required to perform Computer Vision in ROS.

In the next chapter, you will learn how to work with point clouds using PCL, which allows you to work with RGBD cameras.

Point Clouds

Point clouds appeared in the robotics toolbox as a way to intuitively represent and manipulate the information provided by 3D sensors, such as time-of-flight cameras and laser scanners, in which space is sampled in a finite set of points in a 3D frame of reference. The **Point Cloud Library (PCL)** provides a number of data types and data structures to easily represent not only the points of our sampled space, but also the different properties of the sampled space, such as color and normal vectors. PCL also provides a number of state-of-the-art algorithms to perform data processing on our data samples, such as filtering, model estimation, and surface reconstruction.

ROS provides a message-based interface through which PCL point clouds can be efficiently communicated, and a set of conversion functions from native PCL types to ROS messages, in much the same way as it is done with OpenCV images. Aside from the standard capabilities of the ROS API, there are a number of standard packages that can be used to interact with common 3D sensors, such as the widely used Microsoft Kinect or the Hokuyo laser, and visualize the data in different reference frames with the RViz visualizer.

This chapter will provide a background on the PCL, relevant data types, and ROS interface messages that will be used throughout the rest of the sections. Later, a number of techniques will be presented on how to perform data processing using the PCL library and how to communicate the incoming and outgoing data through ROS.

Understanding the PCL

Before we dive into the code, it's important to understand the basic concepts of both the Point Cloud Library and the PCL interface for ROS. As mentioned earlier, the former provides a set of data structures and algorithms for 3D data processing, and the latter provides a set of messages and conversion functions between messages and PCL data structures. All these software packages and libraries, in combination with the capabilities of the distributed communication layer provided by ROS, open up possibilities for many new applications in the robotics field.

In general, PCL contains one very important data structure, which is `Pointcloud`. This data structure is designed as a template class that takes the type of point to be used as a template parameter. As a result of this, the point cloud class is not much more than a container of points that includes all of the common information required by all point clouds regardless of their point type. The following are the most important public fields in a point cloud:

- `header`: This field is of the `pcl::PCLHeader` type and specifies the acquisition time of the point cloud.
- `points`: This field is of the `std::vector<PointT, ... >` type and is the container where all the points are stored. `PointT` in the vector definition corresponds to the class template parameter, that is, the point type.
- `width`: This field specifies the width of the point cloud when organized as an image; otherwise, it contains the number of points in the cloud.
- `height`: This field specifies the height of the point cloud when organized as an image; otherwise, it's always 1.
- `is_dense`: This field specifies whether the cloud contains invalid values (infinite or NaN).
- `sensor_origin_`: This field is of the `Eigen::Vector4f` type, and it defines the sensor acquisition pose in terms of a translation from the origin.
- `sensor_orientation_`: This field is of the `Eigen::Quaternionf` type, and it defines the sensor acquisition pose as a rotation.

These fields are used by PCL algorithms to perform data processing and can be used by the user to create their own algorithms. Once the point cloud structure is understood, the next step is to understand the different point types a point cloud can contain, how PCL works, and the PCL interface for ROS.

Different point cloud types

As described earlier, `pcl::PointCloud` contains a field that serves as a container for the points; this field is of the `PointT` type, which is the template parameter of the `pcl::PointCloud` class and defines the type of point the cloud is meant to store. PCL defines many types of points, but a few of the most commonly used ones are the following:

- `pcl::PointXYZ`: This is the simplest type of point and probably one of the most used; it stores only 3D *XYZ* information.
- `pcl::PointXYZI`: This type of point is very similar to the previous one, but it also includes a field for the intensity of the point. Intensity can be useful when obtaining points based on a certain level of return from a sensor. There are two other standard identical point types to this one—the first one is `pcl::InterestPoint`, which has a field to store strength instead of intensity, and `pcl::PointWithRange`, which has a field to store the range instead of either intensity or strength.
- `pcl::PointXYZRGBA`: This type of point stores 3D information as well as color (`RGB`: red, green, blue) and transparency (`A`: alpha).
- `pcl::PointXYZRGB`: This type is similar to the previous point type, but it differs in that it lacks the transparency field.
- `pcl::Normal`: This is one of the most used types of points; it represents the surface normal at a given point and a measure of its curvature.
- `pcl::PointNormal`: This type is exactly the same as the previous one; it contains the surface normal and curvature information at a given point, but it also includes the 3D *XYZ* coordinates of the point. Variants of this point are `PointXYZRGBNormal` and `PointXYZINormal`, which, as the names suggest, include color (former) and intensity (latter).

Aside from these common types of points, there are many more standard PCL types, such as `PointWithViewpoint`, `MomentInvariants`, `Boundary`, `PrincipalCurvatures`, and `Histogram`. More importantly, the PCL algorithms are all converted to templates so that not only the available types can be used, but also semantically valid user-defined types can be used.

Algorithms in PCL

PCL uses a very specific design pattern throughout the entire library to define point cloud processing algorithms. In general, the problem with these types of algorithms is that they can be highly configurable, and in order to deliver their full potential, the library must provide a mechanism for the user to specify all of the parameters required as well as the commonly used defaults.

In order to solve this problem, PCL developers decided to make each algorithm a class belonging to a hierarchy of classes with specific commonalities. This approach allows PCL developers to reuse existing algorithms in the hierarchy by deriving from them and adding the required parameters for the new algorithm, and it also allows the user to easily provide the parameter values it requires through accessors, leaving the rest to their default value. The following snippet shows how using a PCL algorithm usually looks:

```
pcl::PointCloud<pcl::PointXYZ>::Ptr cloud(new  
    pcl::PointCloud<pcl::PointXYZ>);pcl::PointCloud<pcl::PointXYZ>::Ptr result(new  
    pcl::PointCloud<pcl::PointXYZ>);  
  
pcl::Algorithm<pcl::PointXYZ> algorithm;  
algorithm.setInputCloud(cloud);  
algorithm.setParameter(1.0);  
algorithm.setAnotherParameter(0.33);  
algorithm.process (*result);
```

This approach is only followed when required within the library, so there might be exceptions to the rule, such as the I/O functionality, which is not bound by the same requirements.

The PCL interface for ROS

The PCL interface for ROS provides the means required to communicate PCL data structures through the message-based communication system provided by ROS. To do so, there are several message types defined to hold point clouds as well as other data products from the PCL algorithms. In combination with these message types, a set of conversion functions are also provided to convert from native PCL data types to messages.

Some of the most useful message types are the following:

- `std_msgs::Header`: This is not really a message type, but it is usually part of every ROS message; it holds the information about when the message was sent as well a sequence number and the frame name. The PCL equivalent is `pcl::Header` type.
- `sensor_msgs::PointCloud2`: This is possibly the most important type; this message is used to transfer the `pcl::PointCloud` type. However, it is important to take into account that this message will be deprecated in future versions of PCL in favor of `pcl::PCLPointCloud2`.
- `pcl_msgs::PointIndices`: This type stores indices of points belonging to a point cloud; the PCL equivalent type is `pcl::PointIndices`.
- `pcl_msgs::PolygonMesh`: This holds the information required to describe meshes, that is, vertices and polygons; the PCL equivalent type is `pcl::PolygonMesh`.
- `pcl_msgs::Vertices`: This type holds a set of the vertices as indices in an array, for example, to describe a polygon. The PCL equivalent type is `pcl::Vertices`.
- `pcl_msgs::ModelCoefficients`: This stores the values of the different coefficients of a model, for example, the four coefficients required to describe a plane. The PCL equivalent type is `pcl::ModelCoefficients`.

The previous messages can be converted to and from PCL types with the conversion functions provided by the ROS PCL package. All of the functions have a similar signature, which means that once we know how to convert one type, we know how to convert them all. The following functions are provided in the `pcl_conversions` namespace:

```
void fromPCL(const <PCL Type> &, <ROS Message type> &);  
void moveFromPCL(<PCL Type> &, <ROS Message type> &);  
void toPCL(const <ROS Message type> &, <PCL Type> &);  
void moveToPCL(<ROS Message type> &, <PCL Type> &);
```

Here, the PCL type must be replaced by one of the previously specified PCL types and the ROS message types by their message counterpart. `sensor_msgs::PointCloud2` has a specific set of functions to perform the conversions:

```
void toROSMsg(const pcl::PointCloud<T> &, sensor_msgs::PointCloud2  
&); void fromROSMsg(const sensor_msgs::PointCloud2 &,  
pcl::PointCloud<T> &); void moveFromROSMsg(sensor_msgs::PointCloud2 &, pcl::PointCloud<T>  
&);
```

You might be wondering what the difference between each function and its move version is. The answer is simple; the normal version performs a deep copy of the data, whereas the move versions perform a

shallow copy and nullify the source data container. This is referred to as **move semantics**.

My first PCL program

In this section, you will learn how to integrate PCL and ROS. Knowledge and understanding of how ROS packages are laid out and how to compile are required although the steps will be repeated for simplicity. The example used in this first PCL program has no use whatsoever other than serving as a valid ROS node, which will successfully compile.

The first step is to create the ROS package for this entire chapter in your workspace. This package will depend on the `pcl_conversions`, `pcl_ros`, `pcl_msgs`, and `sensor_msgs` packages:

```
$ catkin_create_pkg chapter10_tutorials pcl_conversions pcl_ros pcl_msgs sensor_msgs
```

The following step is to create the source directory in the package using the following commands:

```
$ rospack profile  
$ rosdep update  
$ roscd chapter10_tutorials  
$ mkdir src
```

In this new source directory, you should create a file named `pcl_sample.cpp` with the following code, which creates a ROS node and publishes a point cloud with 100 elements. Again, what the code does should not really be of any concern to you as it is just for the purpose of having a valid node that uses PCL and compiles without problems:

```
#include <ros/ros.h>  
#include <pcl/point_cloud.h>  
#include <pcl_ros/point_cloud.h>  
#include <pcl_conversions/pcl_conversions.h>  
#include <sensor_msgs/PointCloud2.h>  
  
main (int argc, char** argv)  
{  
    ros::init (argc, argv, "pcl_sample");  
    ros::NodeHandle nh;  
    ros::Publisher pcl_pub =  
        nh.advertise<sensor_msgs::PointCloud2> ("pcl_output", 1);  
  
    sensor_msgs::PointCloud2 output;  
    pcl::PointCloud<pcl::PointXYZ>::Ptr cloud (new  
        pcl::PointCloud<pcl::PointXYZ>);  
  
    // Fill in the cloud data  
    cloud->width = 100;  
    cloud->height = 1;  
    cloud->points.resize (cloud->width * cloud->height);  
  
    //Convert the cloud to ROS message  
    pcl::toROSMsg (*cloud, output);  
  
    pcl_pub.publish(output);  
    ros::spinOnce();  
  
    return 0;  
}
```

The next step is to add PCL libraries to `CMakeLists.txt` so that the ROS node executable can be properly linked against the system's PCL libraries:

```
find_package(PCL REQUIRED)
include_directories(include ${PCL_INCLUDE_DIRS})
link_directories(${PCL_LIBRARY_DIRS})
```

Finally, the lines to generate the executable and link against the appropriate libraries are added:

```
add_executable(pcl_sample src/pcl_sample.cpp)
target_link_libraries(pcl_sample ${catkin_LIBRARIES}
${PCL_LIBRARIES})
```

Once the final step has been reached, the package can be compiled by calling `catkin_make` as usual from the workspace `root` directory.

Creating point clouds

In this first example, the reader will learn how to create PCL point clouds composed solely of pseudorandom points. The PCL point clouds will then be published periodically through a topic named `/pcl_output`. This example provides practical knowledge on how to generate point clouds with custom data and how to convert them to the corresponding ROS message type in order to broadcast point clouds to subscribers. The source code for this first example can be found in the `chapter10_tutorials/src` folder, and it is named `pcl_create.cpp`:

```
#include <ros/ros.h>
#include <pcl/point_cloud.h>
#include <pcl_conversions/pcl_conversions.h>
#include <sensor_msgs/PointCloud2.h>

main (int argc, char **argv)
{
    ros::init (argc, argv, "pcl_create");

    ros::NodeHandle nh;
    ros::Publisher pcl_pub =
        nh.advertise<sensor_msgs::PointCloud2> ("pcl_output", 1);
    pcl::PointCloud<pcl::PointXYZ> cloud;
    sensor_msgs::PointCloud2 output;

    // Fill in the cloud data
    cloud.width = 100;
    cloud.height = 1;
    cloud.points.resize(cloud.width * cloud.height);

    for (size_t i = 0; i < cloud.points.size (); ++i)
    {
        cloud.points[i].x = 1024 * rand () / (RAND_MAX + 1.0f);
        cloud.points[i].y = 1024 * rand () / (RAND_MAX + 1.0f);
        cloud.points[i].z = 1024 * rand () / (RAND_MAX + 1.0f);
    }

    //Convert the cloud to ROS message
    pcl::toROSMsg(cloud, output);
    output.header.frame_id = "odom";

    ros::Rate loop_rate(1);
    while (ros::ok())
    {
        pcl_pub.publish(output);
        ros::spinOnce();
        loop_rate.sleep();
    }
}

return 0;
}
```

The first step in this, and every other snippet, is including the appropriate header files; in this case, we'll include a few PCL-specific headers as well as the standard ROS header and the one that contains the declarations for the `PointCloud2` message:

```
#include <ros/ros.h>
#include <pcl/point_cloud.h>
#include <pcl_conversions/pcl_conversions.h>
#include <sensor_msgs/PointCloud2.h>
```

After the node initialization, a `PointCloud2` ROS publisher is created and advertised; this publisher will later be used to publish the point clouds created through PCL. Once the publisher is created, two

variables are defined. The first one, of the `PointCloud2` type, is the message type that will be used to store the information to be sent through the publisher. The second variable, of the `PointCloud<PointXYZ>` type, is the native PCL type that will be used to generate the point cloud in the first place:

```
ros::Publisher pcl_pub = nh.advertise<sensor_msgs::PointCloud2>
  ("pcl_output", 1);
pcl::PointCloud<pcl::PointXYZ> cloud;
sensor_msgs::PointCloud2 output;
```

The next step is to generate the point cloud with relevant data. In order to do so, we need to allocate the required space in the point cloud structure as well as set the appropriate field. In this case, the point cloud created will be of size `100`. Since this point cloud is not meant to represent an image, the height will only be of size `1`:

```
// Fill in the cloud data
cloud.width = 100;
cloud.height = 1;
cloud.points.resize(cloud.width * cloud.height);
```

With the space allocated and the appropriate fields set, the point cloud is filled with random points between `0` and `1024`:

```
for (size_t i = 0; i < cloud.points.size (); ++i)
{
  cloud.points[i].x = 1024 * rand () / (RAND_MAX + 1.0f);
  cloud.points[i].y = 1024 * rand () / (RAND_MAX + 1.0f);
  cloud.points[i].z = 1024 * rand () / (RAND_MAX + 1.0f);
}
```

At this point, the cloud has been created and filled with data. Since this node is meant to be a data source, the next and last step in this snippet is to convert the PCL point cloud type into a ROS message type and publish it. In order to perform the conversion, the `toROSMsg` function will be used, performing a deep copy of the data from the PCL point cloud type to the `PointCloud2` message.

Finally, the `PointCloud2` message is published periodically at a rate of one hertz in order to have a constant source of information, albeit immutable:

```
//Convert the cloud to ROS message
pcl::toROSMsg(cloud, output);
output.header.frame_id = "odom";

ros::Rate loop_rate(1);
while (ros::ok())
{
  pcl_pub.publish(output);
  ros::spinOnce();
  loop_rate.sleep();
}
```

Perhaps the reader has also noticed that the `frame_id` field in the message header has been set to the `odom` value; this has been done in order to be able to visualize our `PointCloud2` message on the RViz visualizer.

In order to run this example, the first step is to open a terminal and run the `roscore` command:

```
| $ roscore
```

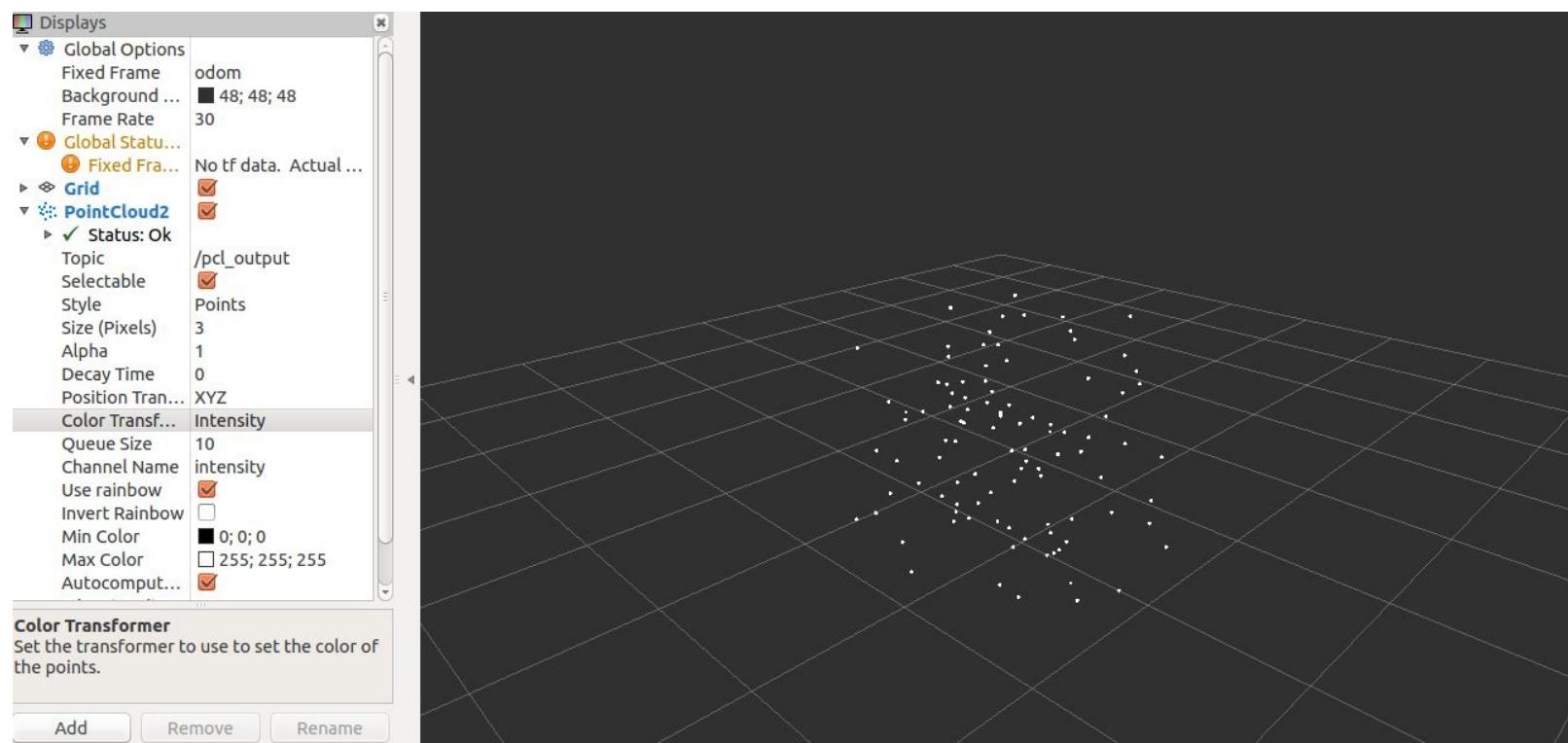
In another terminal, the following command will run the example:

```
| rosrun chapter10_tutorials pcl_create
```

To visualize the point cloud, the RViz visualizer must be run with the following command:

```
| $ rosrun rviz rviz
```

Once `rviz` has been loaded, a `PointCloud2` object needs to be added by clicking on Add and adding the `pcl_output` topic. The reader must make sure to set `odom` as the fixed frame in the Global Options section. If everything has worked properly, a randomly spread point cloud should be shown in the 3D view, just as in the following screenshot:



Loading and saving point clouds to the disk

PCL provides a standard file format to load and store point clouds to the disk as it is a common practice among researchers to share interesting datasets for other people to experiment with. This format is named PCD, and it has been designed to support PCL-specific extensions.

The format is very simple; it starts with a header containing information about the point type and the number of elements in the point cloud, followed by a list of points conforming to the specified type. The following lines are an example of a PCD file header:

```
# .PCD v.5 - Point Cloud Data file format
FIELDS x y z intensity distance sid
SIZE 4 4 4 4 4 4
TYPE F F F F F F
COUNT 1 1 1 1 1 1
WIDTH 460400
HEIGHT 1
POINTS 460400
DATA ascii
```

Reading PCD files can be done through the PCL API, which makes it a very straightforward process. The following example can be found in `chapter10_tutorials/src`, and it is named `pcl_read.cpp`. The following example shows how to load a PCD file and publish the resulting point cloud as a ROS message:

```
#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::NodeHandle nh;
    ros::Publisher pcl_pub =
        nh.advertise<sensor_msgs::PointCloud2> ("pcl_output", 1);

    sensor_msgs::PointCloud2 output;
    pcl::PointCloud<pcl::PointXYZ> cloud;

    pcl::io::loadPCDFile ("test_pcd.pcd", cloud);

    pcl::toROSMsg(cloud, output);
    output.header.frame_id = "odom";

    ros::Rate loop_rate(1);
    while (ros::ok())
    {
        pcl_pub.publish(output);
        ros::spinOnce();
        loop_rate.sleep();
    }

    return 0;
}
```

As always, the first step is to include the required header files. In this particular case, the only new header file that has been added is `pcl/io/pcd_io.h`, which contains the required definitions to load and store point clouds to PCD and other file formats.

The main difference between the previous example and this new one is simply the mechanism used to obtain the point cloud. While in the first example, we manually filled the point cloud with random points, in this case, we just load them from the disk:

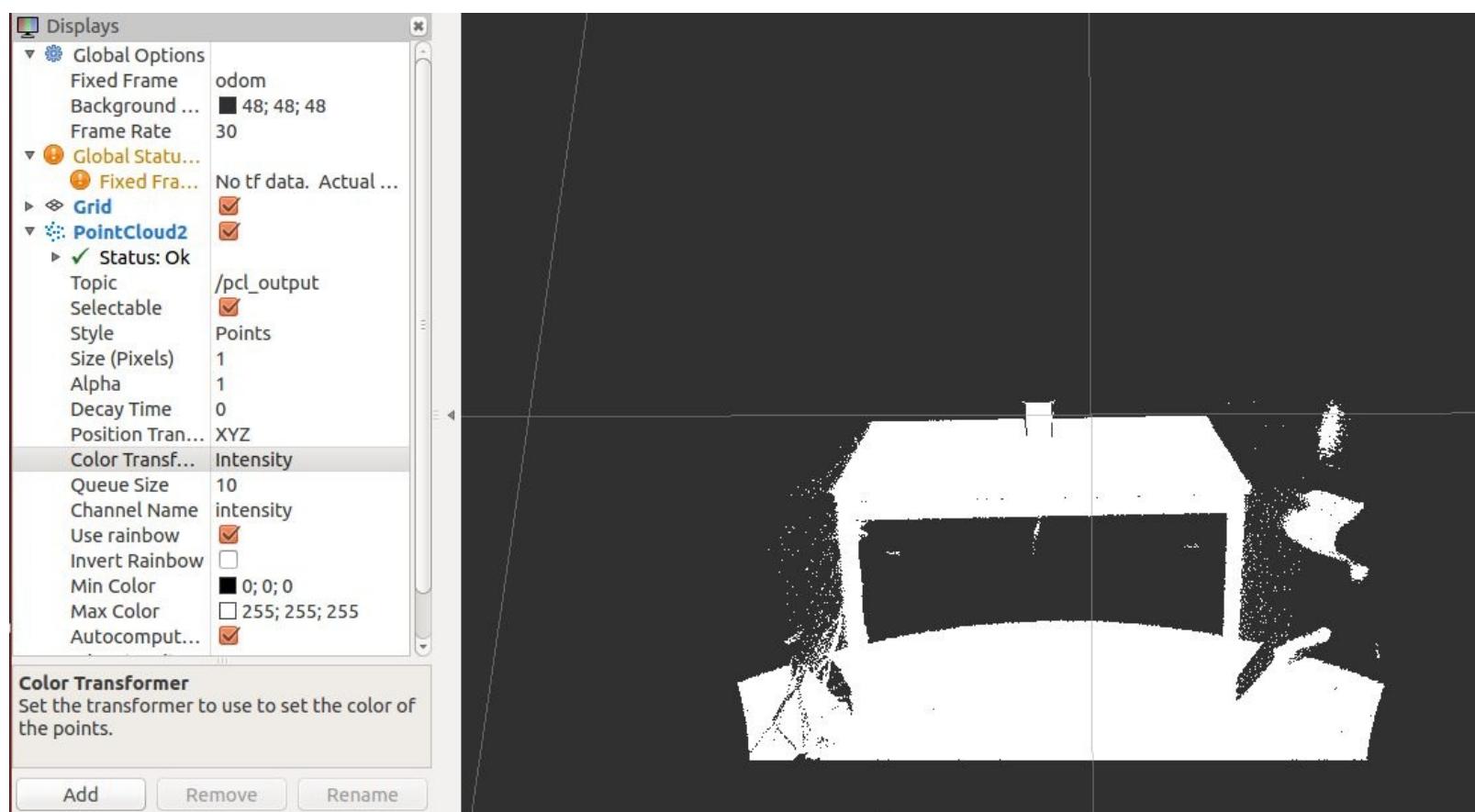
```
| pcl::io::loadPCDFile ("test_pcd.pcd", cloud);
```

As we can see, the process of loading a PCD file has no complexity whatsoever. Further versions of the PCD file format also allow reading and writing of the current origin and orientation of the point cloud.

In order to run the previous example, we need to access the data directory in the package provided, which includes an example PCD file containing a point cloud that will be used further in this chapter:

```
$ roscd chapter10_tutorials/data  
$ rosrun chapter10_tutorials pcl_read
```

As in the previous example, the point cloud can be easily visualized through the RViz visualizer:



Obvious though it may sound, the second interesting operation when dealing with PCD files is creating them. In the following example, our goal is to subscribe to a `sensor_msgs/PointCloud2` topic and store the received point clouds into a file.

The code can be found in `chapter10_tutorials`, and it is named `pcl_write.cpp`:

```
#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>  
  
void cloudCB(const sensor_msgs::PointCloud2 &input)  
{  
    pcl::PointCloud<pcl::PointXYZ> cloud;  
    pcl::fromROSMsg(input, cloud);
```

```

    pcl::io::savePCDFFileASCII ("write_pcd_test.pcd", cloud);
}

main (int argc, char **argv)
{
    ros::init (argc, argv, "pcl_write");

    ros::NodeHandle nh;
    ros::Subscriber bat_sub = nh.subscribe("pcl_output", 10,
                                          cloudCB);

    ros::spin();

    return 0;
}

```

The topic subscribed to is the same used in the two previous examples, namely `pcl_output`, so they can be linked together for testing:

```
| ros::Subscriber bat_sub = nh.subscribe("pcl_output", 10, cloudCB);
```

When a message is received, the `callback` function is called. The first step in this `callback` function is to define a PCL cloud and convert `PointCloud2` that is received, using the `pcl_conversions` function `fromROSMsg`. Finally, the point cloud is saved to the disk in the ASCII format, but it could also be saved in the binary format, which will generate smaller PCD files:

```

void cloudCB(const sensor_msgs::PointCloud2 &input)
{
    pcl::PointCloud<pcl::PointXYZ> cloud;
    pcl::fromROSMsg(input, cloud);
    pcl::io::savePCDFFileASCII ("write_pcd_test.pcd", cloud);
}

```

In order to be able to run this example, it is necessary to have a publisher providing point clouds through the `pcl_output` topic. In this particular case, we will use the `pcl_read` example shown earlier, which fits this requirement. In three different terminals, we will run the `roscore`, `pcl_read`, and `pcl_write` node:

```

$ roscore
$ rosrun chapter10_tutorials chapter10_tutorials pcl_read
$ rosrun chapter10_tutorials chapter10_tutorials pcl_write

```

If everything worked properly, after the first (or second) message is produced, the `pcl_write` node should have created a file named `write_pcd_test.pcd` in the data directory of the `chapter10_tutorials` package.

Visualizing point clouds

PCL provides several ways of visualizing point clouds. The first and simplest is through the basic cloud viewer, which is capable of representing any sort of PCL point cloud in a 3D viewer, while at the same time providing a set of callbacks for user interaction. In the following example, we will create a small node that will subscribe to `sensor_msgs/PointCloud2` and the node will display `sensor_msgs/PointCloud2` using `cloud_viewer (basic)` from the library. The code for this example can be found in the `chapter10_tutorials/src` source directory, and it is named `pcl_visualize.cpp`:

```
#include <iostream>
#include <ros/ros.h>
#include <pcl/visualization/cloud_viewer.h>
#include <sensor_msgs/PointCloud2.h>
#include <pcl_conversions/pcl_conversions.h>

class cloudHandler
{
public:
    cloudHandler()
    : viewer("Cloud Viewer")
    {
        pcl_sub = nh.subscribe("pcl_output", 10,
            &cloudHandler::cloudCB, this);           viewer_timer = nh.createTimer(ros::Duration(0.1),
            &cloudHandler::timerCB, this);
    }

    void cloudCB(const sensor_msgs::PointCloud2 &input)
    {
        pcl::PointCloud<pcl::PointXYZ> cloud;
        pcl::fromROSMsg(input, cloud);

        viewer.showCloud(cloud.makeShared());
    }

    void timerCB(const ros::TimerEvent&)
    {
        if (viewer.wasStopped())
        {
            ros::shutdown();
        }
    }
};

protected:
    ros::NodeHandle nh;
    ros::Subscriber pcl_sub;
    pcl::visualization::CloudViewer viewer;
    ros::Timer viewer_timer;
};

main (int argc, char **argv)
{
    ros::init (argc, argv, "pcl_visualize");

    cloudHandler handler;

    ros::spin();

    return 0;
}
```

The code for this particular example introduces a different pattern; in this case, all of our functionality is encapsulated in a class, which provides a clean way of sharing variables with the `callback` functions, as opposed to using global variables.

The constructor implicitly initializes the node handle through the default constructor, which is automatically called for the missing objects in the initializer list. The cloud handle is explicitly initialized with a very simple string, which corresponds to the window name after everything is correctly initialized. The subscriber to the `pcl_output` topic is set as well as a timer, which will trigger a callback every 100 milliseconds. This timer is used to periodically check whether the window has been closed and, if this is the case, shut down the node:

```
cloudHandler()
: viewer("Cloud Viewer")
{
    pcl_sub = nh.subscribe("pcl_output", 10, &cloudHandler::cloudCB,
    this); viewer_timer = nh.createTimer(ros::Duration(0.1),
    &cloudHandler::timerCB, this);
}
```

The point cloud `callback` function is not very different from the previous examples except that, in this particular case, the PCL point cloud is passed directly to the viewer through the `showCloud` function, which automatically updates the display:

```
void cloudCB(const sensor_msgs::PointCloud2 &input)
{
    pcl::PointCloud<pcl::PointXYZ> cloud;
    pcl::fromROSMsg(input, cloud);

    viewer.showCloud(cloud.makeShared());
}
```

Since the viewer window usually comes with a close button as well as a keyboard shortcut to close the window, it is important to take into account this event and act upon it by, for example, shutting down the node. In this particular case, we are handling the current state of the window in a callback, which is called through a ROS timer every 100 milliseconds. If the viewer has been closed, our action is to simply shut down the node:

```
void timerCB(const ros::TimerEvent&)
{
    if (viewer.wasStopped())
    {
        ros::shutdown();
    }
}
```

To execute this example, and any other for that matter, the first step is to run the `roscore` command in a terminal:

```
| $ roscore
```

In a second terminal, we will run the `pcl_read` example and a source of data, such as a reminder, using the following commands:

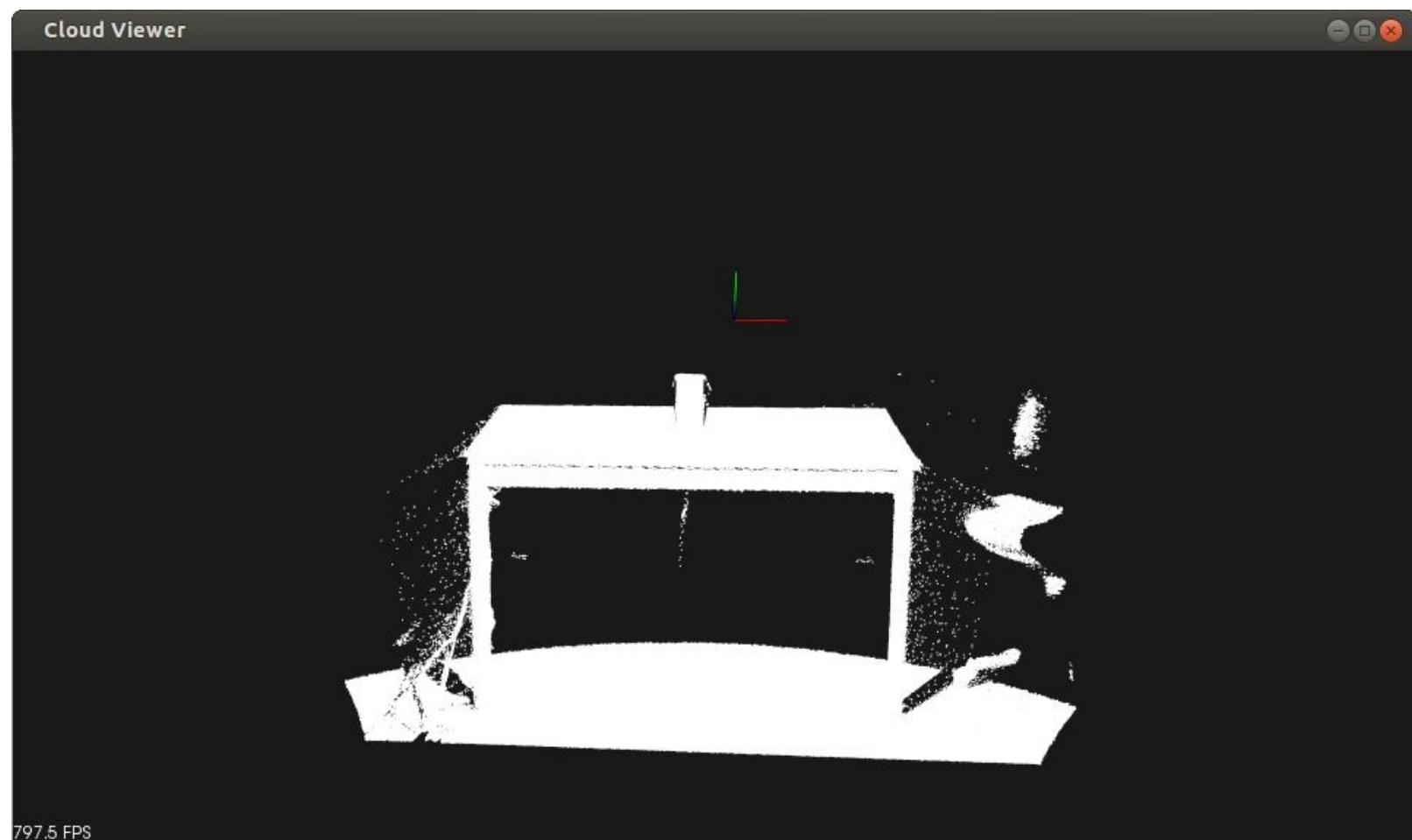
```
| $ roscd chapter10_tutorials/data
| $ rosrun chapter10_tutorials pcl_read
```

Finally, in a third terminal, we will run the following command:

```
| $ rosrun chapter10_tutorials pcl_visualize
```

Running this code will cause a window to launch; this window will display the point cloud contained in

the test PCD file provided with the examples. The following screenshot shows this:



The current example uses the simplest possible viewer, namely the PCL `cloud_viewer`, but the library also provides a much more complex and complete visualization component named **PCLVisualizer**. This visualizer is capable of displaying point clouds, meshes, and surfaces, as well as including multiple viewports and color spaces. An example of how to use this particular visualizer is provided in the `chapter10_tutorials` source directory named `pcl_visualize2.cpp`.

In general, all the visualizers provided by PCL use the same underlying functionality and work in much the same way. The mouse can be used to move around the 3D view; in combination with the shift, it allows you to translate the image, and in combination with the control, it allows you to rotate the image. Finally, upon pressing `H`, the help text is printed in the current terminal, which should look like the following screenshot:

```
$ rosrun chapter6_tutorials pcl_visualize
| Help:
-----
p, P    : switch to a point-based representation
w, W    : switch to a wireframe-based representation (where available)
s, S    : switch to a surface-based representation (where available)

j, J    : take a .PNG snapshot of the current window view
c, C    : display current camera/window parameters
f, F    : fly to point mode

e, E    : exit the interactor
q, Q    : stop and call VTK's TerminateApp

+/-    : increment/decrement overall point size
+/- [+ ALT] : zoom in/out

g, G    : display scale grid (on/off)
u, U    : display lookup table (on/off)

r, R [+ ALT] : reset camera [to viewpoint = {0, 0, 0} -> center_{x, y, z}]

ALT + s, S    : turn stereo mode on/off
ALT + f, F    : switch between maximized window mode and original size

l, L          : list all available geometric and color handlers for the current actor map
ALT + 0..9 [+ CTRL] : switch between different geometric handlers (where available)
0..9 [+ CTRL]  : switch between different color handlers (where available)

SHIFT + left click    : select a point

x, X    : toggle rubber band selection mode for left mouse button
```

Filtering and downsampling

The two main issues that we may face when attempting to process point clouds are excessive noise and excessive density. The former causes our algorithms to misinterpret the data and produce incorrect or inaccurate results, whereas the latter makes our algorithms take a long time to complete their operation. In this section, we will provide insight into how to reduce the amount of noise or outliers of our point clouds and how to reduce the point density without losing valuable information.

The first part is to create a node that will take care of filtering outliers from the point clouds produced in the `pcl_output` topic and sending them back through the `pcl_filtered` topic. The example can be found in the source directory of the `chapter10_tutorials` package, and it is named `pcl_filter.cpp`:

```
#include <ros/ros.h>
#include <pcl/point_cloud.h>
#include <pcl_conversions/pcl_conversions.h>
#include <sensor_msgs/PointCloud2.h>
#include <pcl/filters/statistical_outlier_removal.h>

class cloudHandler
{
public:
    cloudHandler()
    {
        pcl_sub = nh.subscribe("pcl_output", 10,
            &cloudHandler::cloudCB, this);           pcl_pub =
        nh.advertise<sensor_msgs::PointCloud2>("pcl_filtered", 1);
    }

    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);

        pcl::StatisticalOutlierRemoval<pcl::PointXYZ> statFilter;
        statFilter.setInputCloud(cloud.makeShared());
        statFilter.setMeanK(10);
        statFilter.setStddevMulThresh(0.2);
        statFilter.filter(cloud_filtered);

        pcl::toROSMsg(cloud_filtered, output);
        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");
    cloudHandler handler;
    ros::spin();
    return 0;
}
```

Just as with the previous example, this one uses a class that contains a publisher as a member variable

that is used in the `callback` function. The `callback` function defines two PCL point clouds, one for input messages and one for the filtered point cloud. As always, the input point cloud is converted using the standard conversion functions:

```
pcl::PointCloud<pcl::PointXYZ> cloud;
pcl::PointCloud<pcl::PointXYZ> cloud_filtered;
sensor_msgs::PointCloud2 output;

pcl::fromROSMsg(input, cloud);
```

Now, this is where things start getting interesting. In order to perform filtering, we will use the statistical outlier removal algorithm provided by PCL. This algorithm performs an analysis of the point cloud and removes those points that do not satisfy a specific statistical property, which, in this case, is the average distance in a neighborhood, removing all of those points that deviate too much from the average. The number of neighbors to use for the average computation can be set by the `setMeanK` function, and the multiplier on the standard deviation threshold can also be set through `setStddevMulThresh`.

The following piece of code handles the filtering and sets the `cloud_filtered` point cloud with our new *noiseless* cloud:

```
pcl::StatisticalOutlierRemoval<pcl::PointXYZ> statFilter;
statFilter.setInputCloud(cloud.makeShared());
statFilter.setMeanK(10);
statFilter.setStddevMulThresh(0.2);
statFilter.filter(cloud_filtered);
```

Finally, and as always, the filtered cloud is converted to `PointCloud2` and published so that our other algorithms can make use of this new point cloud to provide more accurate results:

```
pcl::toROSMsg (cloud_filtered, output);
pcl_pub.publish(output);
```

In the following screenshot, we can see the result of the previous code when it is applied on the point cloud provided in our test PCD file. The original point cloud can be seen on the left-hand side and the filtered one on the right-hand side. The results are not perfect, but we can observe how much of the noise has been removed, which means that we can now proceed with reducing the density of the filtered point cloud:



Reducing the density of a point cloud, or any other dataset for that matter, is called **downsampling**. There are several techniques that can be used to downsample a point cloud, but some of them are more rigorous or provide better results than others.

In general, the goal of downsampling a point cloud is to improve the performance of our algorithms; for that reason, we need our downsampling algorithms to keep the basic properties and structure of our point cloud so that the end result of our algorithms doesn't change too much.

In the following example, we are going to demonstrate how to perform downsampling on point clouds with **Voxel Grid Filter**. In this case, the input point clouds are going to be the filtered ones from the previous example so that we can chain both examples together to produce better results in further algorithms. The example can be found in the source directory of the `chapter10_tutorials` package, and it's named `pcl_downsampling.cpp`:

```
#include <ros/ros.h>
#include <pcl/point_cloud.h>
#include <pcl_conversions/pcl_conversions.h>
#include <sensor_msgs/PointCloud2.h>
#include <pcl/filters/voxel_grid.h>

class cloudHandler
{
public:
    cloudHandler()
    {
        pcl_sub = nh.subscribe("pcl_filtered", 10,
            &cloudHandler::cloudCB, this);          pcl_pub =
        nh.advertise<sensor_msgs::PointCloud2>("pcl_downsampled",
        1);
    }

    void cloudCB(const sensor_msgs::PointCloud2 &input)
    {
        pcl::PointCloud<pcl::PointXYZ> cloud;
        pcl::PointCloud<pcl::PointXYZ> cloud_downsampled;
        sensor_msgs::PointCloud2 output;

        pcl::fromROSMsg(input, cloud);

        pcl::VoxelGrid<pcl::PointXYZ> voxelSampler;
        voxelSampler.setInputCloud(cloud.makeShared());
        voxelSampler.setLeafSize(0.01f, 0.01f, 0.01f);
        voxelSampler.filter(cloud_downsampled);

        pcl::toROSMsg(cloud_downsampled, output);
        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_downsampling");

    cloudHandler handler;

    ros::spin();

    return 0;
}
```

This example is exactly the same as the previous one, with the only differences being the topics subscribed and published, which, in this case, are `pcl_filtered` and `pcl_downsampled`, and the algorithms used to perform the filtering on the point cloud.

As said earlier, the algorithm used is VoxelGrid Filter, which partitions the point cloud into voxels, or

more accurately a 3D grid, and replaces all of the points contained in each voxel with the centroid of that subcloud. The size of each voxel can be specified through `setLeafsize` and will determine the density of our point cloud:

```
| pcl::VoxelGrid<pcl::PointXYZ> voxelSampler;  
| voxelSampler.setInputCloud(cloud.makeShared());  
| voxelSampler.setLeafSize(0.01f, 0.01f, 0.01f);  
| voxelSampler.filter(cloud_downsampled);
```

The following image shows the results of both the filtered and downsampled images when compared to the original one. You can appreciate how the structure has been kept, the density reduced, and much of the noise completely eliminated:



To execute both examples, as always we start running `roscore`:

```
| $ roscore
```

In the second terminal, we will run the `pcl_read` example and a source of data:

```
| $ roscd chapter10_tutorials/data  
| $ rosrun chapter10_tutorials pcl_read
```

In the third terminal, we will run the filtering example, which will produce the `pcl_filtered` image for the downsampling example:

```
| $ rosrun chapter10_tutorials pcl_filter
```

Finally, in the fourth terminal, we will run the downsampling example:

```
| $ rosrun chapter10_tutorials pcl_downsampling
```

As always, the results can be seen on `rviz`, but in this case, the `pcl_visualizer2` example provided in the package can also be used although you might need to tweak the subscribed topics.

Registration and matching

Registration and matching is a common technique used in several different fields that consists of finding common structures or features in two datasets and using them to stitch the datasets together. In the case of point cloud processing, this can be achieved as easily as finding where one point cloud ends and where the other one starts. These techniques are very useful when obtaining point clouds from moving sources at a high rate, and we have an estimate of the movement of the source. With this algorithm, we can stitch each of those point clouds together and reduce the uncertainty in our sensor pose estimation.

PCL provides an algorithm named **Iterative Closest Point (ICP)** to perform registration and matching. We will use this algorithm in the following example, which can be found in the source directory of the chapter10_tutorials package, and it's named `pcl_matching.cpp`:

```
#include <ros/ros.h>
#include <pcl/point_cloud.h>
#include <pcl/registration/icp.h>
#include <pcl_conversions/pcl_conversions.h>
#include <sensor_msgs/PointCloud2.h>

class cloudHandler
{
public:
    cloudHandler()
    {
        pcl_sub = nh.subscribe("pcl_downsampled", 10,
            &cloudHandler::cloudCB, this);           pcl_pub =
        nh.advertise<sensor_msgs::PointCloud2>("pcl_matched", 1);
    }

    void cloudCB(const sensor_msgs::PointCloud2 &input)
    {
        pcl::PointCloud<pcl::PointXYZ> cloud_in;
        pcl::PointCloud<pcl::PointXYZ> cloud_out;
        pcl::PointCloud<pcl::PointXYZ> cloud_aligned;
        sensor_msgs::PointCloud2 output;

        pcl::fromROSMsg(input, cloud_in);

        cloud_out = cloud_in;

        for (size_t i = 0; i < cloud_in.points.size (); ++i)
        {
            cloud_out.points[i].x = cloud_in.points[i].x + 0.7f;
        }

        pcl::IterativeClosestPoint<pcl::PointXYZ, pcl::PointXYZ>
        icp;
        icp.setInputSource(cloud_in.makeShared());
        icp.setInputTarget(cloud_out.makeShared());

        icp.setMaxCorrespondenceDistance(5);
        icp.setMaximumIterations(100);
        icp.setTransformationEpsilon (1e-12);
        icp.setEuclideanFitnessEpsilon(0.1);

        icp.align(cloud_aligned);

        pcl::toROSMsg(cloud_aligned, output);
        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_matching");
    cloudHandler handler;
    ros::spin();
    return 0;
}

```

This example uses the `pcl_downsampled` topic as the input source of point clouds in order to improve the performance of the algorithm; the end result is published in the `pcl_matched` topic. The algorithm used for registration and matching takes three-point clouds—the first one is the point cloud to transform, the second one is the fixed cloud to which the first one should be aligned, and the third one is the end result point cloud:

```

pcl::PointCloud<pcl::PointXYZ> cloud_in;
pcl::PointCloud<pcl::PointXYZ> cloud_out;
pcl::PointCloud<pcl::PointXYZ> cloud_aligned;

```

To simplify matters and since we don't have a continuous source of point clouds, we are going to use the same original point cloud as the fixed cloud but displaced on the x axis. The expected behavior of the algorithm would then be to align both point clouds together:

```

cloud_out = cloud_in;
for (size_t i = 0; i < cloud_in.points.size (); ++i)
{
    cloud_out.points[i].x = cloud_in.points[i].x + 0.7f;
}

```

The next step is to call the ICP algorithm to perform the registration and matching. This iterative algorithm uses **Singular Value Decomposition (SVD)** to calculate the transformations to be done on the input point cloud towards decreasing the gap to the fixed point cloud. The algorithm has three basic stopping conditions:

- The difference between the previous and current transformations is smaller than a certain threshold. This threshold can be set through the `setTransformationEpsilon` function.
- The number of iterations has reached the maximum set by the user. This maximum can be set through the `setMaximumIterations` function.
- Finally, the sum of the Euclidean squared errors between two consecutive steps in the loop is below a certain threshold. This specific threshold can be set through the `setEuclideanFitnessEpsilon` function.

Another interesting parameter that is used to improve the accuracy of the result is the correspondence distance, which can be set through the `setMaxCorrespondanceDistance` function. This parameter defines the minimum distance that two correspondent points need to have between them to be considered in the alignment process.

With all of these parameters, the fixed point cloud and the input point cloud, the algorithm is capable of performing the registration and matching and returning the end result point cloud after the iterative transformations:

```
| pcl::IterativeClosestPoint<pcl::PointXYZ, pcl::PointXYZ> icp;
```

```
| icp.setInputSource(cloud_in.makeShared());
| icp.setInputTarget(cloud_out.makeShared());
| icp.setMaxCorrespondenceDistance(5);
| icp.setMaximumIterations(100);
| icp.setTransformationEpsilon (1e-12);
| icp.setEuclideanFitnessEpsilon(0.1);
| icp.align(cloud_aligned);
```

Finally, the resulting point cloud is converted into `PointCloud2` and published through the corresponding topic:

```
| pcl::toROSMsg(cloud_aligned, output);
| pcl_pub.publish(output);
```

In order to run this example, we need to follow the same instructions as the filtering and downsampling example, starting with `roscore` in one Terminal:

```
| $ roscore
```

In a second Terminal, we will run the `pcl_read` example and a source of data:

```
| $ roscd chapter10_tutorials/data
| $ rosrun chapter10_tutorials pcl_read
```

In a third Terminal, we will run the filtering example:

```
| $ rosrun chapter10_tutorials pcl_filter
```

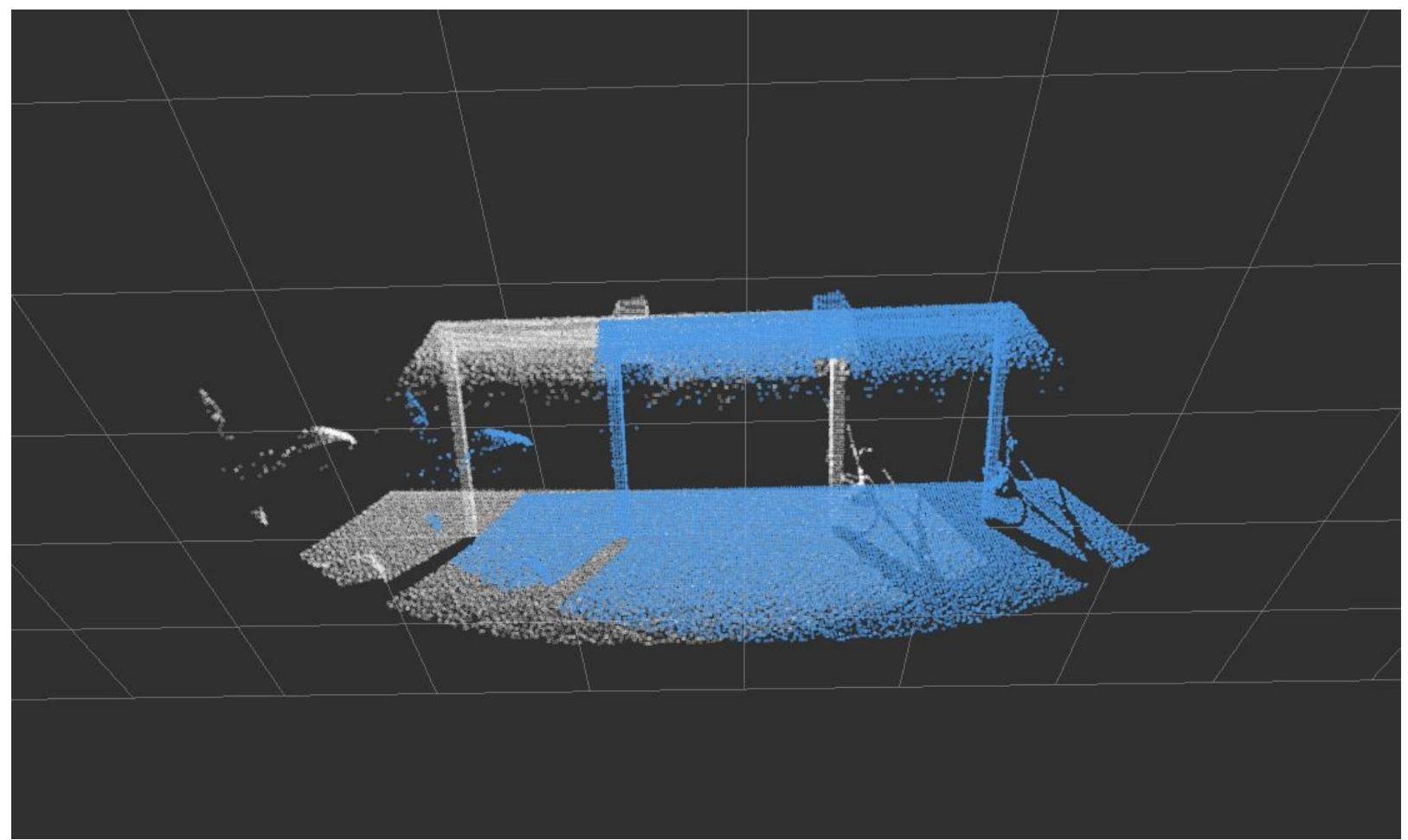
In a fourth Terminal, we will run the downsampling example:

```
| $ rosrun chapter10_tutorials pcl_downsampling
```

Finally, we will run the registration and matching node that requires the `pcl_downsampled` topic, which is produced by the chain of nodes run before:

```
| $ rosrun chapter10_tutorials pcl_matching
```

The end result can be seen in the following image, which has been obtained from `rviz`. The blue one is the original point cloud obtained from the PCD file, and the white point cloud is the aligned one obtained from the ICP algorithm. It has to be noted that the original point cloud was translated in the `x` axis, so the results are consistent with the point cloud, completely overlapping the translated image, as shown in the following screenshot:



Partitioning point clouds

Often times, when processing our point clouds, we might need to perform operations that require accessing a local region of a point cloud or manipulating the neighborhood of specific points. Since point clouds store data in a one-dimensional data structure, these kinds of operations are inherently complex. In order to solve this issue, PCL provides two spatial data structures, named the **kd-tree** and the **octree**, which can provide an alternative and more structured representation of any point cloud.

As the name suggests, an octree is basically a tree structure in which each node has eight children, and which can be used to partition the 3D space. In contrast, the kd-tree is a binary tree in which nodes represent k-dimensional points. Both data structures are very interesting, but, in this particular example, we are going to learn how to use the octree to search and retrieve all the points surrounding a specific point. The example can be found in the source directory of the `chapter10_tutorials` package, and it's named `pcl_partitioning.cpp`:

```
#include <ros/ros.h>
#include <pcl/point_cloud.h>
#include <pcl_conversions/pcl_conversions.h>
#include <sensor_msgs/PointCloud2.h>
#include <pcl/octree/octree.h>

class cloudHandler
{
public:
    cloudHandler()
    {
        pcl_sub = nh.subscribe("pcl_downsampled", 10,
            &cloudHandler::cloudCB, this); pcl_pub =
        nh.advertise<sensor_msgs::PointCloud2>("pcl_partitioned",
        1);
    }

    void cloudCB(const sensor_msgs::PointCloud2 &input)
    {
        pcl::PointCloud<pcl::PointXYZ> cloud;
        pcl::PointCloud<pcl::PointXYZ> cloud_partitioned;
        sensor_msgs::PointCloud2 output;

        pcl::fromROSMsg(input, cloud);

        float resolution = 128.0f;
        pcl::octree::OctreePointCloudSearch<pcl::PointXYZ> octree
        (resolution);

        octree.setInputCloud (cloud.makeShared());
        octree.addPointsFromInputCloud ();

        pcl::PointXYZ center_point;
        center_point.x = 0 ;
        center_point.y = 0.4;
        center_point.z = -1.4;

        float radius = 0.5;
        std::vector<int> radiusIdx;
        std::vector<float> radiusSQDist;
        if (octree.radiusSearch (center_point, radius, radiusIdx,
        radiusSQDist) > 0)
        {
            for (size_t i = 0; i < radiusIdx.size (); ++i)
            {
                cloud_partitioned.points.push_back
                (cloud.points[radiusIdx[i]]);
            }
        }
    }
}
```

```

    }

    pcl::toROSMsg(cloud_partitioned, output);
    output.header.frame_id = "odom";
    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_partitioning");

    cloudHandler handler;

    ros::spin();

    return 0;
}

```

As usual, this example uses the `pcl_downsampled` topic as an input source of point clouds and publishes the resulting partitioned point cloud to the `pcl_partitioned` topic. The handler function starts by converting the input point cloud to a PCL point cloud. The next step is to create an octree-searching algorithm, which requires passing a resolution value that will determine the size of the voxels at the lowest level of the tree and, consequently, other properties such as the tree's depth. The algorithm also requires to be given the point cloud to explicitly load the points:

```

float resolution = 128.0f;
pcl::octree::OctreePointCloudSearch<pcl::PointXYZ>
    octree(resolution);

octree.setInputCloud (cloud.makeShared());
octree.addPointsFromInputCloud ();

```

The next step is to define the center point of the partition; in this case, it has been handpicked to be close to the top of the point cloud:

```

pcl::PointXYZ center_point;
center_point.x = 0;
center_point.y = 0.4;
center_point.z = -1.4;

```

We can now perform a search in a radius around that specific point using the `radiusSearch` function from the octree search algorithm. This particular function is used to output arguments that return the indices of the points that fall in that radius and the squared distance from those points to the center point provided. With those indices, we can then create a new point cloud containing only the points belonging to the partition:

```

float radius = 0.5;
std::vector<int> radiusIdx;
std::vector<float> radiusSQDist;
if (octree.radiusSearch (center_point, radius, radiusIdx,
    radiusSQDist) > 0)
{
    for (size_t i = 0; i < radiusIdx.size (); ++i)
    {
        cloud_partitioned.points.push_back
            (cloud.points[radiusIdx[i]]);
    }
}

```

Finally, the point cloud is converted to the `Pointcloud2` message type and published in the output topic:

```
| pcl::toROSMsg( cloud_partitioned, output );
| output.header.frame_id = "odom";
| pcl_pub.publish( output );
```

In order to run this example, we need to run the usual chain of nodes, starting with `roscore`:

```
| $ roscore
```

In the second terminal, we can run the `pcl_read` example and a source of data:

```
| $ roscd chapter10_tutorials/data
| $ rosrun chapter10_tutorials pcl_read
```

In the third terminal, we will run the filtering example:

```
| $ rosrun chapter10_tutorials pcl_filter
```

In the fourth terminal, we will run the downsampling example:

```
| $ rosrun chapter10_tutorials pcl_downsampling
```

Finally, we will run this example:

```
| $ rosrun chapter10_tutorials pcl_partitioning
```

In the following image, we can see the end result of the partitioning process. Since we handpicked the point to be close to the top of the point cloud, we managed to extract part of the cup and the table. This example only shows a tiny fraction of the potential of the octree data structure, but it's a good starting point to further your understanding:



Segmentation

Segmentation is the process of partitioning a dataset into different blocks of data satisfying certain criteria. The segmentation process can be done in many different ways and with varied criteria; sometimes, it may involve extracting structured information from a point cloud based on a statistical property, and in other cases, it can simply require extracting points in a specific color range.

In many cases, our data might fit a specific mathematical model, such as a plane, line, or sphere, among others. When this is the case, it is possible to use a model estimation algorithm to calculate the parameters for the model that fits our data. With those parameters, it is then possible to extract the points belonging to that model and evaluate how well they fit it.

In this example, we are going to show how to perform model-based segmentation of a point cloud. We are going to constrain ourselves to a planar model, which is one of the most common mathematical models you can usually fit to a point cloud. For this example, we will also perform the model estimation using a widespread algorithm named **Random Sample Consensus (RANSAC)**, which is an iterative algorithm capable of performing accurate estimations even in the presence of outliers.

The example code can be found in the `chapter10_tutorials` package, and its named `pcl_planar_segmentation.cpp`:

```
#include <ros/ros.h>
#include <pcl/point_cloud.h>
#include <pcl/conversions/pcl_conversions.h>
#include <pcl/ModelCoefficients.h>
#include <pcl/sample_consensus/method_types.h>
#include <pcl/sample_consensus/model_types.h>
#include <pcl/segmentation/sac_segmentation.h>
#include <pcl/filters/extract_indices.h>
#include <sensor_msgs/PointCloud2.h>

class cloudHandler
{
public:
    cloudHandler()
    {
        pcl_sub = nh.subscribe("pcl_downsampled", 10,
            &cloudHandler::cloudCB, this);           pcl_pub =
        nh.advertise<sensor_msgs::PointCloud2>("pcl_segmented",
        1);           ind_pub =
        nh.advertise<pcl_msgs::PointIndices>("point_indices", 1);      coef_pub =
        nh.advertise<pcl_msgs::ModelCoefficients>("planar_coef",
        1);
    }

    void cloudCB(const sensor_msgs::PointCloud2 &input)
    {
        pcl::PointCloud<pcl::PointXYZ> cloud;
        pcl::PointCloud<pcl::PointXYZ> cloud_segmented;

        pcl::fromROSMsg(input, cloud);

        pcl::ModelCoefficients coefficients;
        pcl::PointIndices::Ptr inliers(new pcl::PointIndices());

        pcl::SACSegmentation<pcl::PointXYZ> segmentation;
        segmentation.setModelType(pcl::SACMODEL_PLANE);
        segmentation.setMethodType(pcl::SAC_RANSAC);
        segmentation.setMaxIterations(1000);
        segmentation.setDistanceThreshold(0.01);
        segmentation.setInputCloud(cloud.makeShared());
    }
}
```

```

segmentation.segment(*inliers, coefficients);

pcl_msgs::ModelCoefficients ros_coefficients;
pcl_conversions::fromPCL(coefficients, ros_coefficients);
ros_coefficients.header.stamp = input.header.stamp;
coef_pub.publish(ros_coefficients);

pcl_msgs::PointIndices ros_inliers;
pcl_conversions::fromPCL(*inliers, ros_inliers);
ros_inliers.header.stamp = input.header.stamp;
ind_pub.publish(ros_inliers);

pcl::ExtractIndices<pcl::PointXYZ> extract;
extract.setInputCloud(cloud.makeShared());
extract.setIndices(inliers);
extract.setNegative(false);
extract.filter(cloud_segmented);

sensor_msgs::PointCloud2 output;
pcl::toROSMsg(cloud_segmented, output);
pcl_pub.publish(output);
}

protected:
ros::NodeHandle nh;
ros::Subscriber pcl_sub;
ros::Publisher pcl_pub, ind_pub, coef_pub;
};

main(int argc, char **argv)
{
    ros::init(argc, argv, "pcl_planar_segmentation");

    cloudHandler handler;

    ros::spin();

    return 0;
}

```

As the reader might have noticed, two new message types are being used in the advertised topics. As their names suggest, the `ModelCoefficients` messages store the coefficients of a mathematical model, and `PointIndices` stores the indices of the points of a point cloud. We will publish these as an alternative way of representing the extracted information, which could then be used in combination with the original point cloud (`pcl_downsampled`) to extract the correct point. As a hint, this can be done by setting the timestamp of the published objects to the same timestamp of the original point cloud message and using ROS message filters:

```

pcl_pub = nh.advertise<sensor_msgs::PointCloud2>("pcl_segmented",
1); ind_pub = nh.advertise<pcl_msgs::PointIndices>("point_indices",
1); coef_pub =
nh.advertise<pcl_msgs::ModelCoefficients>("planar_coef", 1);

```

As always, in the `callback` function, we perform the conversion from the `PointCloud2` message to the point cloud type. In this case, we also define two new objects that correspond to the native `ModelCoefficients` and `PointIndices` types, which will be used by the segmentation algorithm:

```

pcl::PointCloud<pcl::PointXYZ> cloud;
pcl::PointCloud<pcl::PointXYZ> cloud_segmented;

pcl::fromROSMsg(input, cloud);

pcl::ModelCoefficients coefficients;
pcl::PointIndices::Ptr inliers(new pcl::PointIndices());

```

The segmentation algorithm lets us define `ModelType` and `MethodType`, with the former being the mathematical model we are looking to fit and the latter being the algorithm to use. As we explained before, we are

using RANSAC due to its robustness against outliers. The algorithm also lets us define the two stopping criteria: the maximum number of iterations (`setMaxIterations`) and the maximum distance to the model (`setDistanceThreshold`). With those parameters set, plus the input point cloud, the algorithm can then be performed, returning the inliers (points which fall in the model) and the coefficients of the model:

```
pcl::SACSegmentation<pcl::PointXYZ> segmentation;
segmentation.setModelType(pcl::SACMODEL_PLANE);
segmentation.setMethodType(pcl::SAC_RANSAC);
segmentation.setMaxIterations(1000);

segmentation.setDistanceThreshold(0.01);
segmentation.setInputCloud(cloud.makeShared());
segmentation.segment(*inliers, coefficients);
```

Our next step is to convert and publish the inliers and the model coefficients. As usual, conversions are performed with the standard functions, but you might notice that the namespace and signature of the conversion function is different from the one being used for point cloud conversions. To further improve this example, these messages also include the timestamp of the original point cloud in order to link them together. This also allows the use of the ROS message filters on other nodes to create callbacks containing objects that are linked together:

```
pcl_msgs::ModelCoefficients ros_coefficients;
pcl_conversions::fromPCL(coefficients, ros_coefficients);
ros_coefficients.header.stamp = input.header.stamp;
coef_pub.publish(ros_coefficients);

pcl_msgs::PointIndices ros_inliers;
pcl_conversions::fromPCL(*inliers, ros_inliers);
ros_inliers.header.stamp = input.header.stamp;
ind_pub.publish(ros_inliers);
```

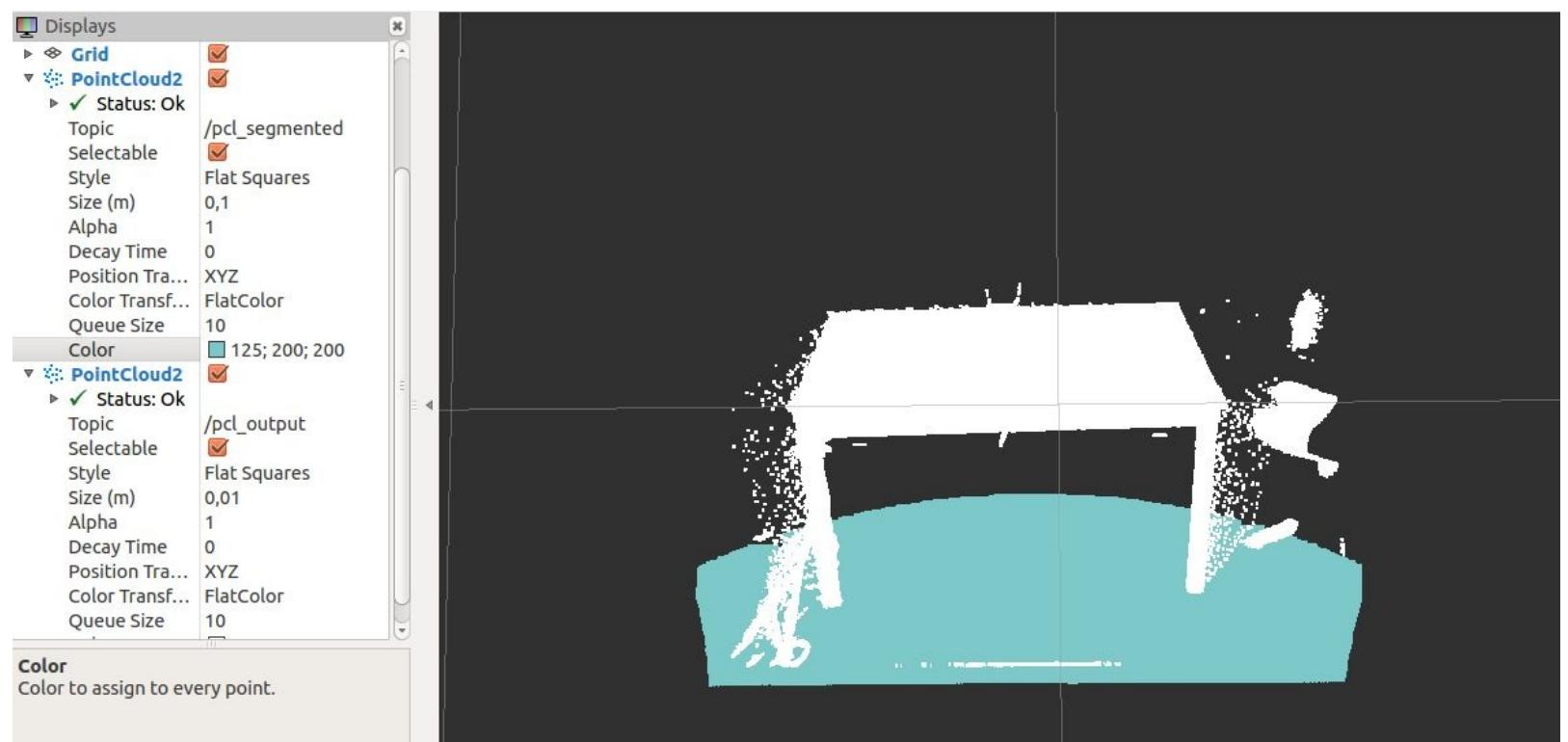
In order to create the segmented point cloud, we extract the inliers from the point cloud. The easiest way to do this is with the `ExtractIndices` object, but it could be easily done by simply looping through the indices and pushing the corresponding points into a new point cloud:

```
pcl::ExtractIndices<pcl::PointXYZ> extract;
extract.setInputCloud(cloud.makeShared());
extract.setIndices(inliers);
extract.setNegative(false);
extract.filter(cloud_segmented);
```

Finally, we convert the segmented point cloud into a `PointCloud2` message type and we publish it:

```
sensor_msgs::PointCloud2 output;
pcl::toROSMsg (cloud_segmented, output);
pcl_pub.publish(output)
```

The result can be seen in the following image; the original point cloud is represented in white and the segmented inliers are represented in bluish. In this particular case, the floor was extracted as it's the biggest flat surface. This is quite convenient as it is probably one of the main elements we will usually want to extract from our point clouds:



Summary

In this chapter, we have explored the different tools, algorithms, and interfaces that can be used to work with point clouds in ROS. The reader might have noticed that we have tried to link the examples together to provide more insight into how these kinds of nodes might be used in a reusable manner. In any case, given the computational price of point cloud processing, any kind of architectural design will be inextricably linked to the computational capabilities of the system at hand.

The data flow of our examples should start with all the data producers, which are `pcl_create` and `pcl_read`. It should continue to the data filters, which are `pcl_filter` and `pcl_downsampling`. After the filtering is performed, more complex information can be extracted through `pcl_planar_segmentation`, `pcl_partitioning` and `pcl_matching`. Finally, the data can be written to disk through `pcl_write` or visualized through `pcl_visualize`.

The main objective of this particular chapter was to provide clear and concise examples of how to integrate the basic capabilities of the PCL library with ROS, something which can be limited to messages and conversion functions. In order to accomplish this goal, we have taken the liberty of also explaining the basic techniques and common algorithms used to perform data processing on point clouds as we are aware of the growing importance of this kind of knowledge.

Mastering ROS for Robotics Programming

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

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:

```
| &lt;link name="&lt;name of the link>">  
|   &lt;inertial>.....&lt;/inertial>  
|     &lt;visual> .....&lt;/visual>  
|     &lt;collision>.....&lt;/collision>  
&lt;/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.

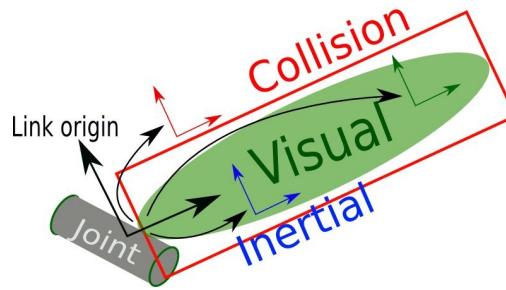


Figure 1 : Visualization of a URDF link

- `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:

```
| &lt;joint name="&lt;name of the joint>">  
|   &lt;parent link="link1"/>  
|   &lt;child link="link2"/>
```

```
&lt;calibration .... />
&lt;dynamics damping .... />
&lt;limit effort .... />
&lt;/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:

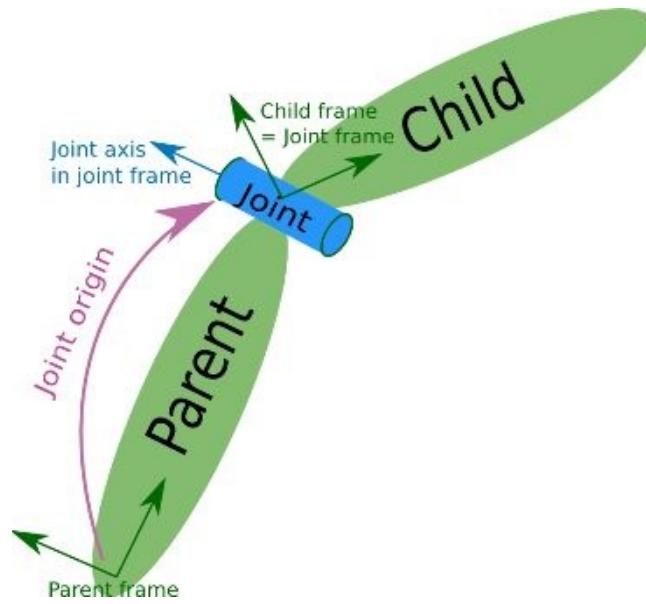


Figure 2 : Visualization of a URDF joint

- **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:

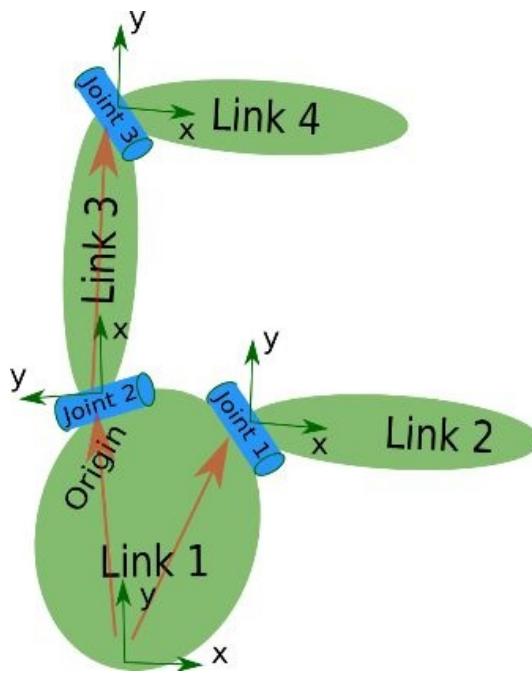


Figure 3 : Visualization of a robot model having joints and links

- `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:

```
&lt;gazebo reference="link_1">
  &lt;material>Gazebo/Black&lt;/material>
&lt;/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 section'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.

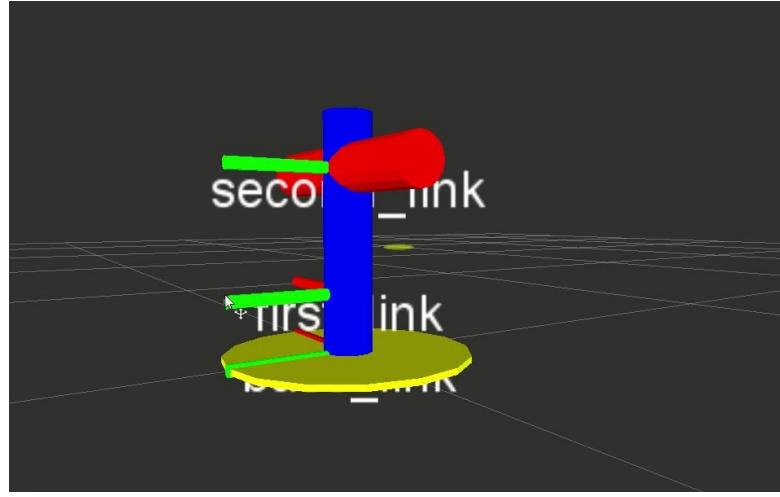


Figure 4 : Visualization of a pan and tilt mechanism in RViz

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:

```
&lt;?xml version="1.0"?>
&lt;robot name="pan_tilt">

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

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

  &lt;link name="pan_link">
    &lt;visual>
      &lt;geometry>
        &lt;cylinder length="0.4" radius="0.04"/>
      &lt;/geometry>
    &lt;/visual>
  &lt;/link>
```

```
&lt;/geometry>
&lt;origin rpy="0 0 0" xyz="0 0 0.09"/>
&lt;material name="red">
  &lt;color rgba="0 0 1 1"/>
  &lt;/material>
&lt;/visual>
&lt;/link>

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

&lt;link name="tilt_link">
  &lt;visual>
    &lt;geometry>
      &lt;cylinder length="0.4" radius="0.04"/>
    &lt;/geometry>
    &lt;origin rpy="0 1.5 0" xyz="0 0 0"/>
    &lt;material name="green">
      &lt;color rgba="1 0 0 1"/>
    &lt;/material>
  &lt;/visual>
  &lt;/link>
&lt;/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:

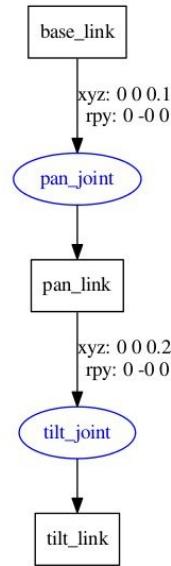


Figure 5 : Graph of joint and links in pan and tilt mechanism

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:

```
&lt;launch>
  &lt;arg name="model" />
  &lt;param name="robot_description" textfile="$(find mastering_ros_robot_description_pkg)/urdf/pan_tilt.urdf" />
  &lt;param name="use_gui" value="true"/>

  &lt;node name="joint_state_publisher" pkg="joint_state_publisher" type="joint_state_publisher" />
  &lt;node name="robot_state_publisher" pkg="robot_state_publisher" type="state_publisher" />
  &lt;node name="rviz" pkg="rviz" type="rviz" args="-d $(find mastering_ros_robot_description_pkg)/urdf.rviz"
required="true" />

&lt;/launch>
```

We can launch the model using the following command:

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

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

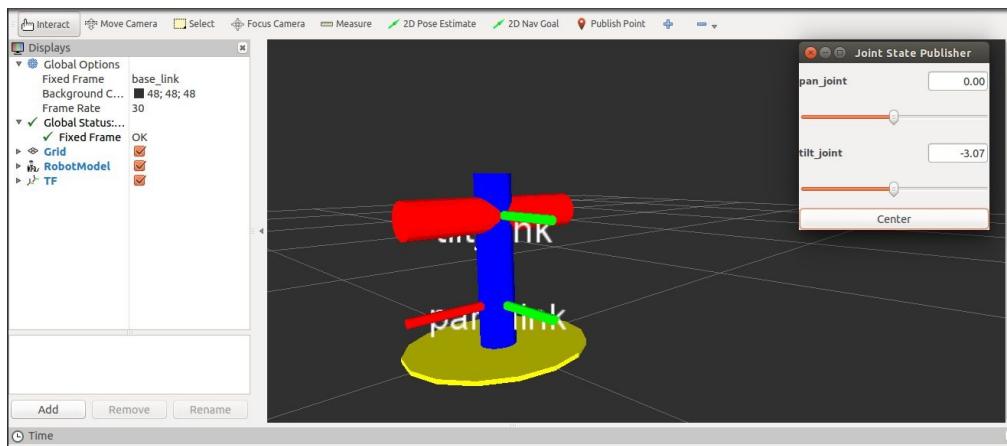


Figure 6 : Joint level of pan and tilt mechanism

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:

```
| &lt;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:

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

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

  &lt;inertial>
    &lt;mass value="1"/>
    &lt;inertia ixx="1.0" ixy="0.0" ixz="0.0" iyy="1.0" iyz="0.0" izz="1.0"/>
  &lt;/inertial>
.....
&lt;/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:

```
| &lt;?xml version="1.0"?>
| &lt;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:

```
&lt;xacro:property name="base_link_length" value="0.01" />
&lt;xacro:property name="base_link_radius" value="0.2" />

&lt;xacro:property name="pan_link_length" value="0.4" />
&lt;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:

```
&lt;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:

```
| &lt;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:

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

```
| &lt;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 URDF 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 URDF 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):

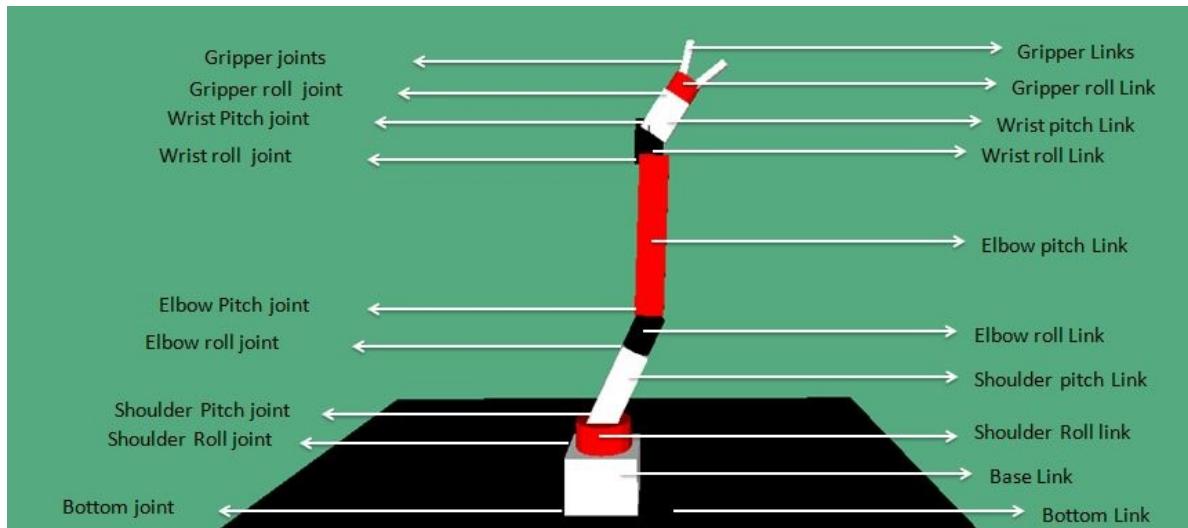


Figure 8 : Joints and Links of seven dof arm robot

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:

```
&lt;?xml version="1.0"?>
&lt;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:

```
&lt;property name="deg_to_rad" value="0.01745329251994329577"/>
&lt;property name="M_PI" value="3.14159"/>

&lt;property name="elbow_pitch_len" value="0.22" />
&lt;property name="elbow_pitch_width" value="0.04" />
&lt;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:

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

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

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

```
| &lt;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 16, *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:

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

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

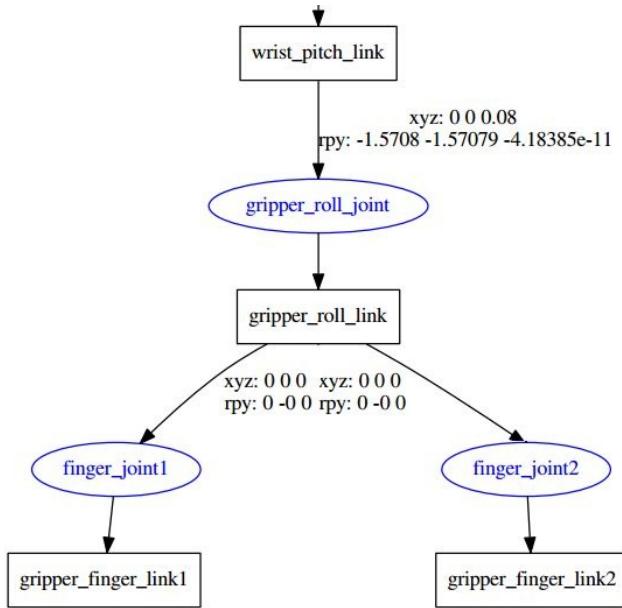


Figure 9 : Graph of the end effector section of seven dof arm robot

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:

```
&lt;launch>
  &lt;arg name="model" />

  &lt;!-- Parsing xacro and loading robot_description parameter -->
  &lt;param name="robot_description" command="$(find xacro)/xacro.py $(find
mastering_ros_robot_description_pkg)/urdf/ seven_dof_arm.xacro" />

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

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

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

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

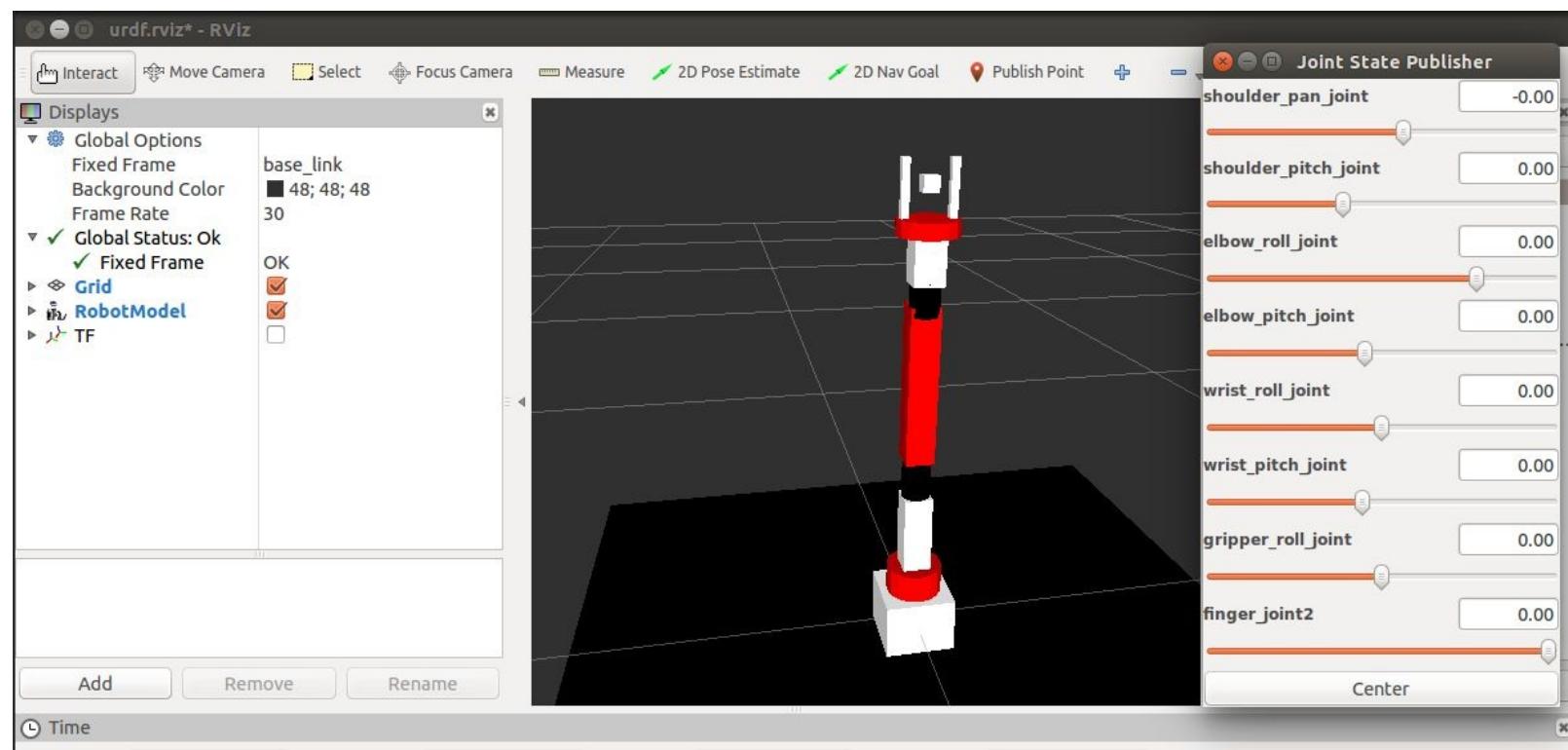


Figure 10 : Seven dof arm in RViz with joint_state_publisher

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:

```
&lt;param name="use_gui" value="true"/>  
&lt;!-- Starting Joint state publisher node which will publish the joint values -->  
&lt;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:

```
&lt;!-- Starting robot state publish which will publish tf -->
&lt;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:

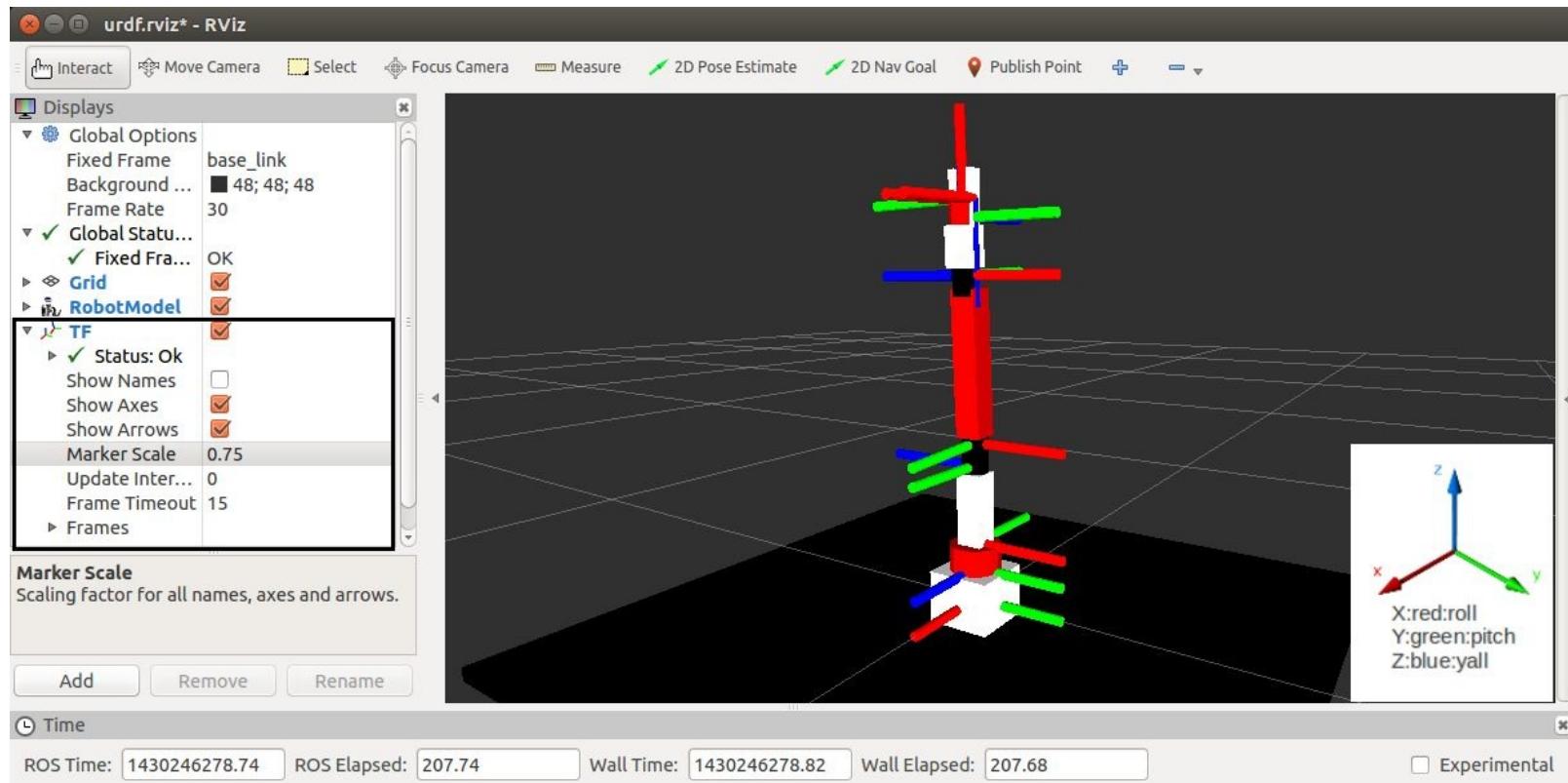


Figure 11 : TF view of seven dof arm in RViz

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:

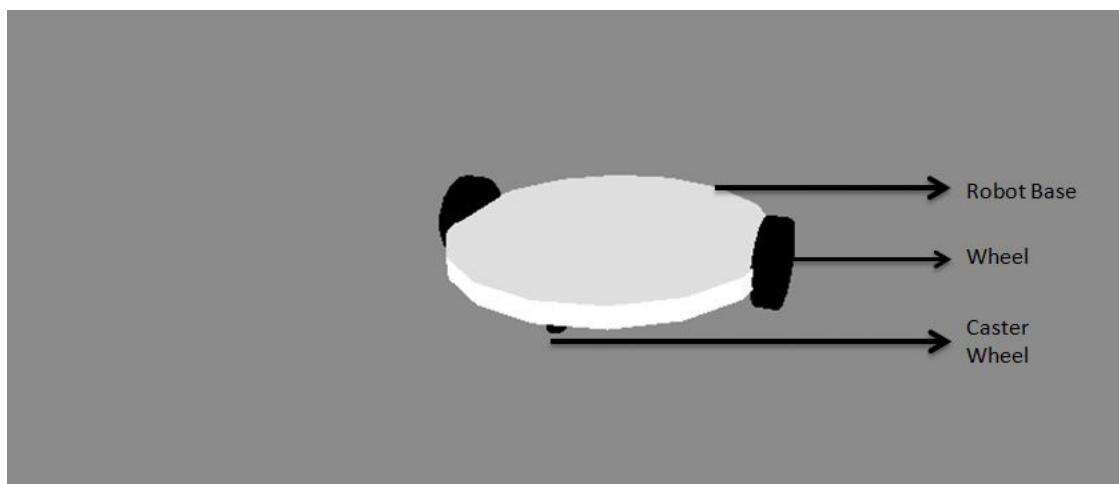


Figure 12 : 3D model of differential drive mobile robot

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:

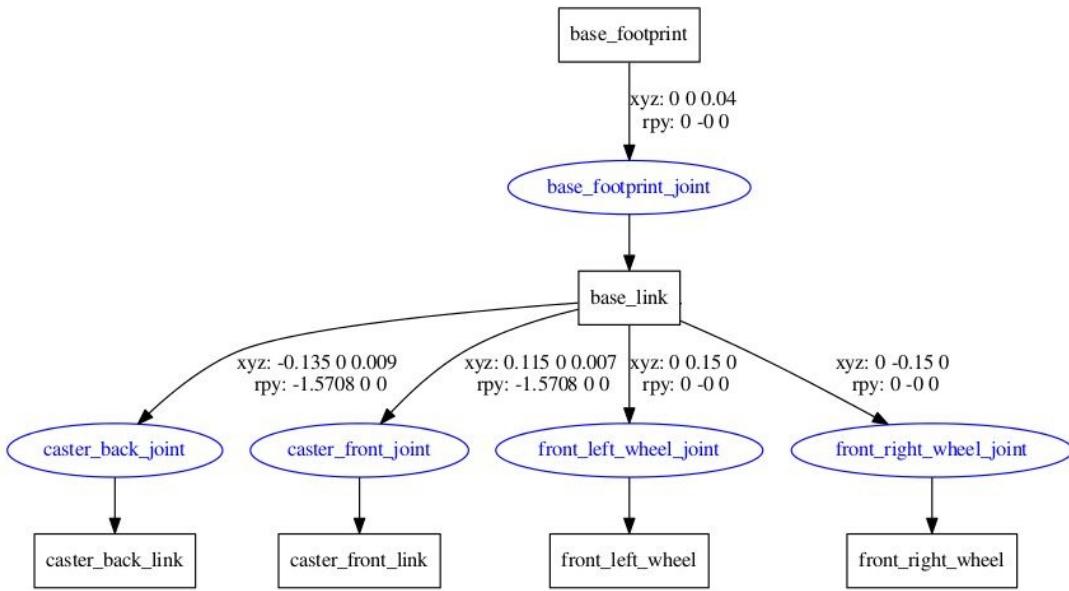


Figure 13 : Graphical representation of the links and joints in mobile robot

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

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

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

```
&lt;xacro:macro name="wheel" params="fb lr parent translateX translateY flipY"> &lt;!--fb : front, back ; lr: left, right -->
    &lt;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 `gazeboreference` tag:

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

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

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

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

&lt;/xacro:macro>
&lt;/robot>
```

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

```
&lt;wheel fb="front" lr="right" parent="base_link" translateX="0" translateY="-0.5" flipY="-1"/>
&lt;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:

```
&lt;!-- Macro for calculating inertia of cylinder -->
&lt;macro name="cylinder_inertia" params="m r h">
  &lt;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}" />
&lt;/macro>
```

The launch file definition for displaying this root model in RViz is given here.

The launch file is named `view_mobile_robot.launch`:

```
&lt;launch>
  &lt;arg name="model1" />
  &lt;!-- Parsing xacro and setting robot_description parameter -->
  &lt;param name="robot_description" command="$(find xacro)/xacro.py $(find
mastering_ros_robot_description_pkg)/urdf/diff_wheeled_robot.xacro" />
  &lt;!-- Setting gui parameter to true for display joint slider -->
  &lt;param name="use_gui" value="true"/>
  &lt;!-- Starting Joint state publisher node which will publish the joint values -->
  &lt;node name="joint_state_publisher" pkg="joint_state_publisher" type="joint_state_publisher" />
  &lt;!-- Starting robot state publish which will publish tf -->
  &lt;node name="robot_state_publisher" pkg="robot_state_publisher" type="state_publisher" />
  &lt;!-- Launch visualization in rviz -->
  &lt;node name="rviz" pkg="rviz" type="rviz" args="-d $(find mastering_ros_robot_description_pkg)/urdf.rviz"
required="true" />
&lt;/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:

```
$ rosrun mastering_ros_robot_description_pkg
view_mobile_robot.launch
```

The screenshot of the robot in RViz is as follows:

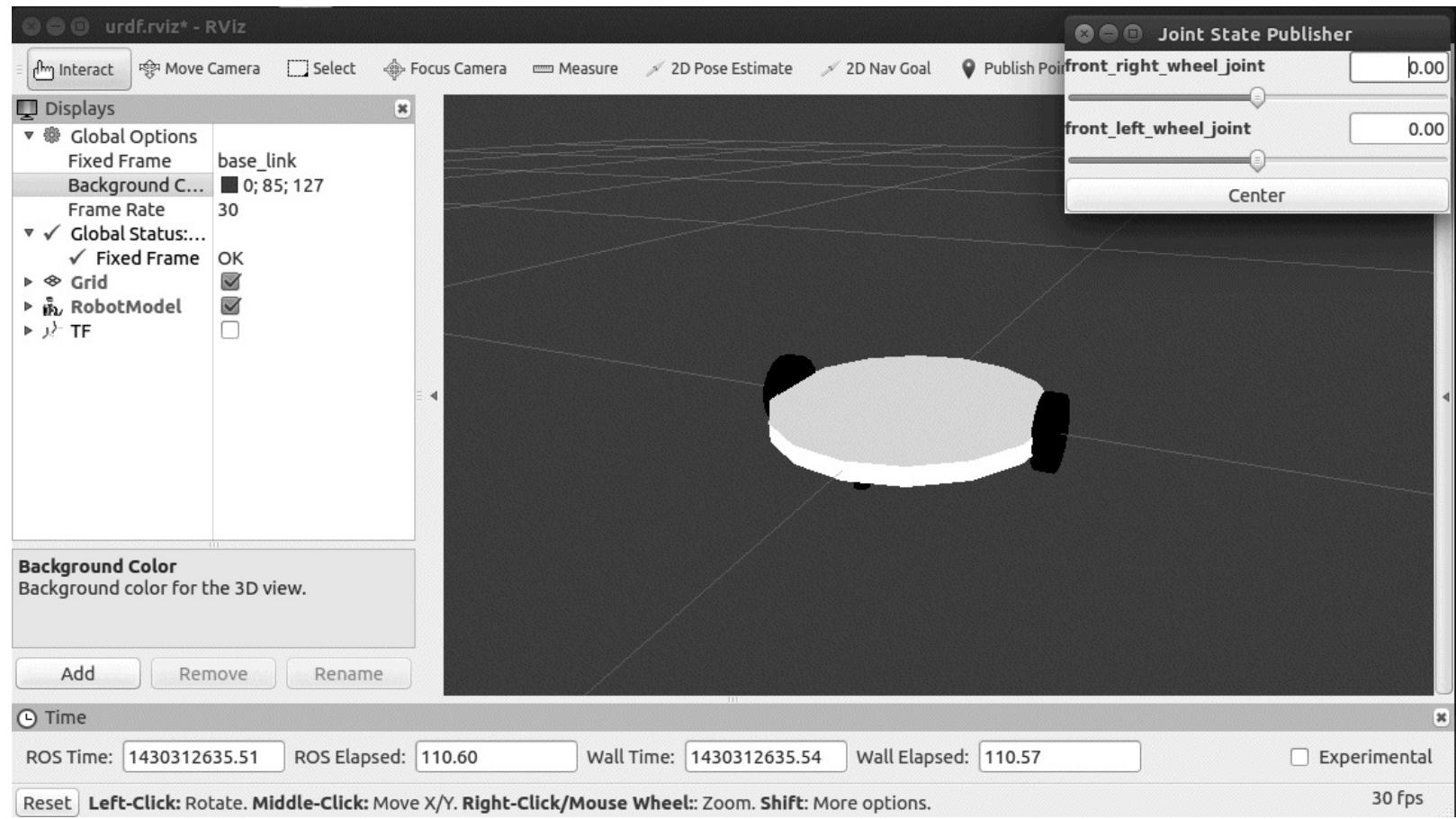


Figure 14 : Visualizing mobile robot in RViz with joint state publisher.

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.

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:

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

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

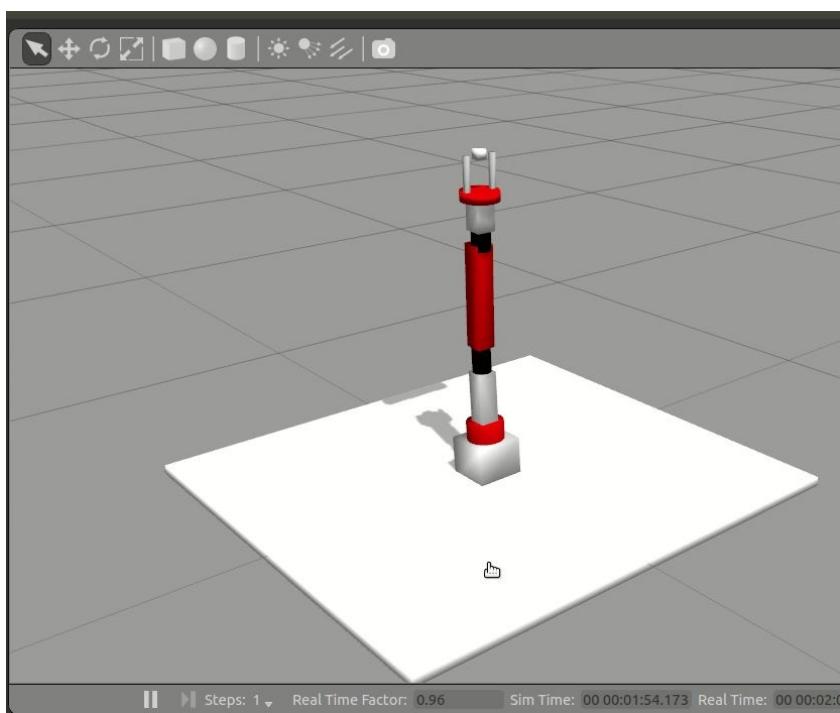


Figure 1 : Simulation of Seven DOF arm in Gazebo

Let's discuss the `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:

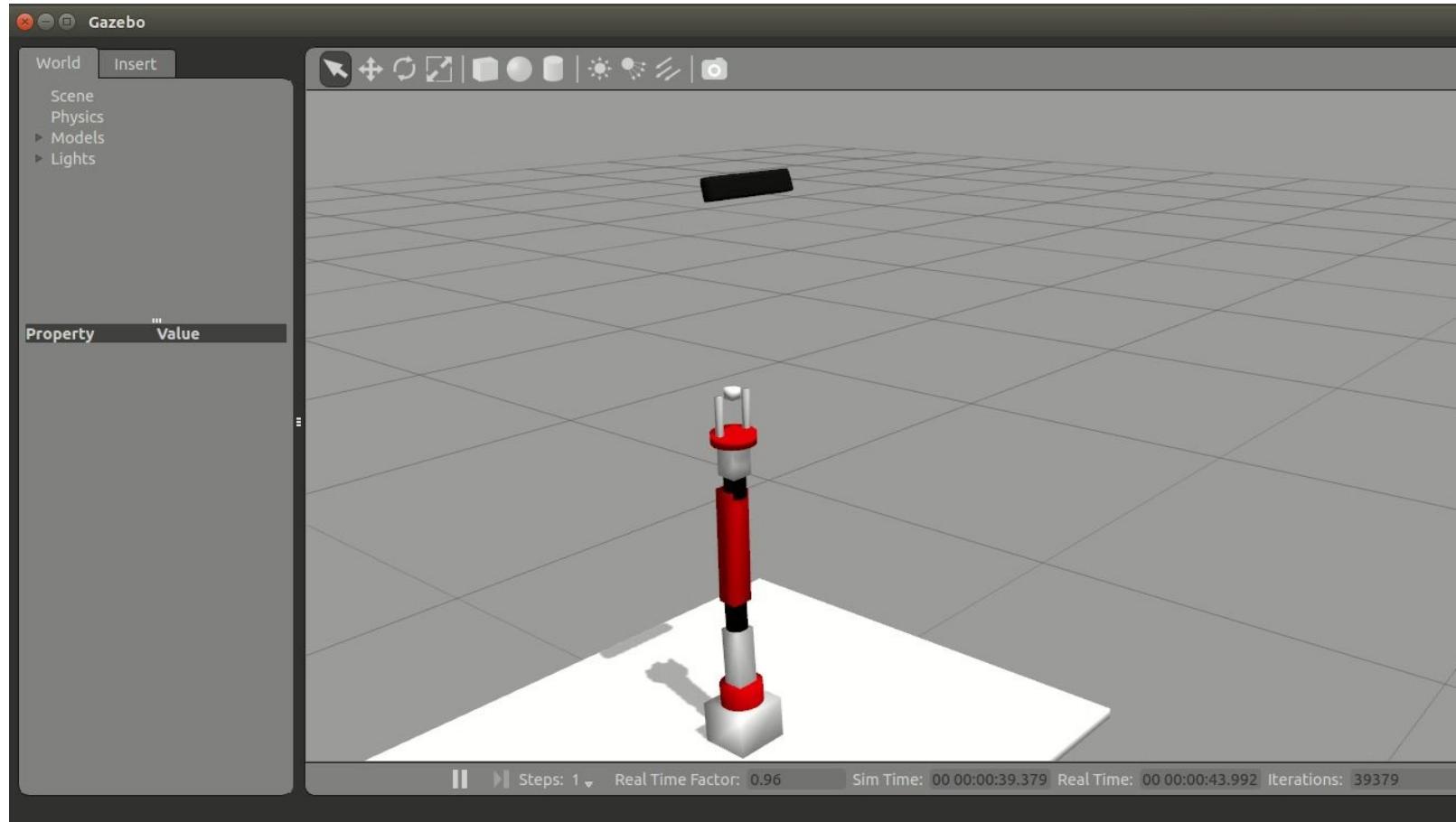


Figure 2 : Simulation of seven DOF arm with Asus Xtion Pro in Gazebo

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:

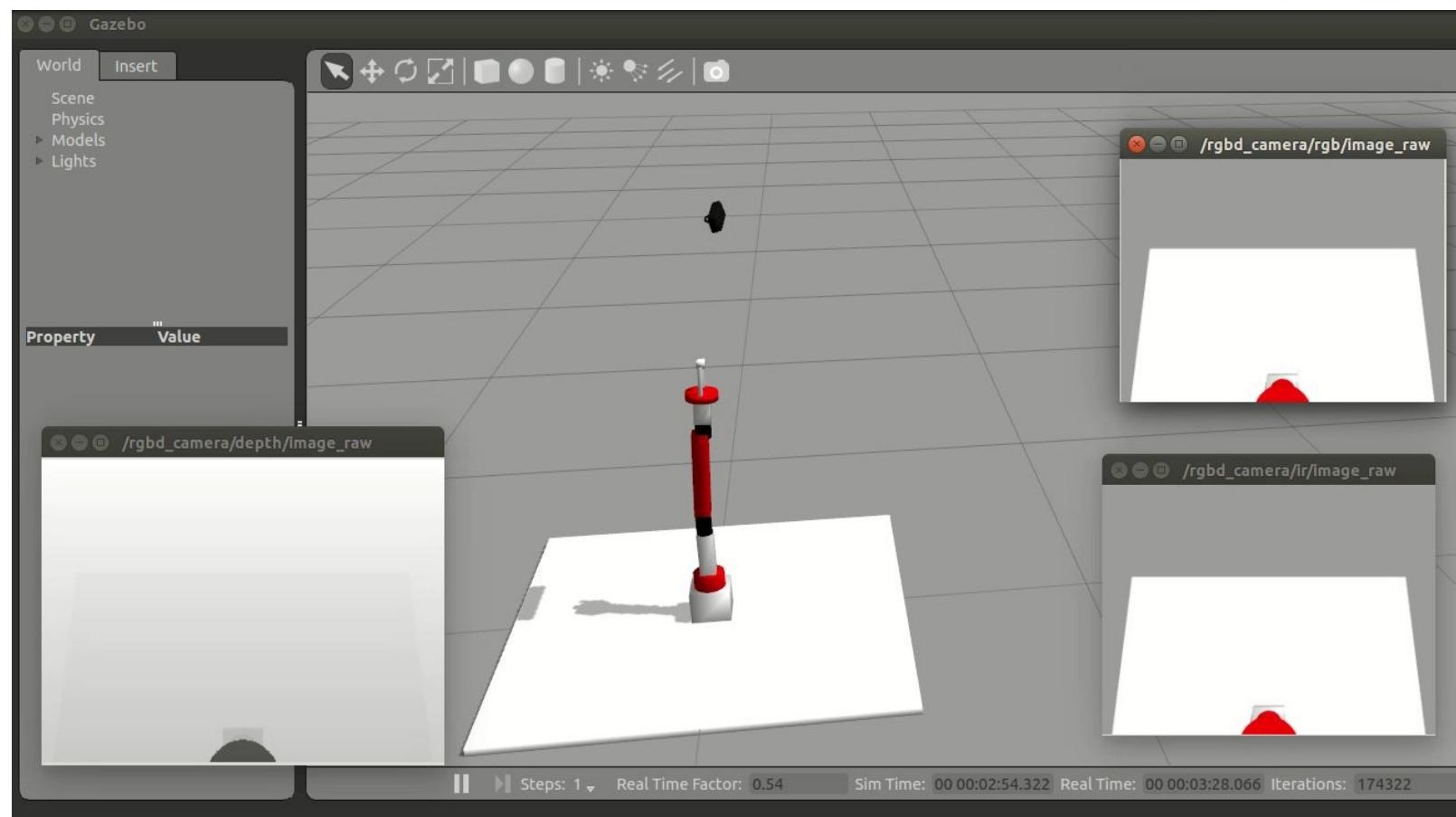


Figure 4 : Viewing images of Xtion Pro in Gazebo

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:

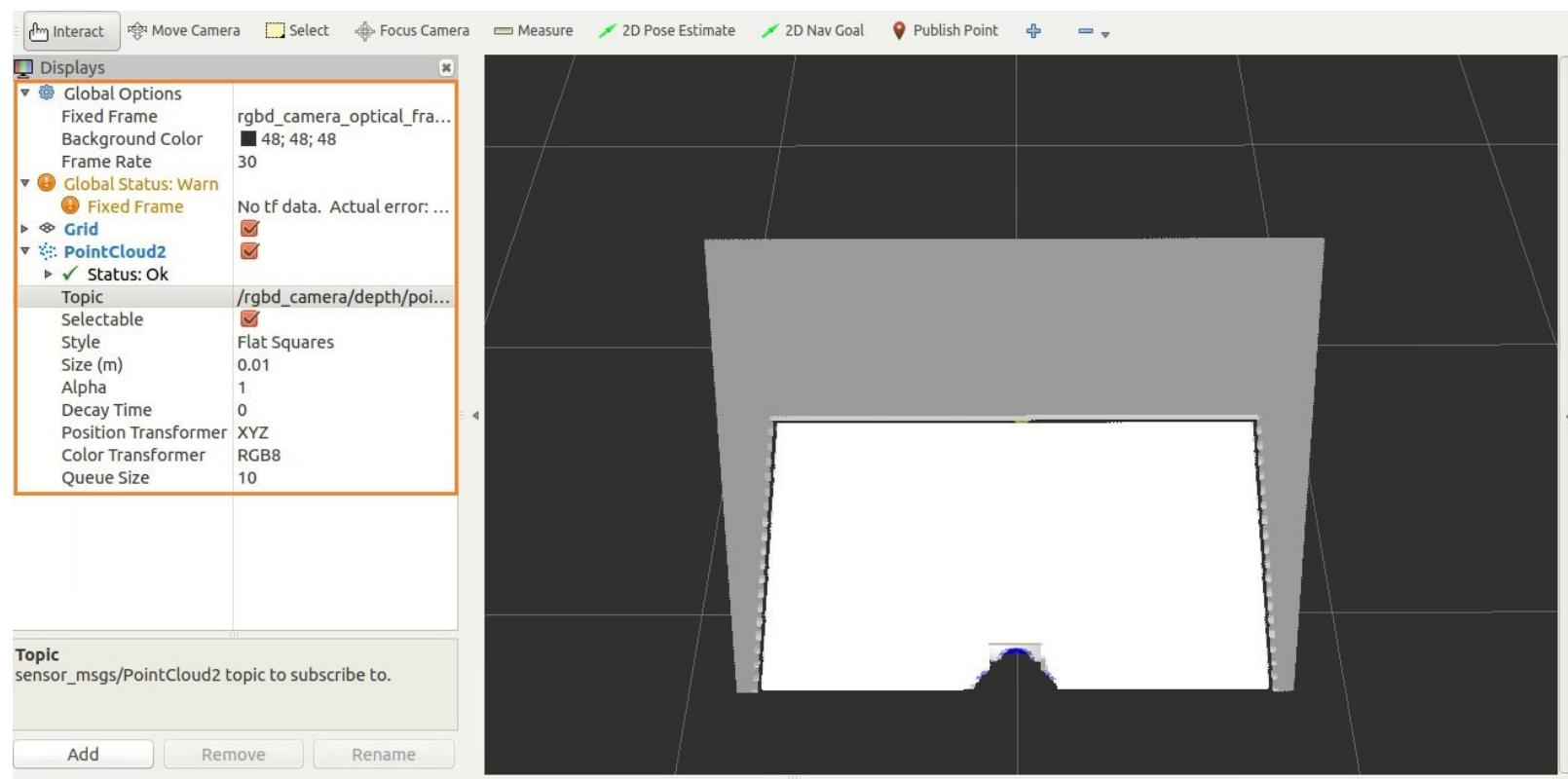


Figure 5 : Viewing point cloud data from Xtion Pro in RViz

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:

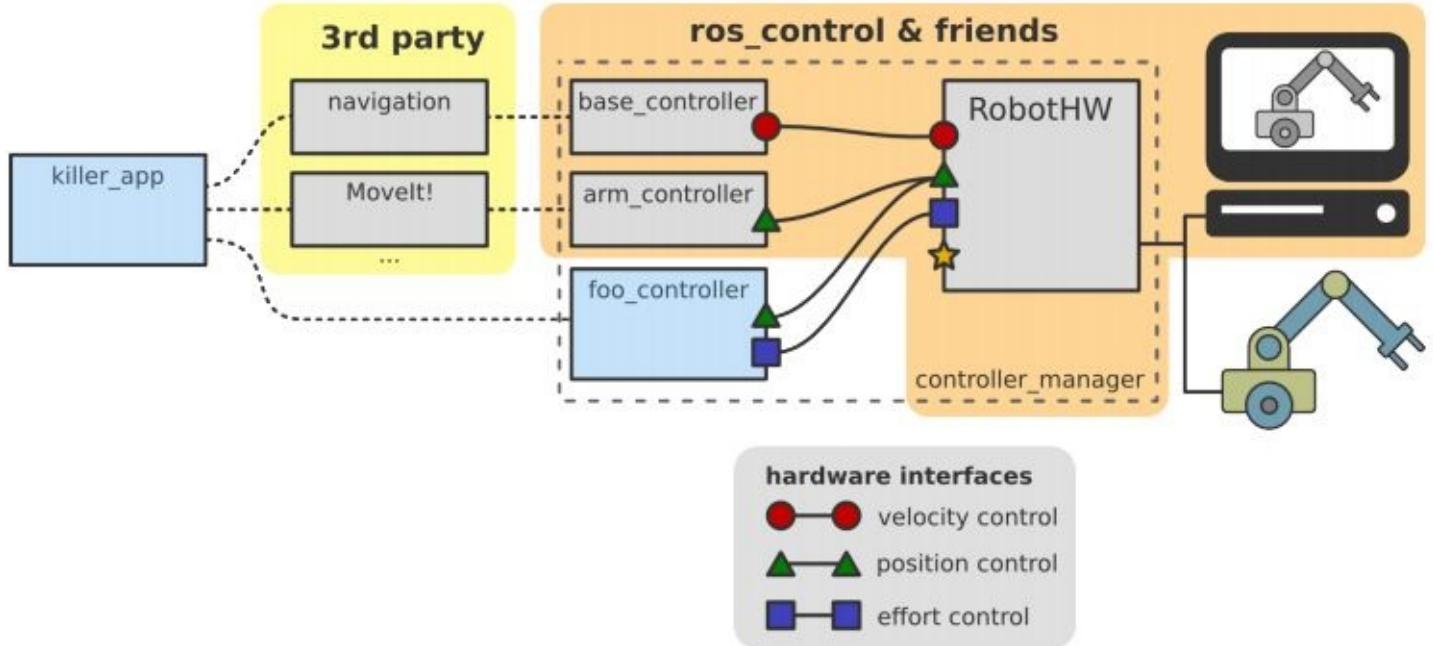


Figure 6 : Interacting ROS controllers with Gazebo

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:

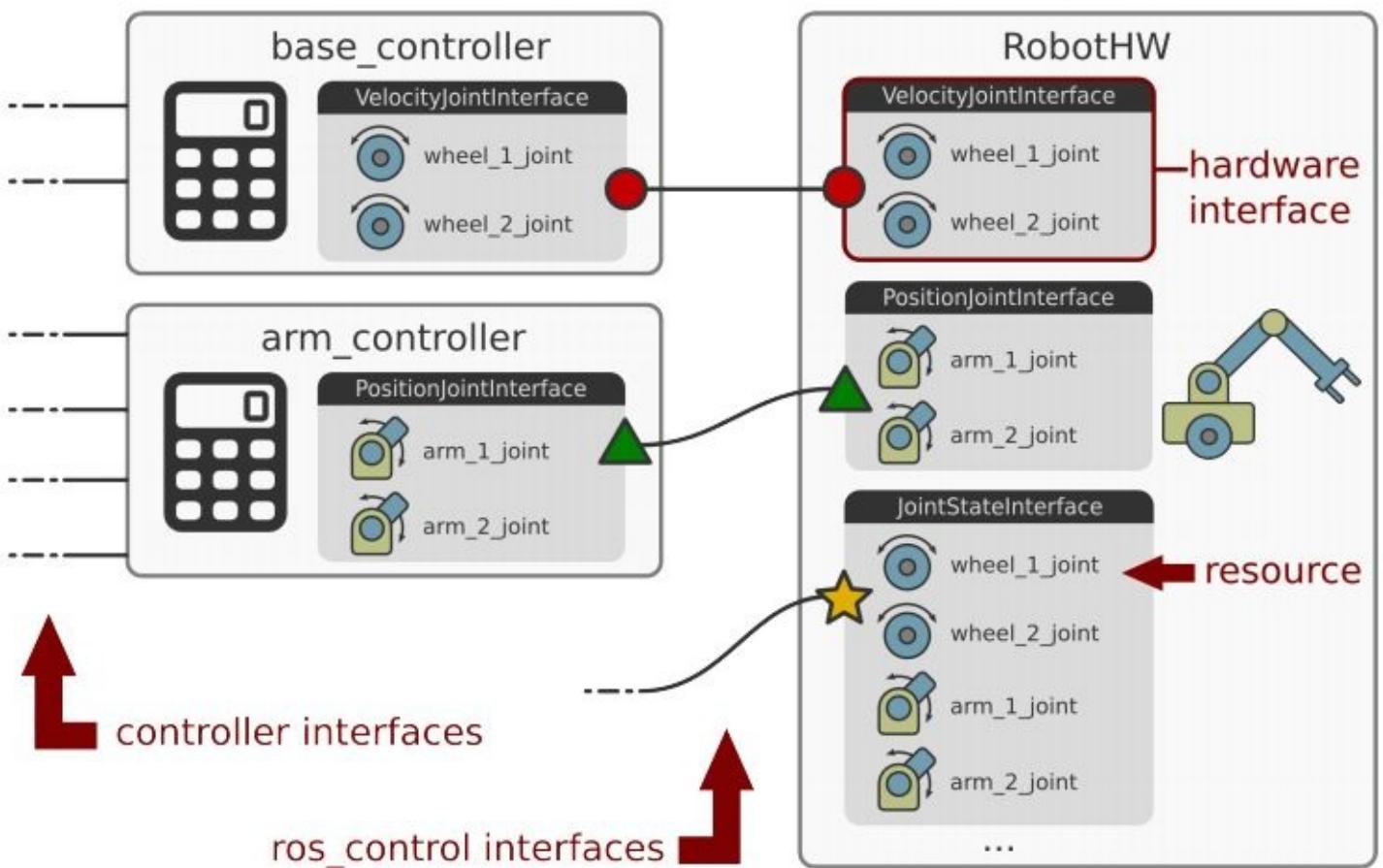


Figure 7 : Illustration of ROS controllers and hardware interfaces

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_arm_gazebo seven_dof_arm_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
```

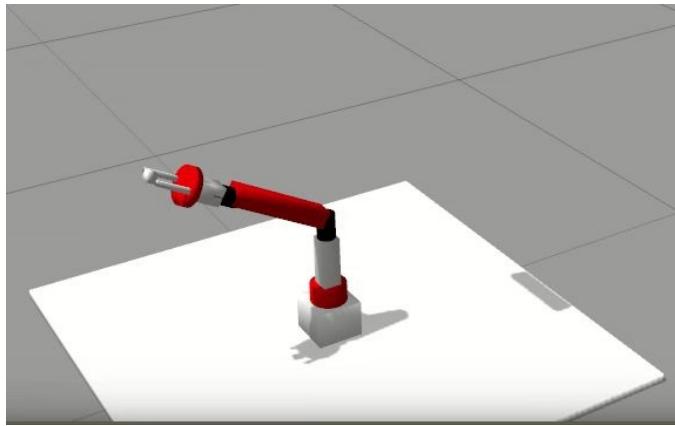


Figure 10 : Moving a joint of the arm in Gazebo

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:

```
| $ rosrun 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:

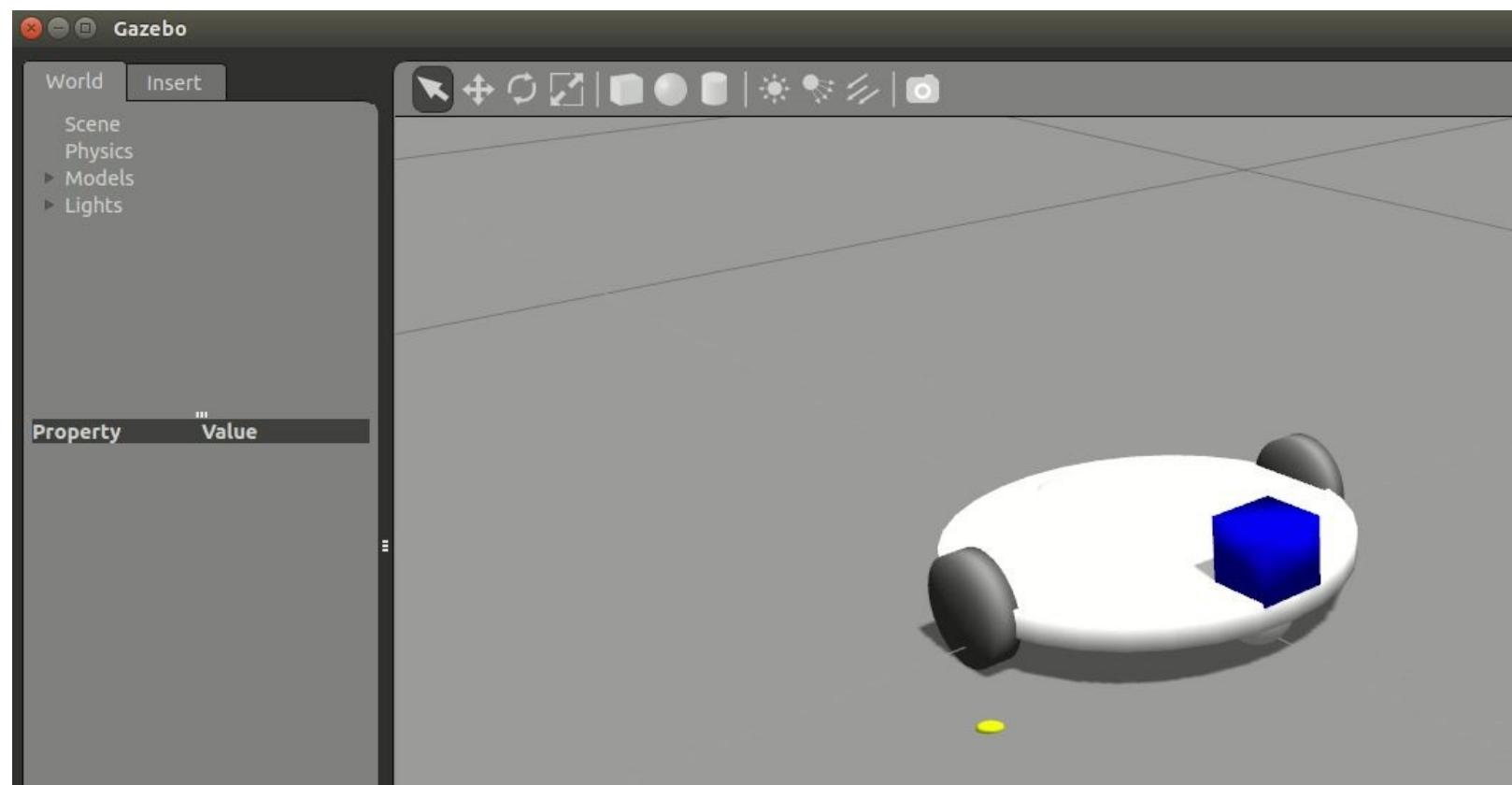


Figure 11 : Differential wheeled robot in Gazebo

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:

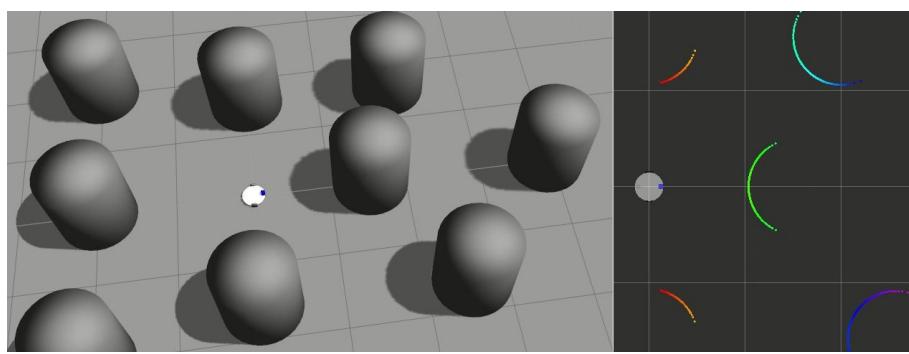


Figure 12 : Differential drive robot in random object in Gazebo

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.

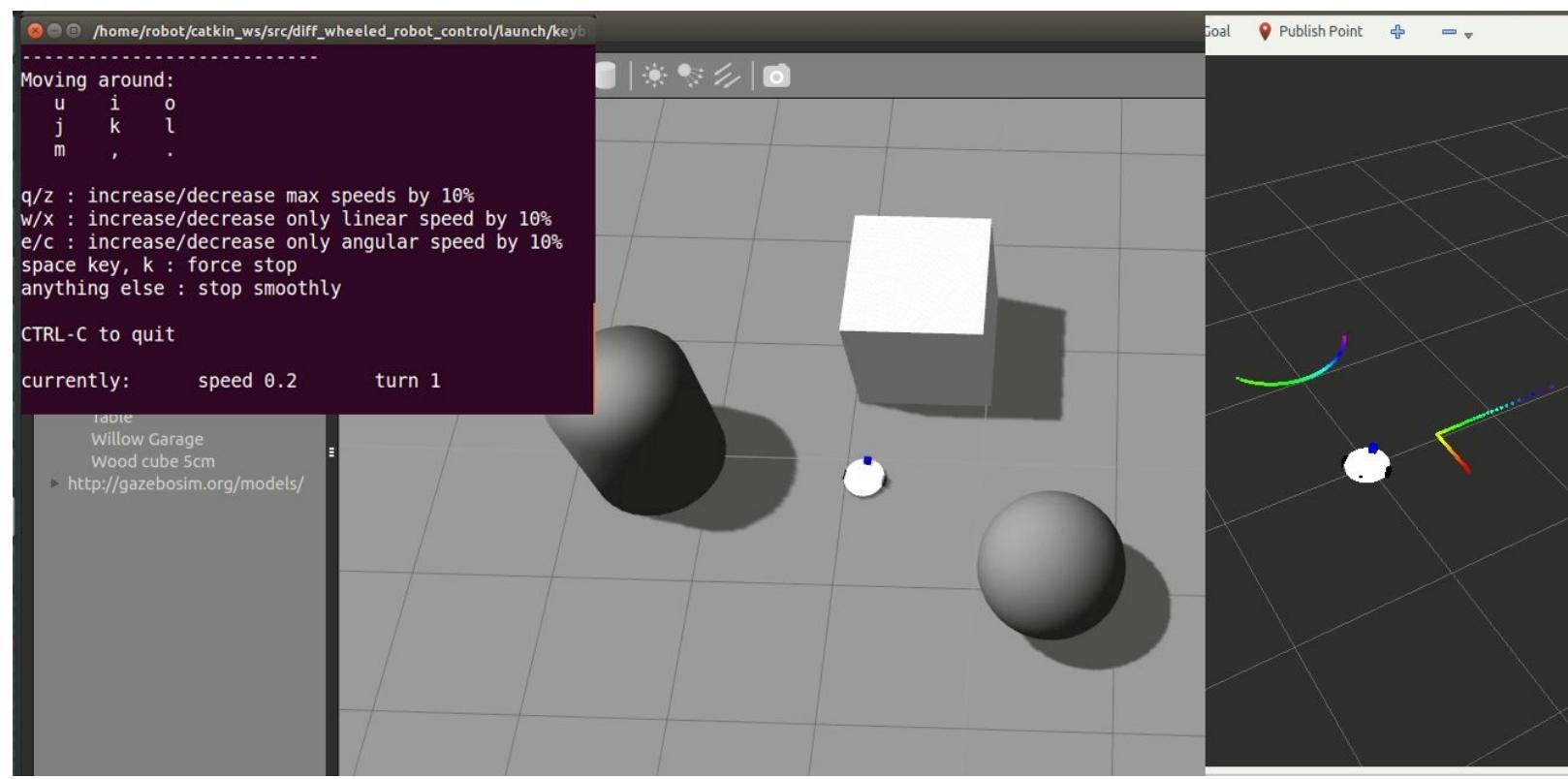


Figure 13 : Moving differential drive robot in Gazebo using teleoperation

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>.

Working with Pluginlib, Nodelets, and Gazebo Plugins

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:

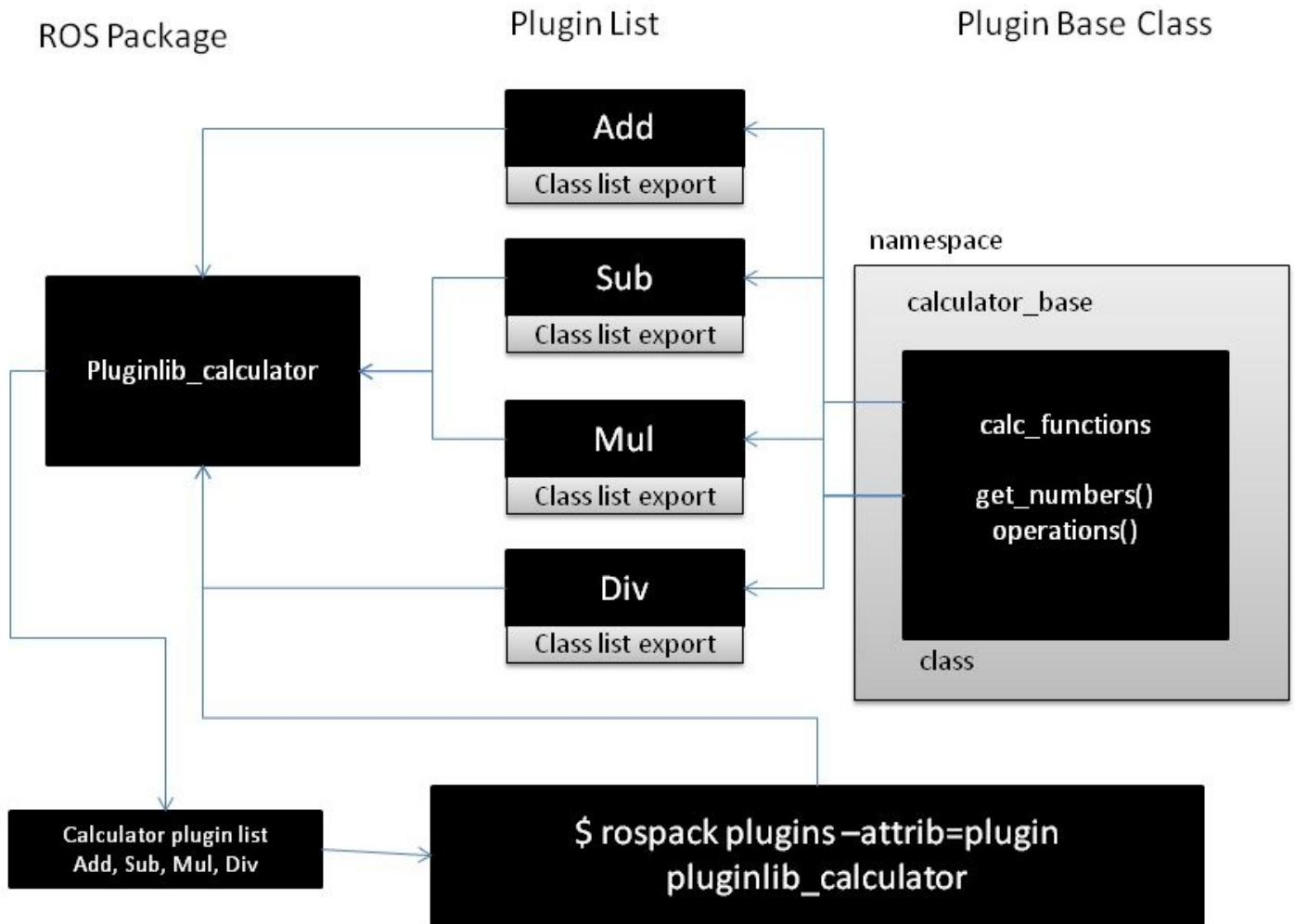


Figure 1: Organization of plugins in the calculator application

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:

```
&lt;library path="lib/libpluginlib_calculator">
  &lt;class name="pluginlib_calculator/Add" type="calculator_plugins::Add"
    base_class_type="calculator_base::calc_functions">
    &lt;description>This is a add plugin.&lt;/description>
  &lt;/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:

```
&lt;export>
  &lt;pluginlib_calculator plugin="${prefix}/calculator_plugins.xml"
  />
&lt;/export>
```

In order to work this export command properly, we should insert the following lines in `package.xml`:

```
&lt;build_depend>pluginlib_calculator&lt;/build_depend>
&lt;run_depend>pluginlib_calculator&lt;/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.

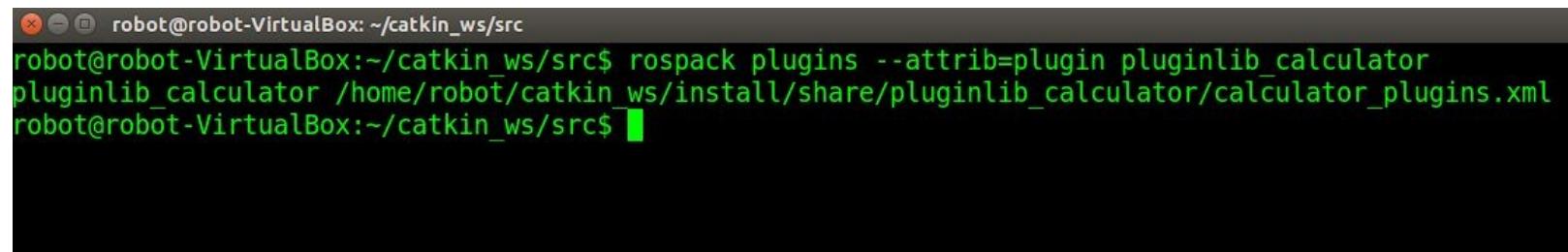
We are almost done with all the settings and now it's time to build the package using the `catkin_make` command.

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 screenshot of a terminal window titled "robot@robot-VirtualBox: ~/catkin_ws/src". The window contains the command "rospack plugins --attrib=plugin pluginlib_calculator" followed by its output: "pluginlib_calculator /home/robot/catkin_ws/install/share/pluginlib_calculator/calculator_plugins.xml". The terminal has a dark background with light-colored text.

Figure 2: The result of the plugin query

Step 9 - Running the plugin loader

We can run the `calculator_loader` using the following command:

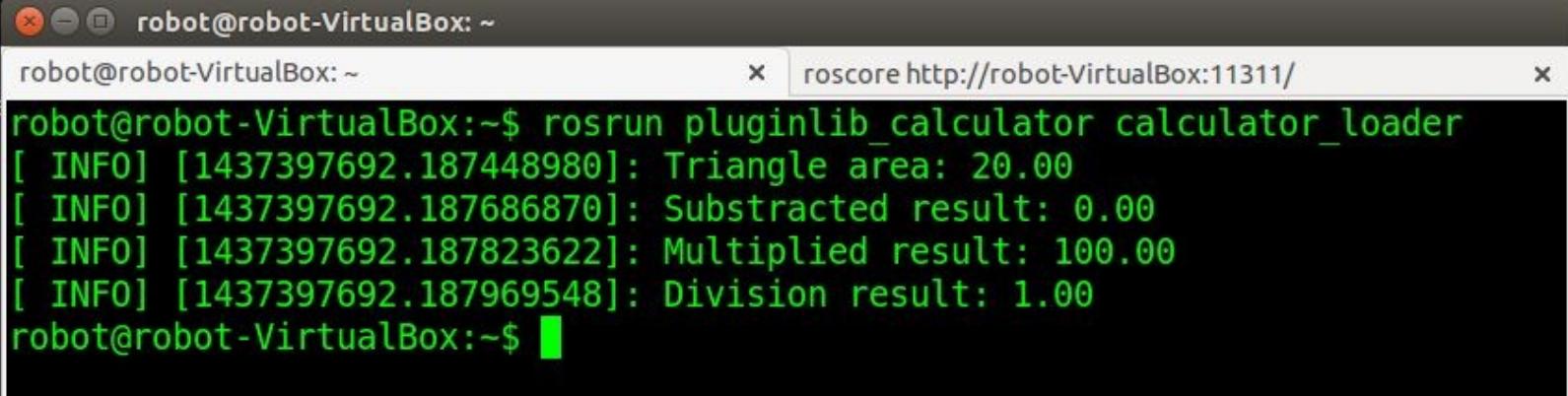
- Run the `roscore`, as follows:

```
| $ 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:



A screenshot of a terminal window titled "robot@robot-VirtualBox: ~". The window contains two tabs: "robot@robot-VirtualBox: ~" and "roscore http://robot-VirtualBox:11311/". The main terminal window displays the output of the command `rosrun pluginlib_calculator calculator_loader`. The output shows four INFO messages indicating the results of different calculations: "Triangle area: 20.00", "Subtracted result: 0.00", "Multiplied result: 100.00", and "Division result: 1.00".

```
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_macro.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:

```
&lt;library path="libnodelet_hello_world">
  &lt;class name="nodelet_hello_world/Hello" type="nodelet_hello_world::Hello" base_class_type="nodelet::Nodelet">
    &lt;description>
      A node to republish a message
    &lt;/description>
  &lt;/class>
&lt;/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:

```
&lt;export>
  &lt;nodelet plugin="${prefix}/hello_world.xml"/>
&lt;/export>

&lt;build_depend>nodelet_hello_world&lt;/build_depend>
&lt;run_depend>nodelet_hello_world&lt;/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:

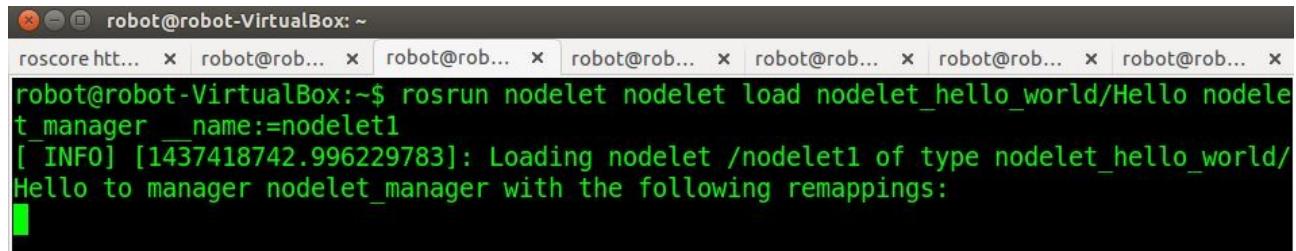
```
robot@robot-VirtualBox:~$ rosrun nodelet nodelet manager __name:=nodelet_manager
[ INFO] [1437416267.697084914]: Initializing nodelet with 2 worker threads.
```

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"
```

```
robot@robot-VirtualBox:~$ rostopic pub /nodelet1/msg_in std_msgs/String "Hello"
publishing and latching message. Press ctrl-C to terminate
[]
```

```
robot@robot-VirtualBox:~$ rostopic echo /nodelet1/msg_out
data: Hello
--
```

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`:

```
&lt;launch>
  &lt;!-- Started nodelet manager -->
  &lt;node pkg="nodelet" type="nodelet" name="standalone_nodelet" args="manager" output="screen"/>
  &lt;!-- Starting first nodelet -->
  &lt;node pkg="nodelet" type="nodelet" name="test1" args="load nodelet_hello_world/Hello standalone_nodelet" output="screen">
    &lt;/node>
  &lt;!-- Starting second nodelet -->
  &lt;node pkg="nodelet" type="nodelet" name="test2" args="load nodelet_hello_world/Hello standalone_nodelet" output="screen">
    &lt;/node>
&lt;/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:

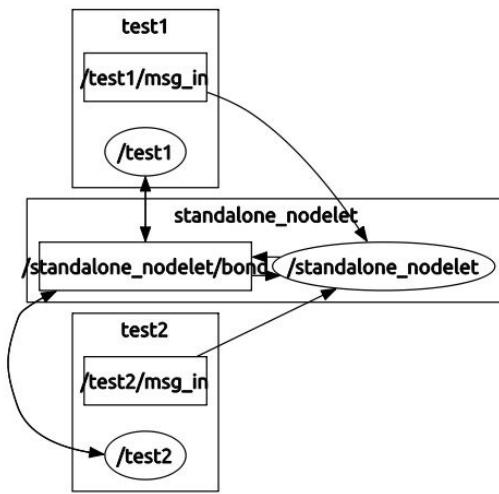


Figure 10: A two-node instance of a nodelet

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 | NodeGraph 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_tutorials$ 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:

Source



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.

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 11, 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.

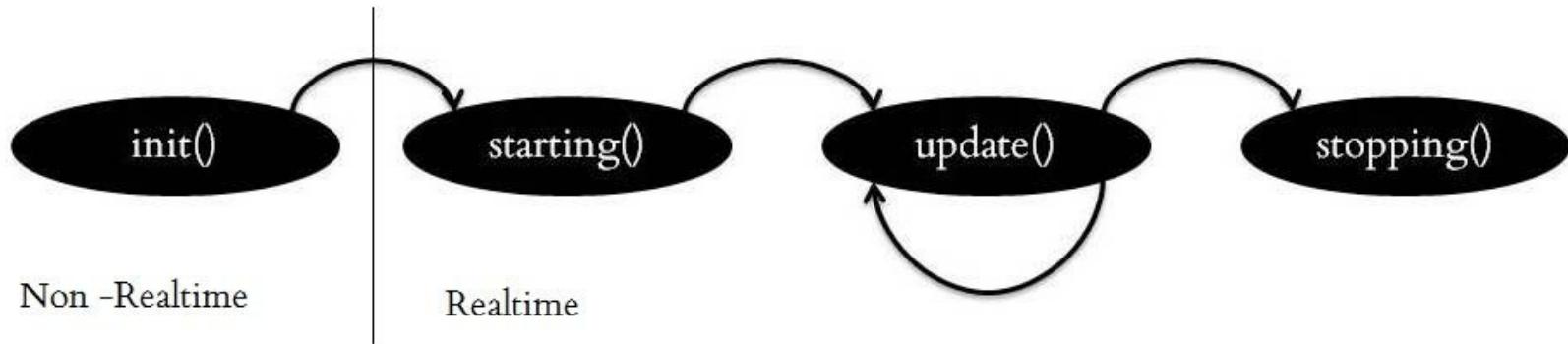


Figure 1: Workflow of the controller

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:

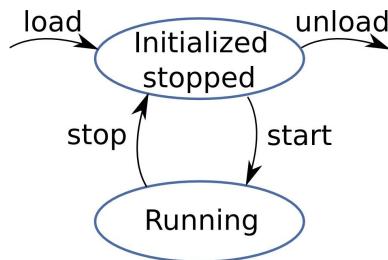


Figure 2: Working of controller manager

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`:

```
&lt;library path="lib/libmy_controller_lib">
  &lt;class name="my_controller_pkg/MyControllerPlugin"
    type="my_controller_ns::MyControllerClass"
    base_class_type="pr2_controller_interface::Controller" />
&lt;/library>
```

Step 6 – Updating package.xml

We need to update the package.xml for pointing the controller_plugins.xml file:

```
&lt;export>
  &lt;pr2_controller_interface plugin="${prefix}/controller_plugins.xml" />
&lt;/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_shoulder_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:

```
&lt;launch>
  &lt;rosparam file="$(find my_controller_pkg)/my_controller.yaml" command="load" />

  &lt;!--We can use spawner tool to start running the custom controller -->
  &lt;node pkg="pr2_controller_manager" type="spawner" args="my_controller_name" name="my_controller_spawner" />
&lt;/launch>
```

Step 11 – Running controller along with PR2 simulation in Gazebo

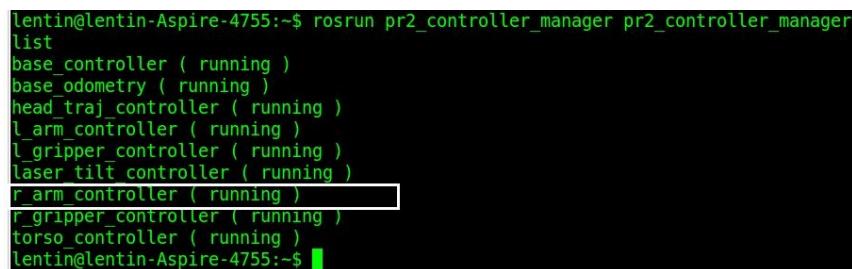
After creating the controller launch files, we have to test it on PR2. We can launch the PR2 robot simulation using following command:

```
| $roslaunch 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 ) r_arm_controller ( running )
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:

```
| $ roslaunch 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:

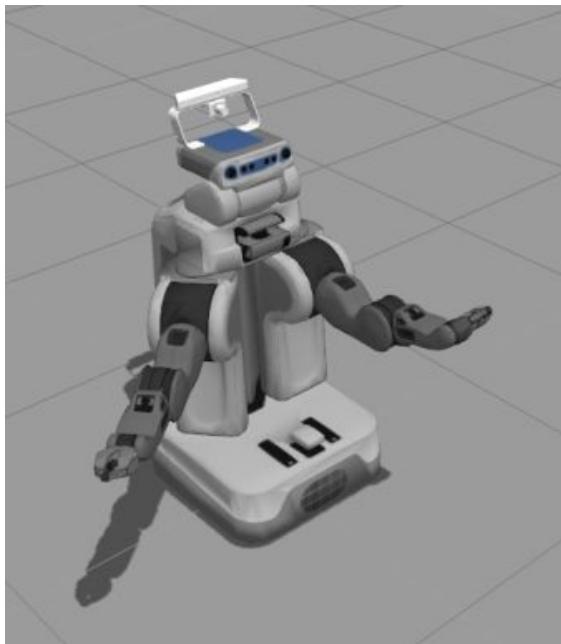


Figure 5: PR2 right hand joint working using our controller

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 rewritten and formed a new set of packages called `ros_control` (http://wiki.ros.org/ros_control).

The `ros_control` implement 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 11, 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:

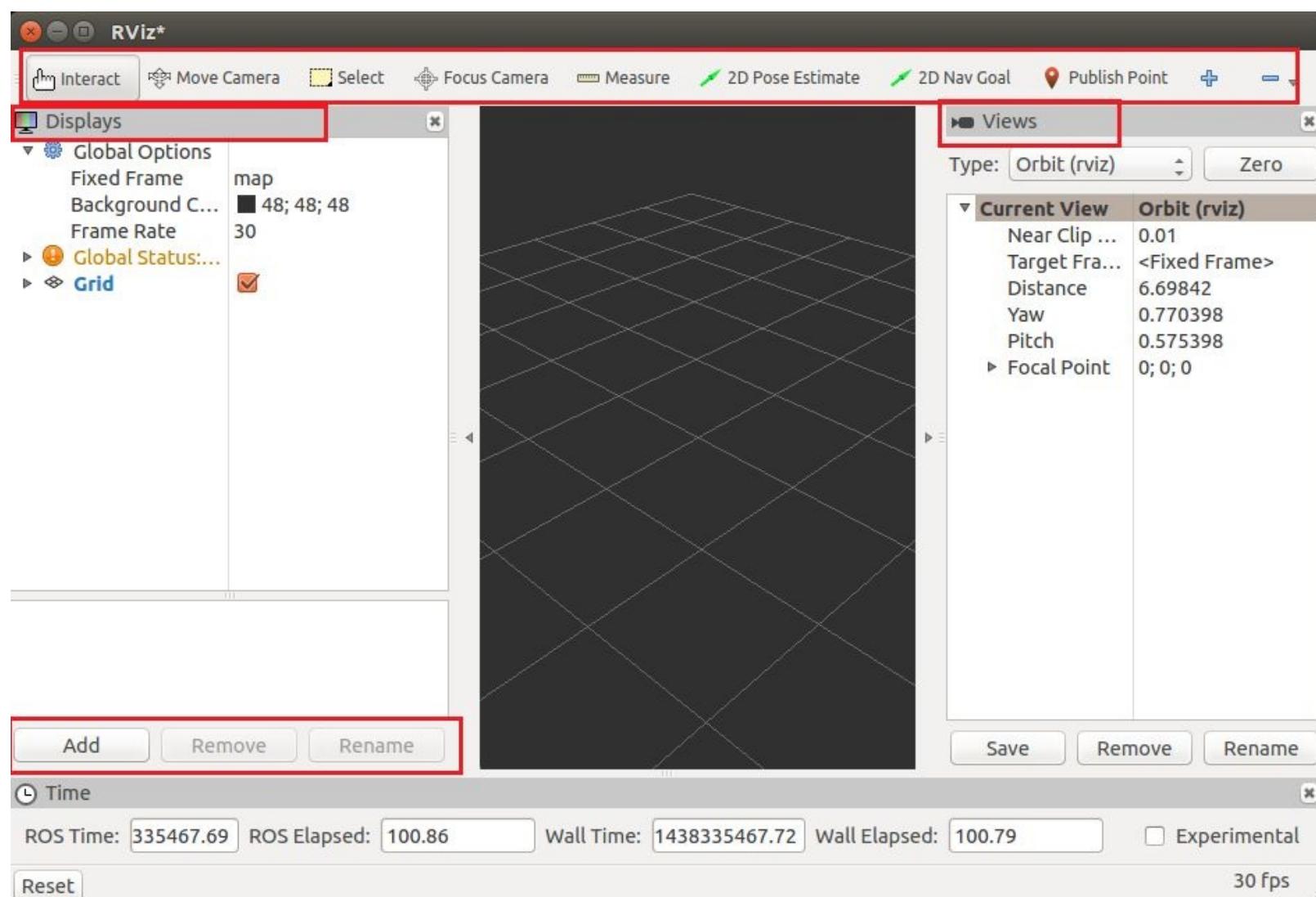


Figure 6: RViz and its toolbars

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_plugin_tutorial.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_plugin_tutorial.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:

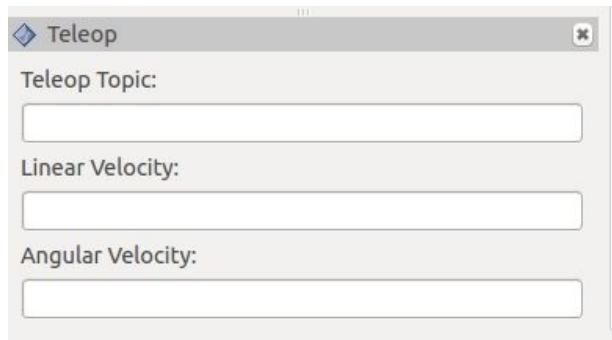


Figure 7: RViz teleop plugin

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-invoke 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 times out. 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:

```
&lt;library path="lib/librviz_telop_commander">
  &lt;class name="rviz_telop_commander/Teleop"
    type="rviz_telop_commander::TeleopPanel"
    base_class_type="rviz::Panel">
    &lt;description>
      A panel widget allowing simple diff-drive style robot base control.
    &lt;/description>
  &lt;/class>
&lt;/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:

```
&lt;export>
  &lt;rviz plugin="${prefix}/plugin_description.xml"/>
&lt;/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:

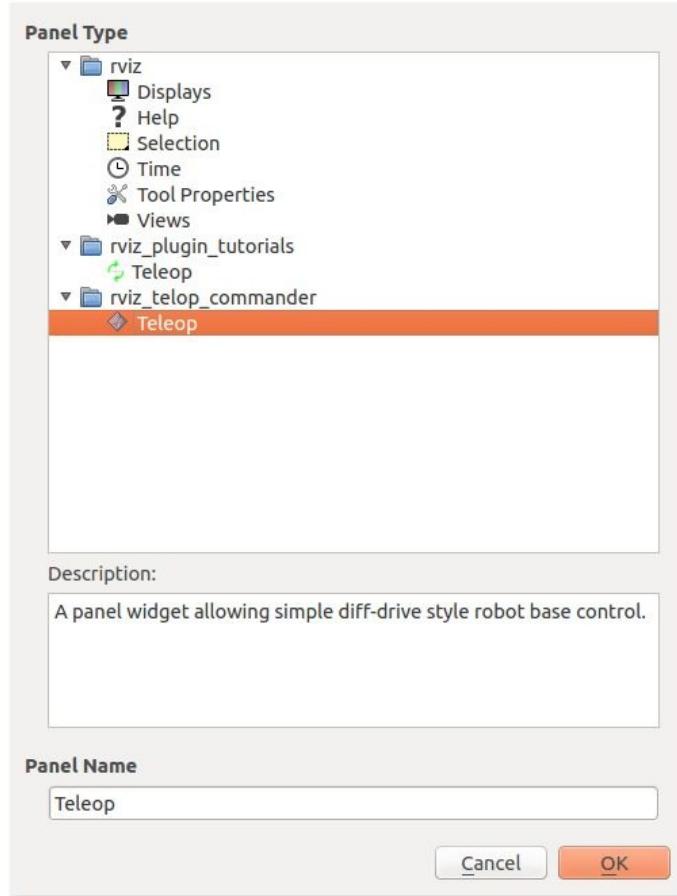


Figure 8: Loading teleop node from RViz

If we load the Teleop plugin from the list, we will get a panel like the following:

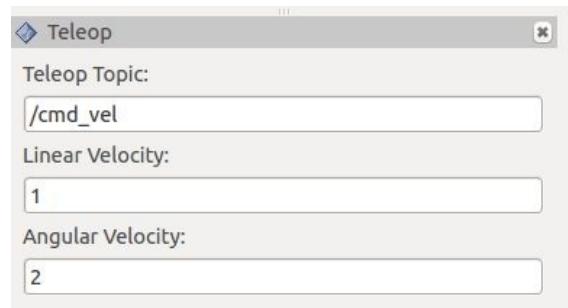


Figure 9: Loading teleop node from RViz

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.

Interfacing I/O Boards, Sensors, and Actuators to ROS

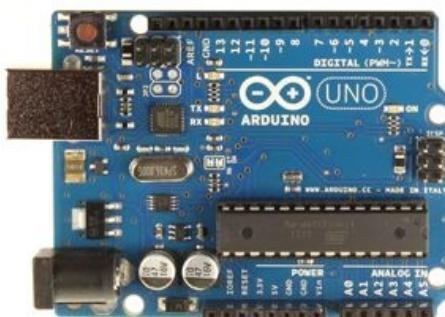
In the last two chapters, we discussed different kinds of plugin frameworks that are used in ROS. In this chapter, we are going to discuss interfacing of some hardware components, such as sensors and actuators to ROS. We will see interfacing of sensors using I/O boards such as Arduino, Raspberry Pi, 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.



Beginner : Arduino UNO



Intermediate : Arduino Mega



Advanced : Arduino Due

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:

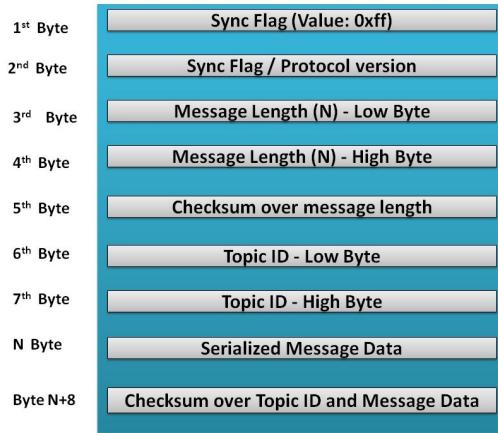


Figure 2 : rosserial packet representation

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 `roslib`, 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:

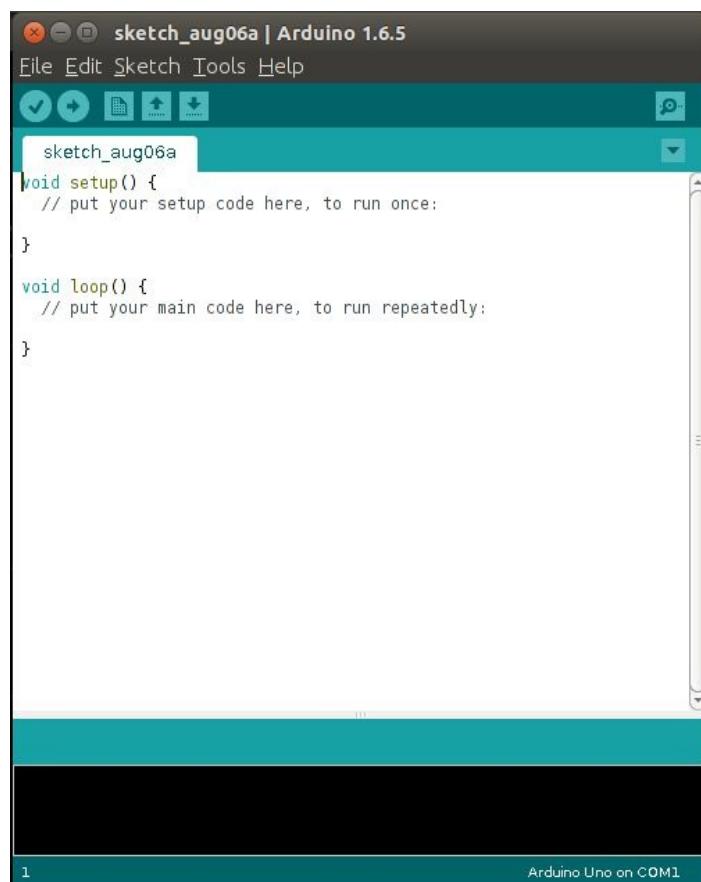


Figure 3 : Arduino IDE

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.

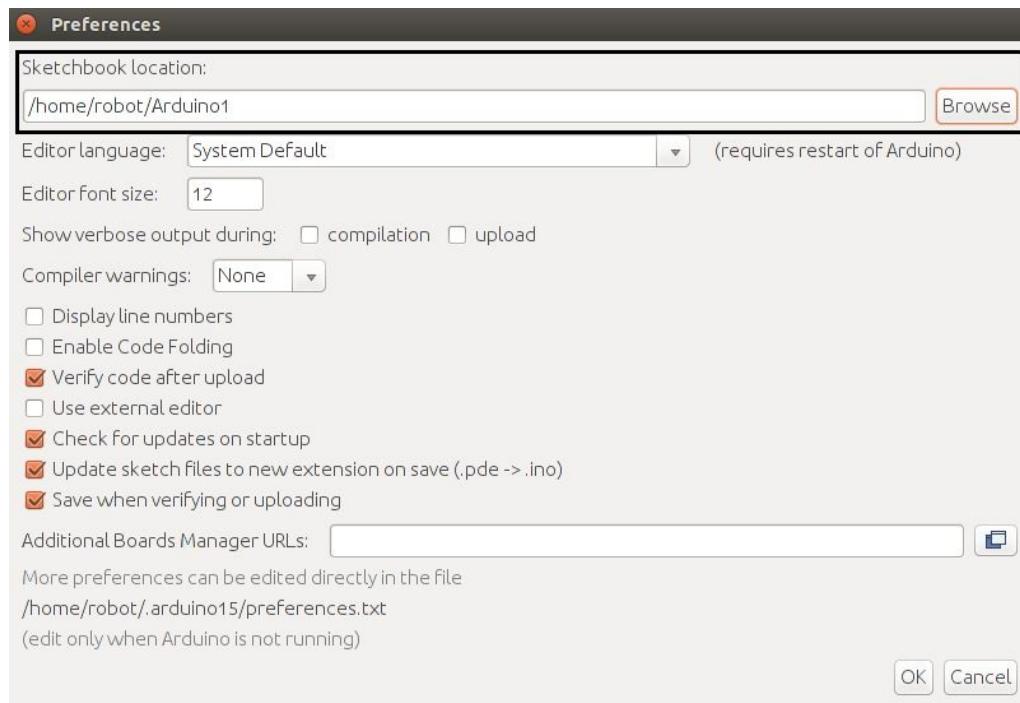


Figure 4 : Preference of Arduino IDE

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:

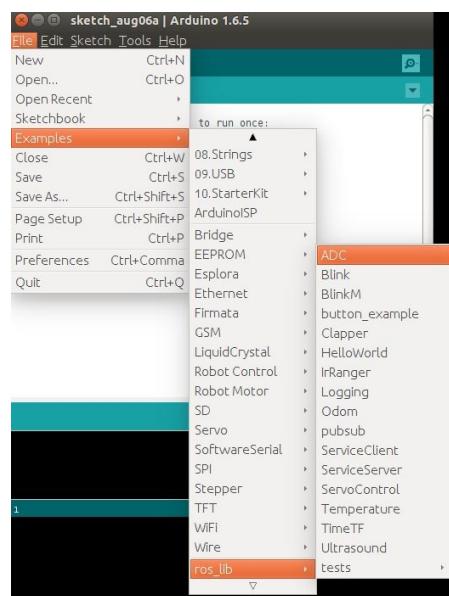


Figure 5 : List of Arduino - ROS examples

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:

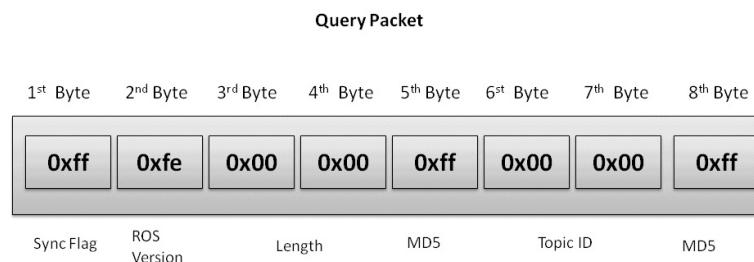


Figure 7 : Structure of Query Packet

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:

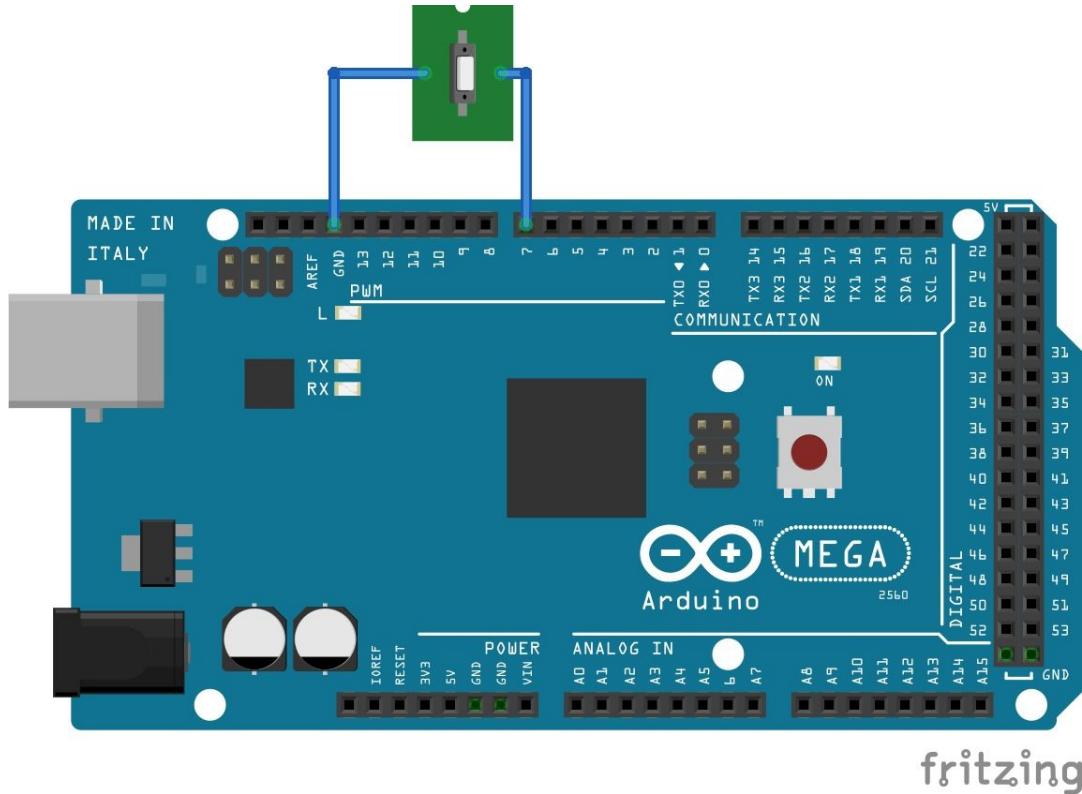


Figure 10 : Interfacing the push button to Arduino

```
/*
 * 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:

```
---  
data: False  
---  
data: True  
---  
data: False  
---  
data: False  
---  
data: True  
---
```

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::

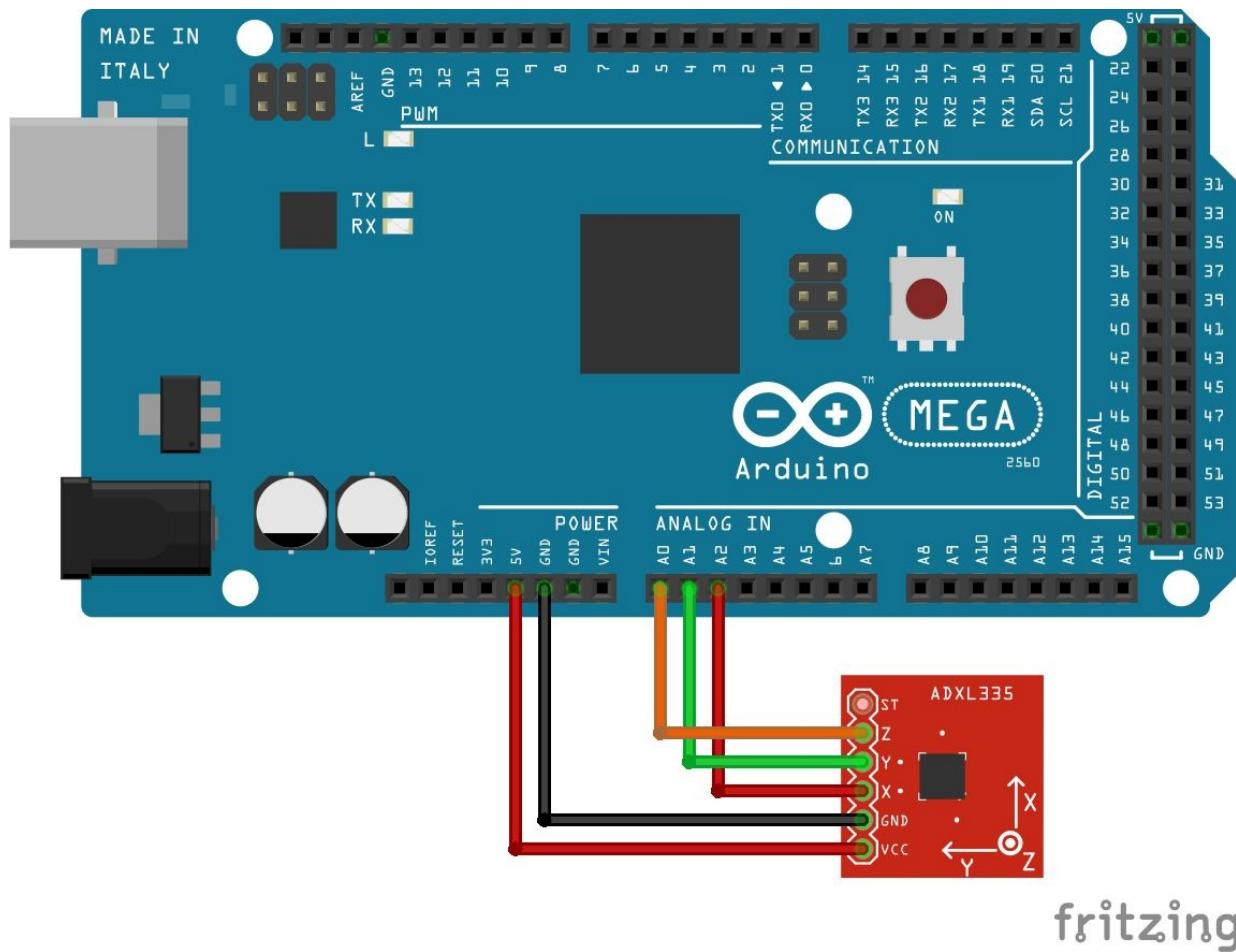


Figure 12 : Interfacing Arduino - ADXL 335

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 eliminate 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:

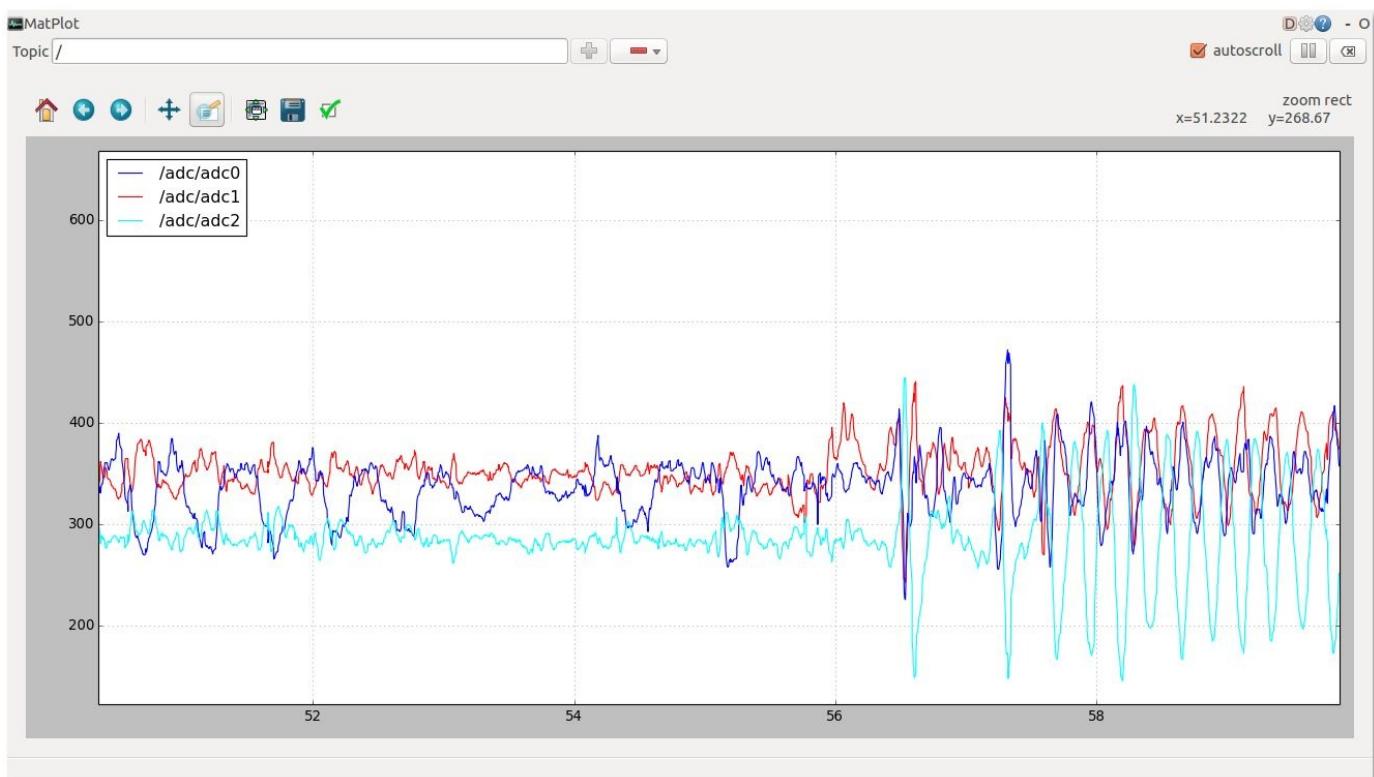


Figure 13 : Plotting ADXL 335 values using rqt_plot

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:

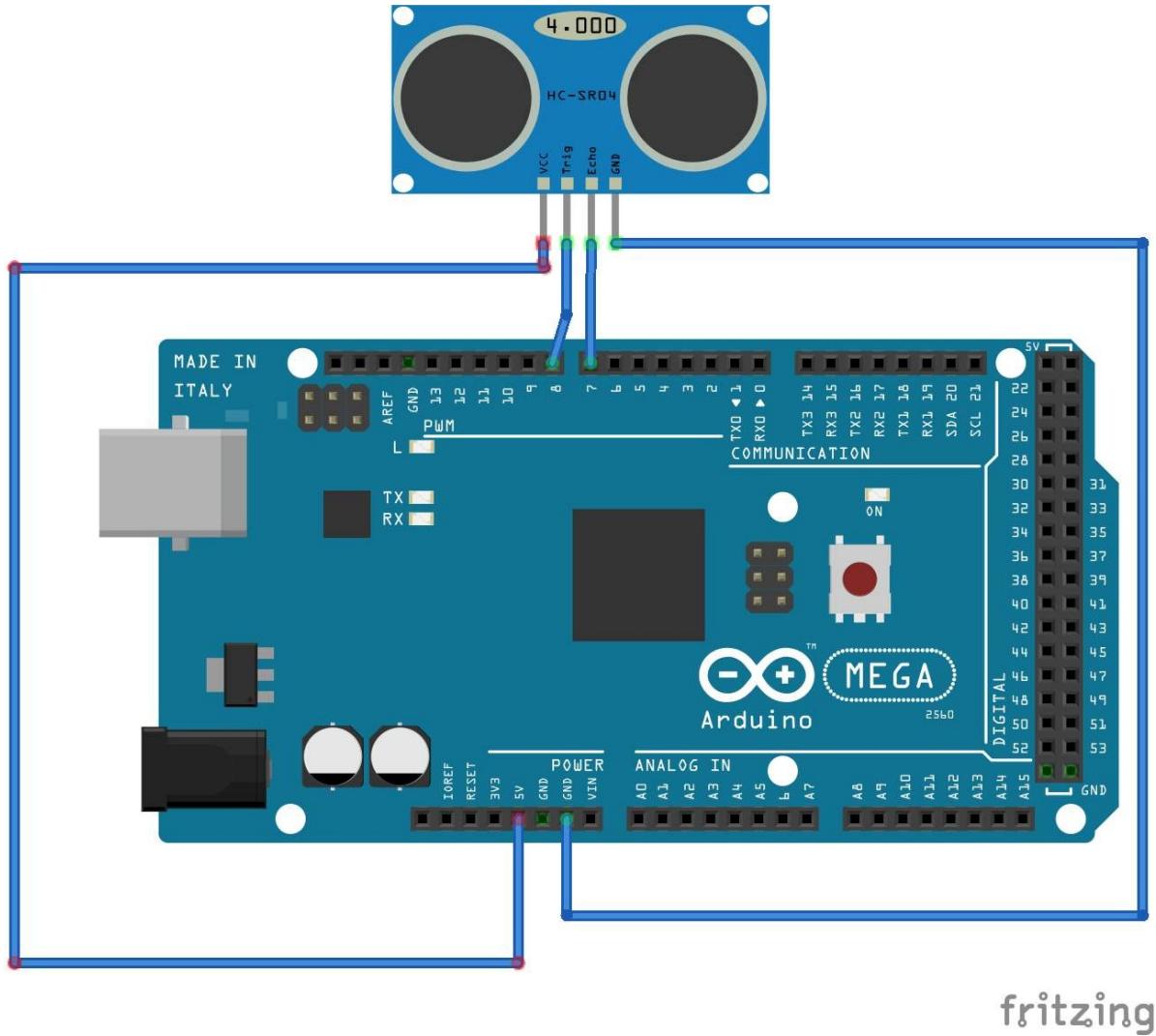


Figure 14 : Plotting ADXL 335 values using rqt_plot

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 = $17150 * \text{Time}$ (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_Ultrasound(){
    int val = 0;
    for(int i=0; i<4; i++) {
        digitalWrite(trigPin, LOW);
        delayMicroseconds(2);
        ... (rest of the code for reading duration)
        ...
        range_msg.duration = duration;
        range_msg.distance = distance;
        pub_range.publish(range_msg);
    }
}
```

```

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
```

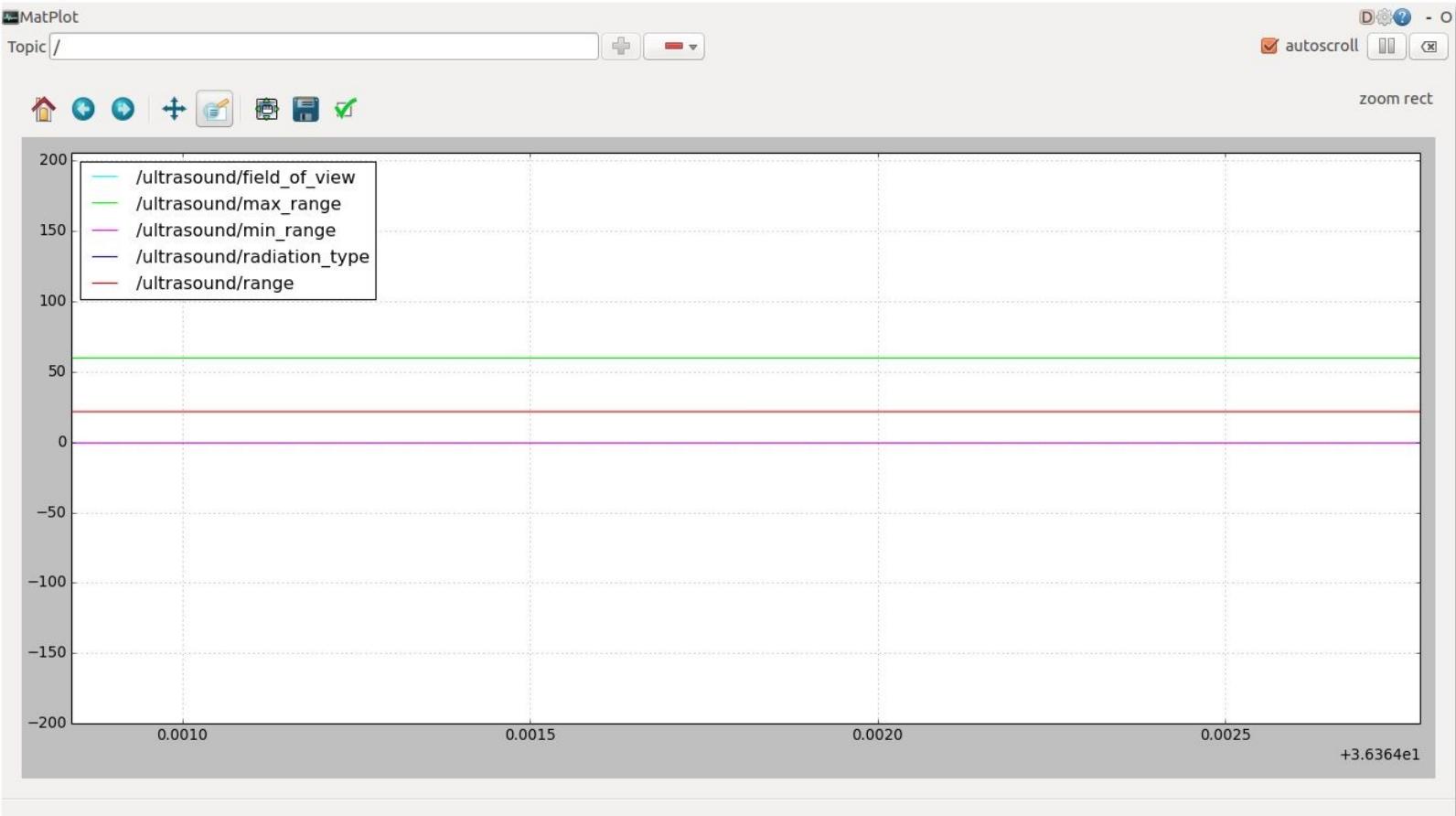


Figure 15 : Plotting ultrasonic sound sensor distance value

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:

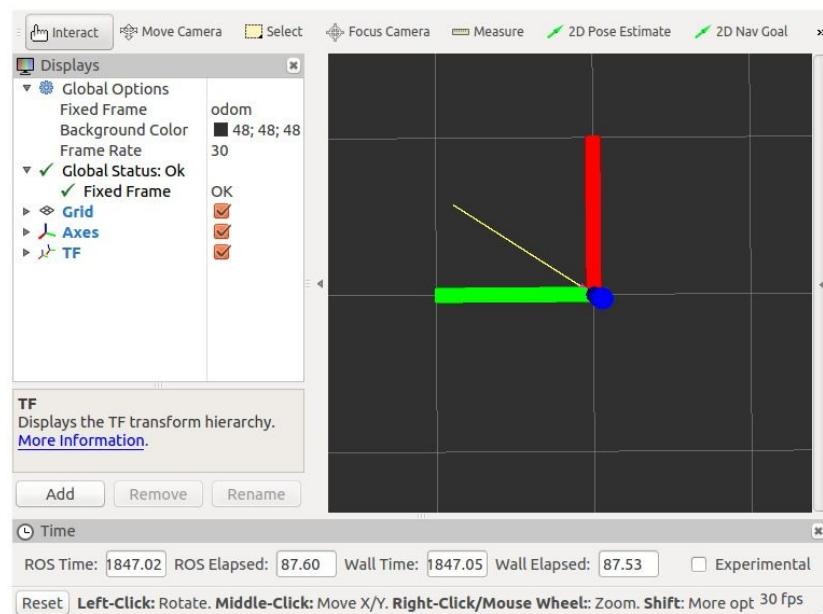


Figure 16 : Visualizing odom data from Arduino

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 16, 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 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:

We can get ROS based images for Raspbian and Ubuntu from the preceding link.
In this section, 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.

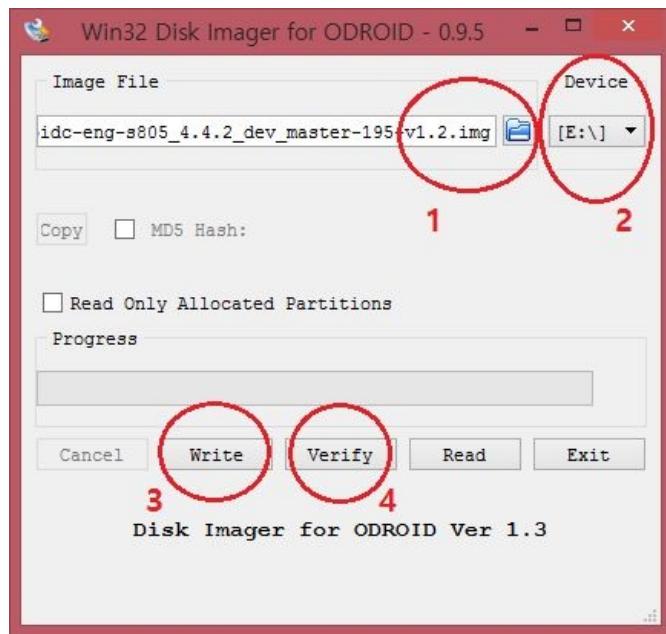


Figure 19 : Win32 Disk Imager for Odroid-C1

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:

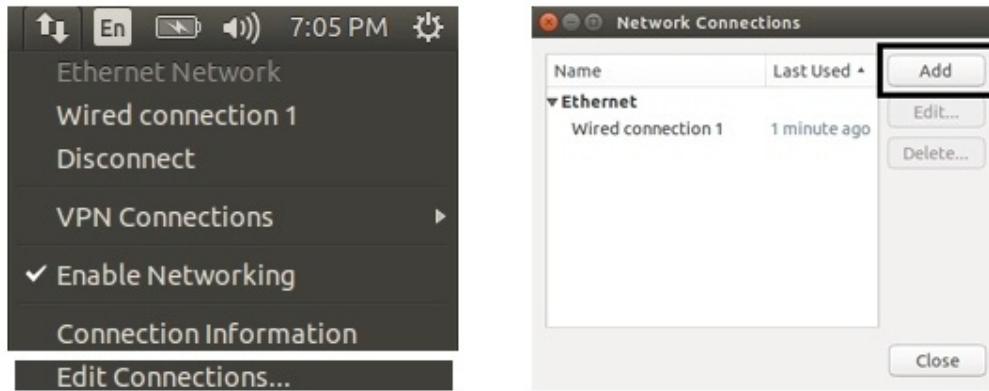


Figure 20 : Configuring a network connection in Ubuntu

- 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:

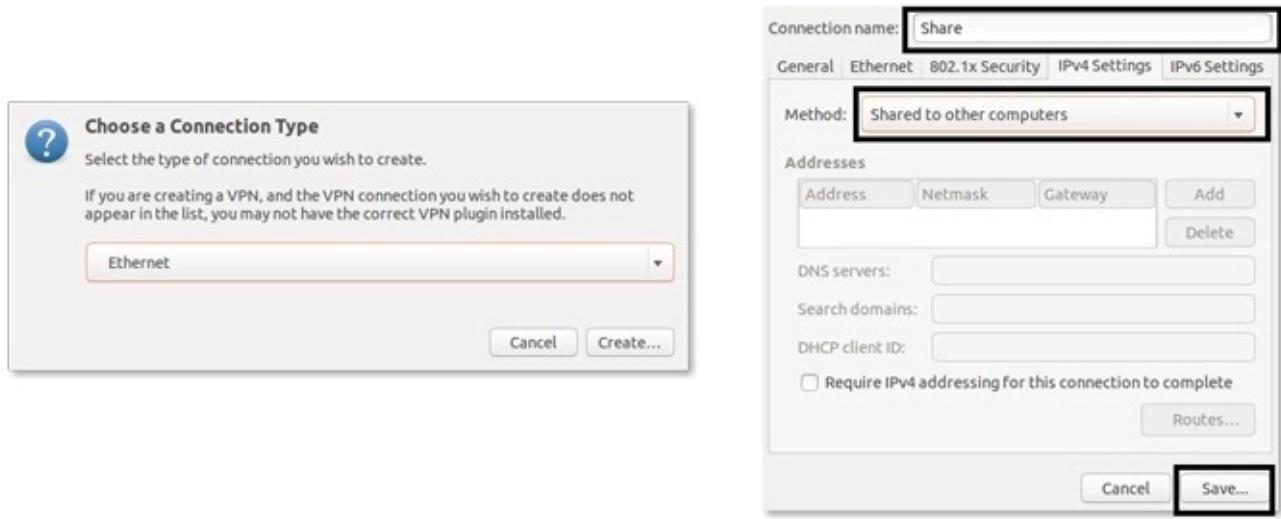
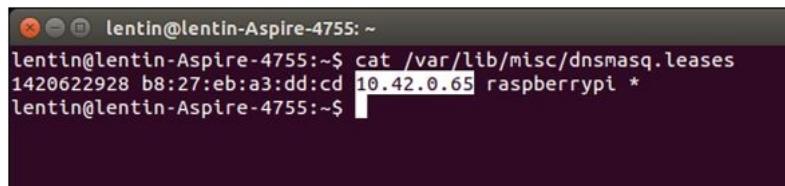


Figure 21 : Creating a new connection for sharing through the Ethernet

- 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: ~
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 | |
| | | 3V3 | | | 1 | 2 | | 5V0 | |
| | | 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 | | 5 |
| 13 | 106 | MISO | GPIOX.BIT9 | MISO | 21 | 22 | #103 | GPIOX.BIT6 | |
| 14 | 105 | SCLK | GPIOX.BIT8 | SCLK | 23 | 24 | CE0 | GPIOX.BIT20 | 117 |
| | | | GND | 25 | 26 | #118 | GPIOX.BIT21 | | 118 |
| | | I2CB_SDA | SDA2 | 27 | 28 | SCL2 | I2CB_SCL | | 11 |
| 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 | | 26 |
| 24 | 97 | | GPIOX.BIT0 | #97 | 35 | 36 | #98 | GPIOX.BIT1 | 98 |
| | | | ADC.AIN1 | AIN1 | 37 | 38 | 1V8 | 1V8 | 27 |
| | | | GND | 39 | 40 | AIN0 | 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://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.

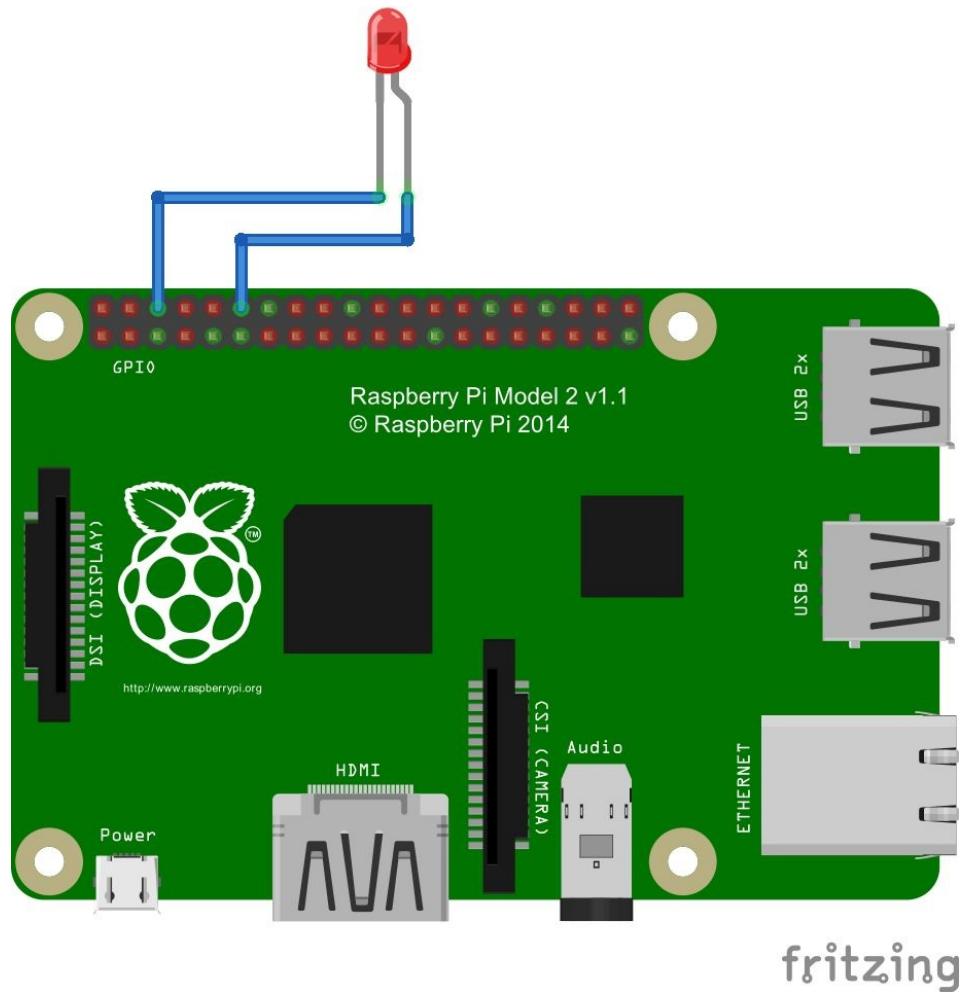


Figure 25 : Blinking an LED using Raspberry Pi 2

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.

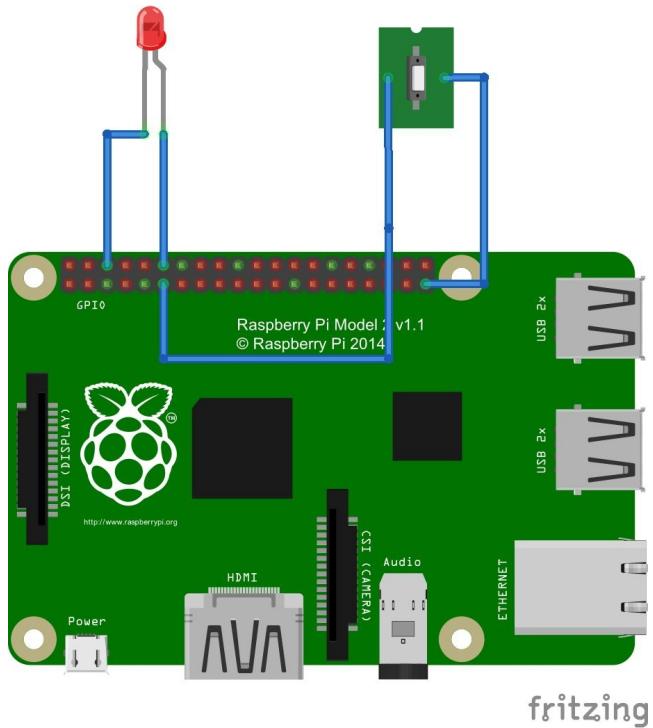


Figure 26 : LED + button in Raspberry Pi 2

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.

Programming Vision Sensors using ROS, OpenCV, 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 packages 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/video0` 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:

```
&lt;launch>
  &lt;node name="usb_cam" pkg="usb_cam" type="usb_cam_node" output="screen" >
    &lt;param name="video_device" value="/dev/video0" />
    &lt;param name="image_width" value="640" />
    &lt;param name="image_height" value="480" />
    &lt;param name="pixel_format" value="yuyv" />
    &lt;param name="camera_frame_id" value="usb_cam" />
    &lt;param name="io_method" value="mmap"/>
  &lt;/node>

  &lt;!-- Launching image_view node -->
  &lt;node name="image_view" pkg="image_view" type="image_view" respawn="false" output="screen">
    &lt;remap from="image" to="/usb_cam/image_raw"/>
    &lt;param name="autosize" value="true" />
  &lt;/node>
&lt;/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:

```
setting /run_id to a2751c16-4719-11e5-87e2-9439e54d7dda
process[rosout-1]: started with pid [7193]
started core service [/rosout]
process[usb_cam-2]: started with pid [7210]
process[image_view-3]: started with pid [7238]
[ INFO] [1440061115.268901882]: Using transport "raw"
[ INFO] [1440061115.444921009]: using default calibration URL
[ INFO] [1440061115.445240220]: camera calibration URL: file:///home/lentin/.ros
/camera_info/head_camera.yaml
[ INFO] [1440061115.445450655]: Unable to open camera calibration file [/home/le
ntin/.ros/camera_info/head_camera.yaml]
[ WARN] [1440061115.445595230]: Camera calibration file /home/lentin/.ros/camera
_info/head_camera.yaml not found.
[ INFO] [1440061115.445742030]: Starting 'head_camera' (/dev/video0) at 640x480
via mmap (yuyv) at 30 FPS
[ WARN] [1440061115.531282236]: unknown control 'white_balance_temperature_auto'
[ WARN] [1440061115.538576973]: unknown control 'focus_auto'
```

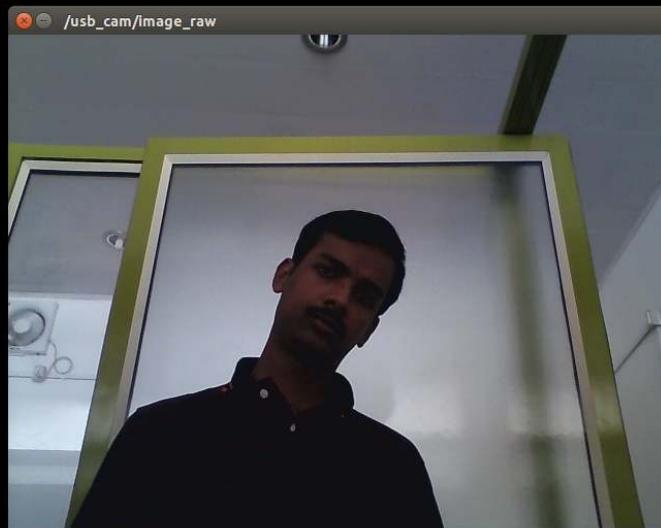


Figure 1 : USB camera view using image view tool

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:

```
$ rosrun 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:

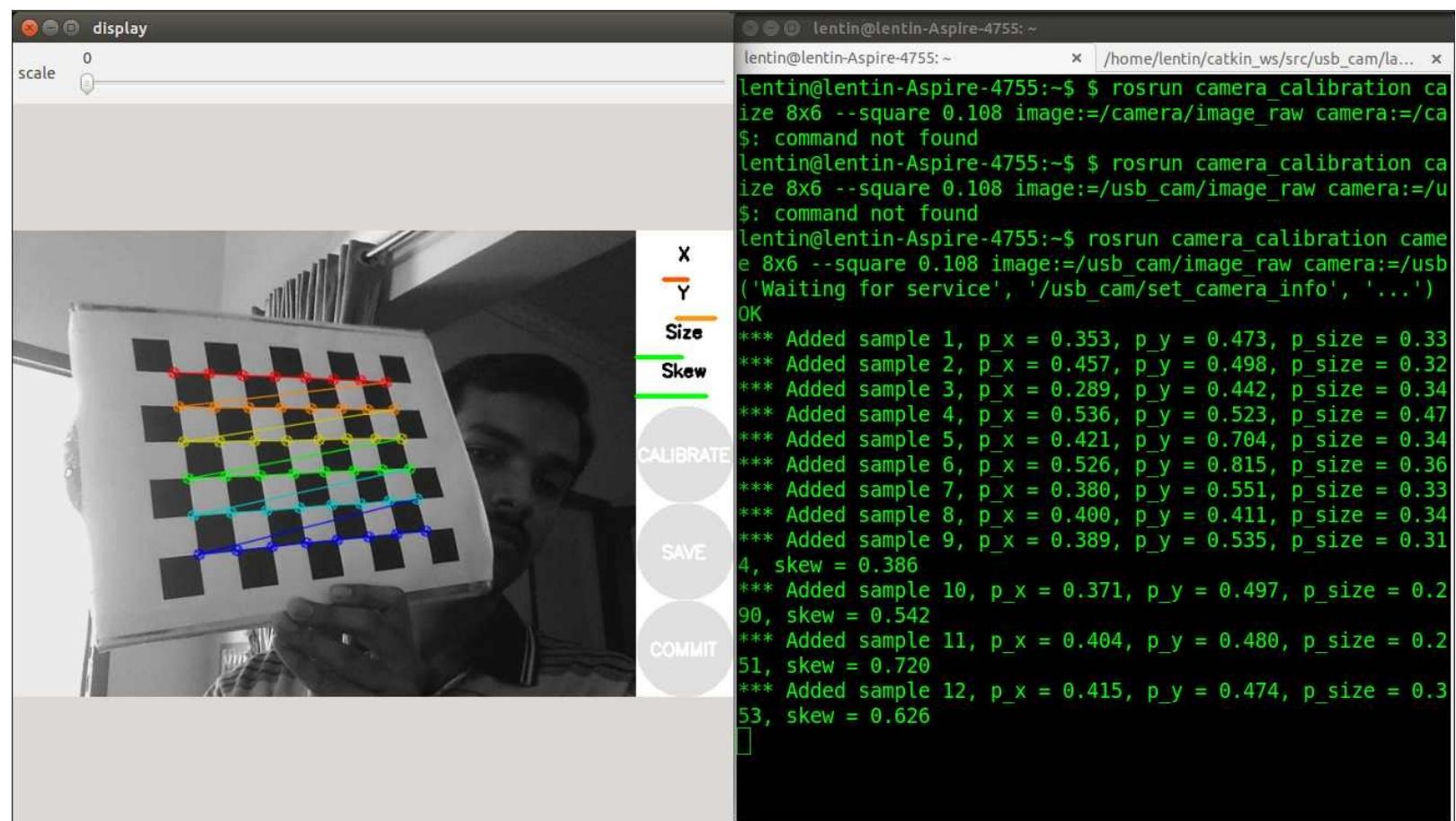


Figure 3: ROS camera calibration

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`.

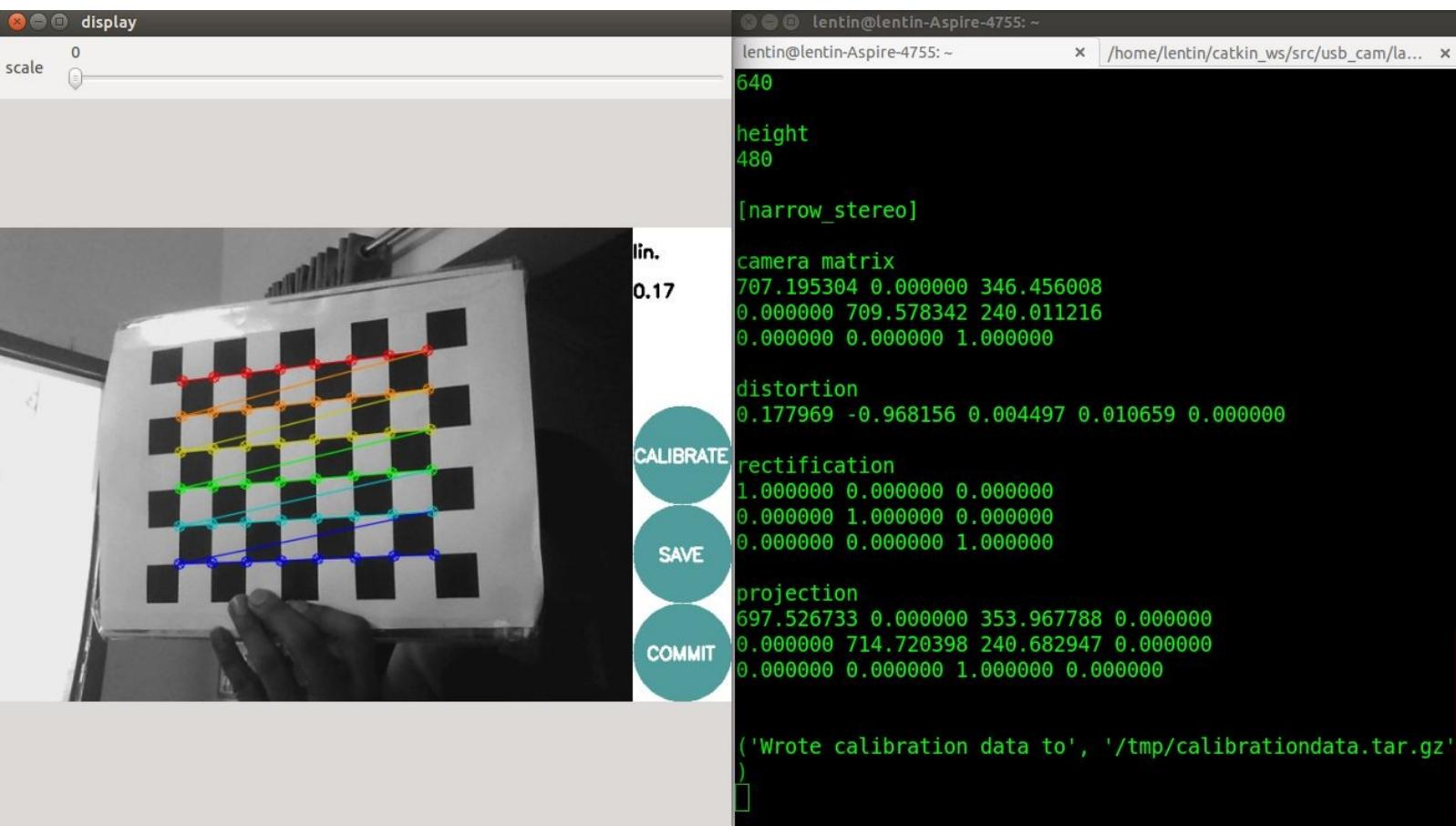


Figure 4 : Generating camera calibration file

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:

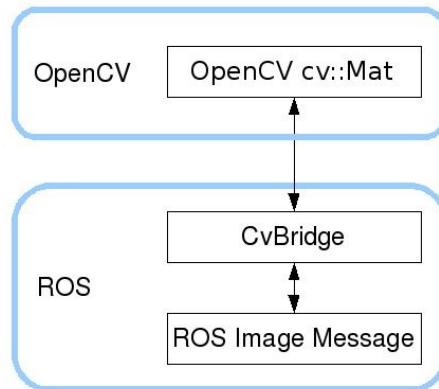


Figure 5 : Converting images using CvBridge

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:

```
| $ rosrun 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.

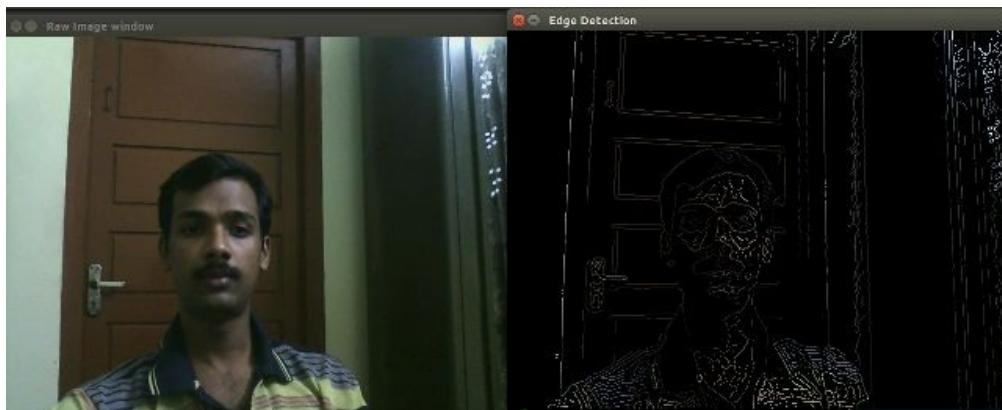


Figure 6 : Raw image and edge detected image

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:

```
| $ roslaunch 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 `lib-freenect`. 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.

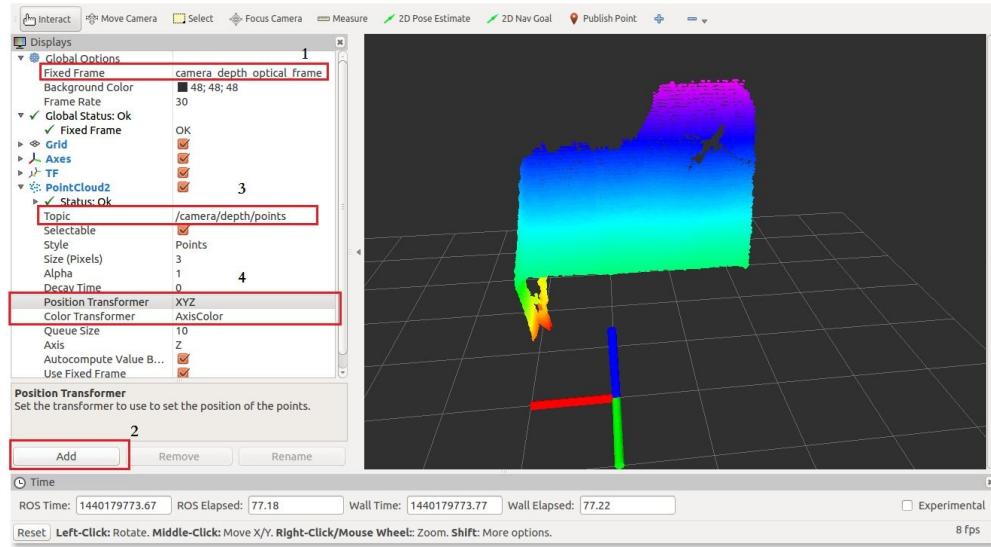


Figure 8: Unregistered point cloud view in RViz

We can enable the registered point cloud by using Dynamic Reconfigure GUI, by using the following command:

```
| $ rosrun rqt_reconfigure rqt_reconfigure
```

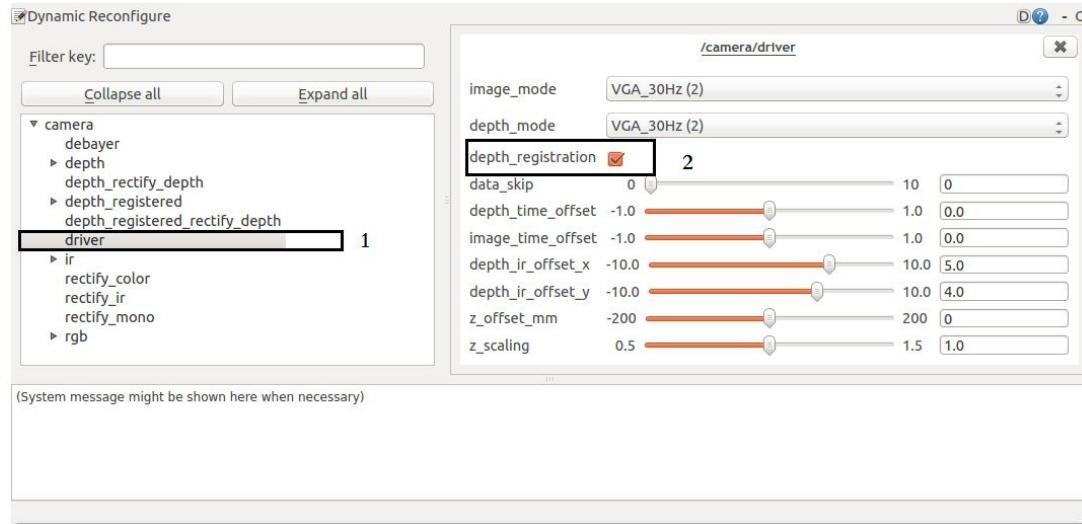


Figure 9: Dynamic Reconfigure GUI

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.

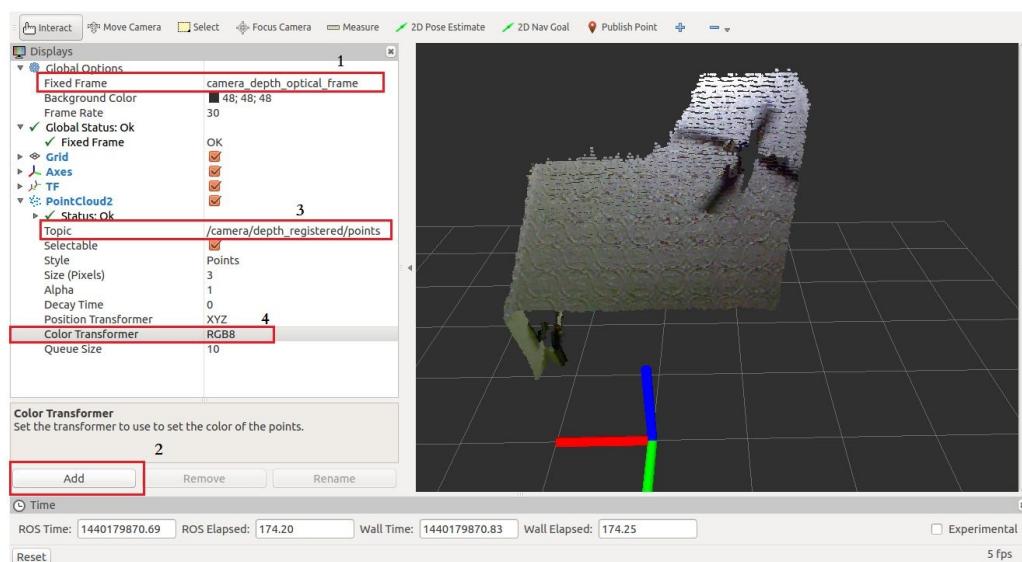


Figure 10: The registered point cloud

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 Real Sense

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
```

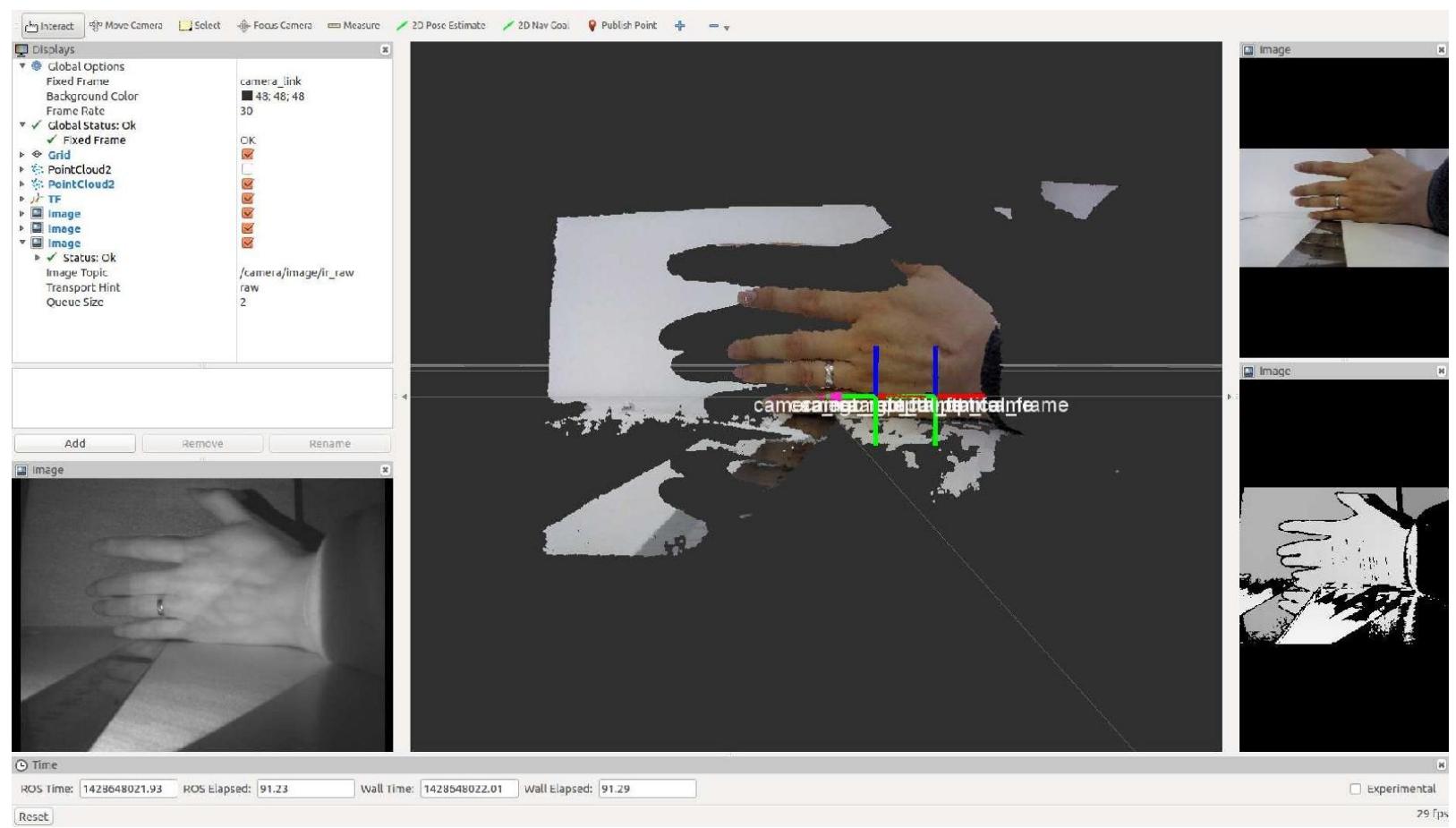


Figure 12: Intel Real Sense view in RViz

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
```

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.

```
&lt;launch>
  &lt;!-- "camera" should uniquely identify the device. All topics
       are pushed down      into the "camera" namespace, and it is prepended to tf
       frame ids. -->
  &lt;arg name="camera"      default="camera"/>
  &lt;arg name="publish_tf"  default="true"/>
  .....
  .....

  &lt;group if="$(arg scan_processing)">
    &lt;node pkg="nodelet" type="nodelet"
      name="depthimage_to_laserscan" args="load
      depthimage_to_laserscan/DepthImageToLaserScanNodelet $(arg
      camera)/$(arg camera)_nodelet_manager">      &lt;!-- 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. -->
    &lt;param name="scan_height" value="10"/>
    &lt;param name="output_frame_id" value="/$(arg
      camera)_depth_frame"/>
    &lt;param name="range_min" value="0.45"/>
    &lt;remap from="image" to="$(arg camera)/$(arg
      depth)/image_raw"/>
    &lt;remap from="scan" to="$(arg scan_topic)"/>
    .....
  .....

  &lt;/group>
  &lt;/launch>
```

The following code snippet will launch the nodelet for converting the depth image to laser scanner:

```
&lt;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:

```
| $ roslaunch fake_laser_pkg start_laser.launch
```

We will see the data in RViz, as shown in the following image:

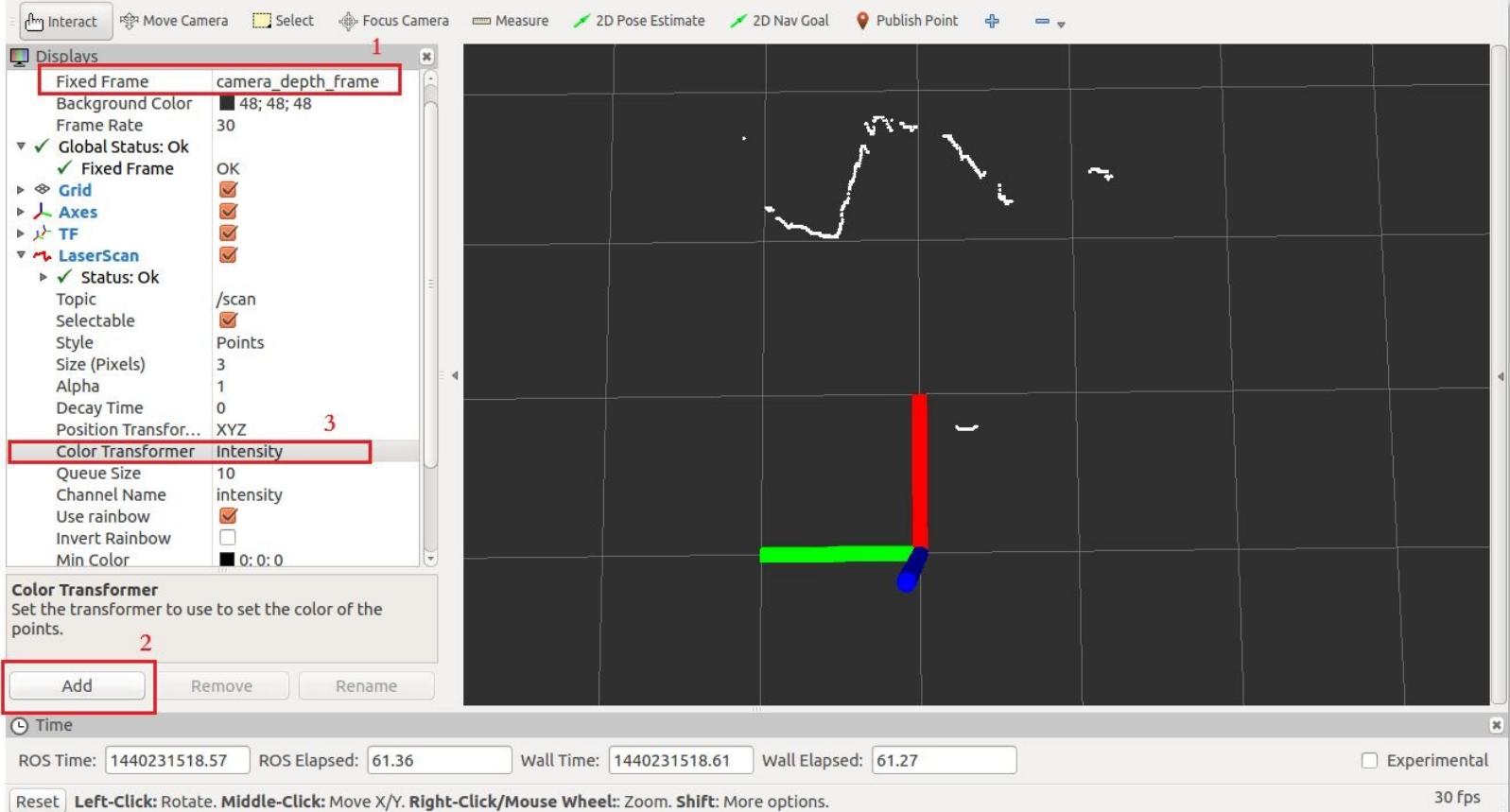


Figure 13: Laser scan in RViz

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+r /dev/ttyACM0
```

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:

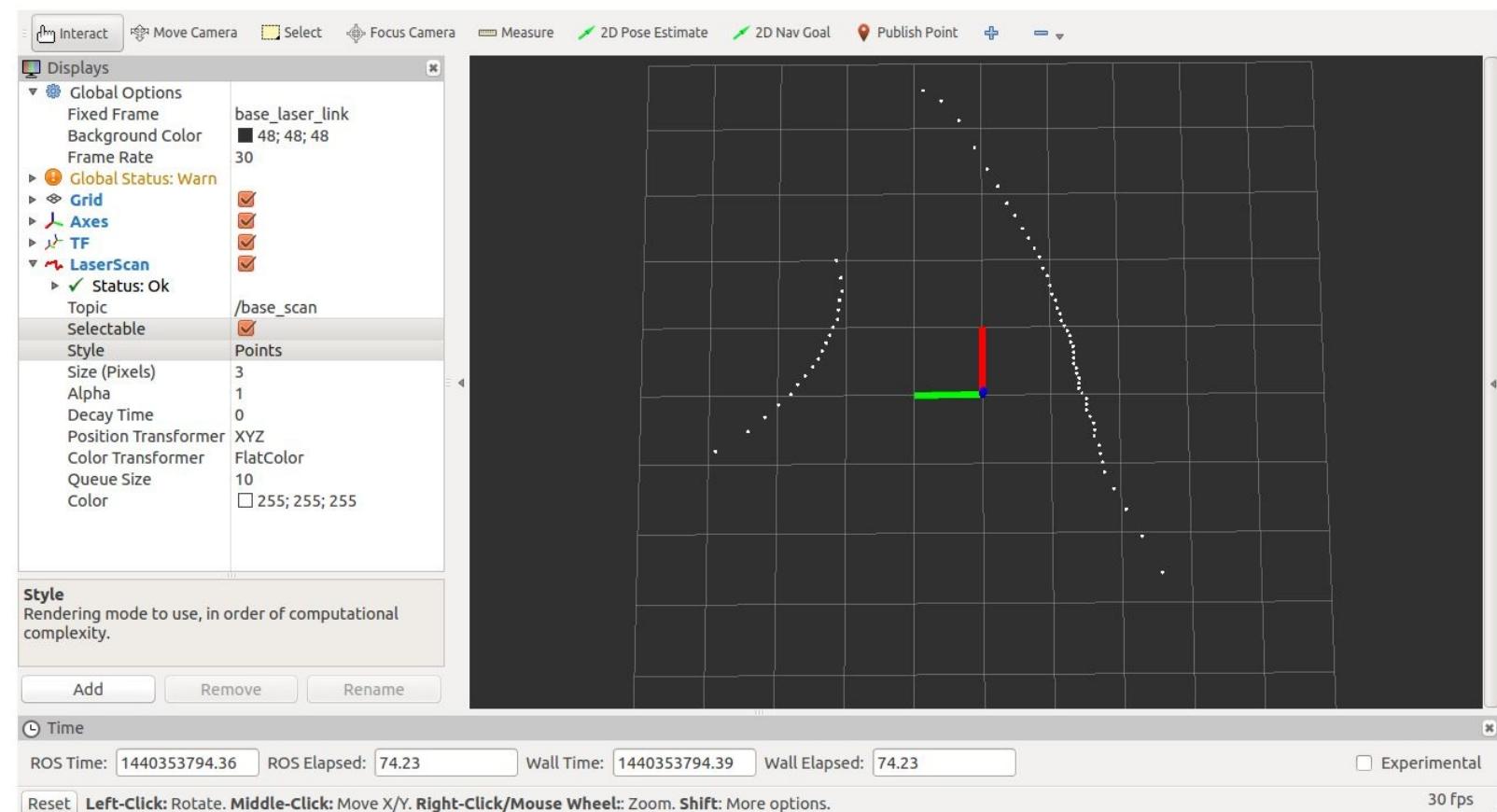


Figure 16: Hokuyo Laser scan data in RViz

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
```

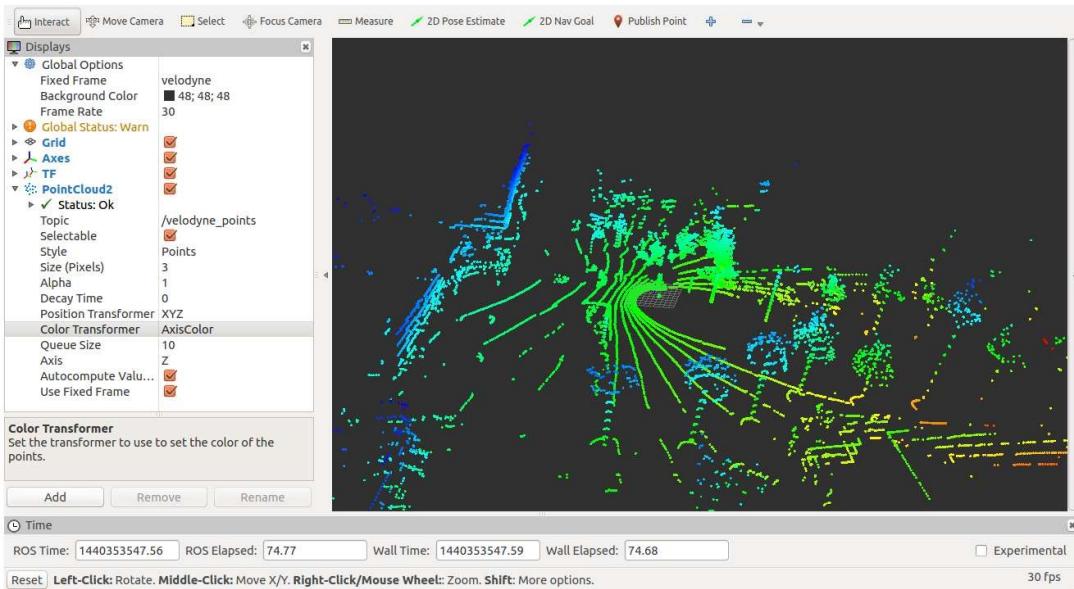


Figure 18: Velodyne point cloud view in RViz

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<pcl::PointXYZ> 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<pcl::PointXYZ> 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:

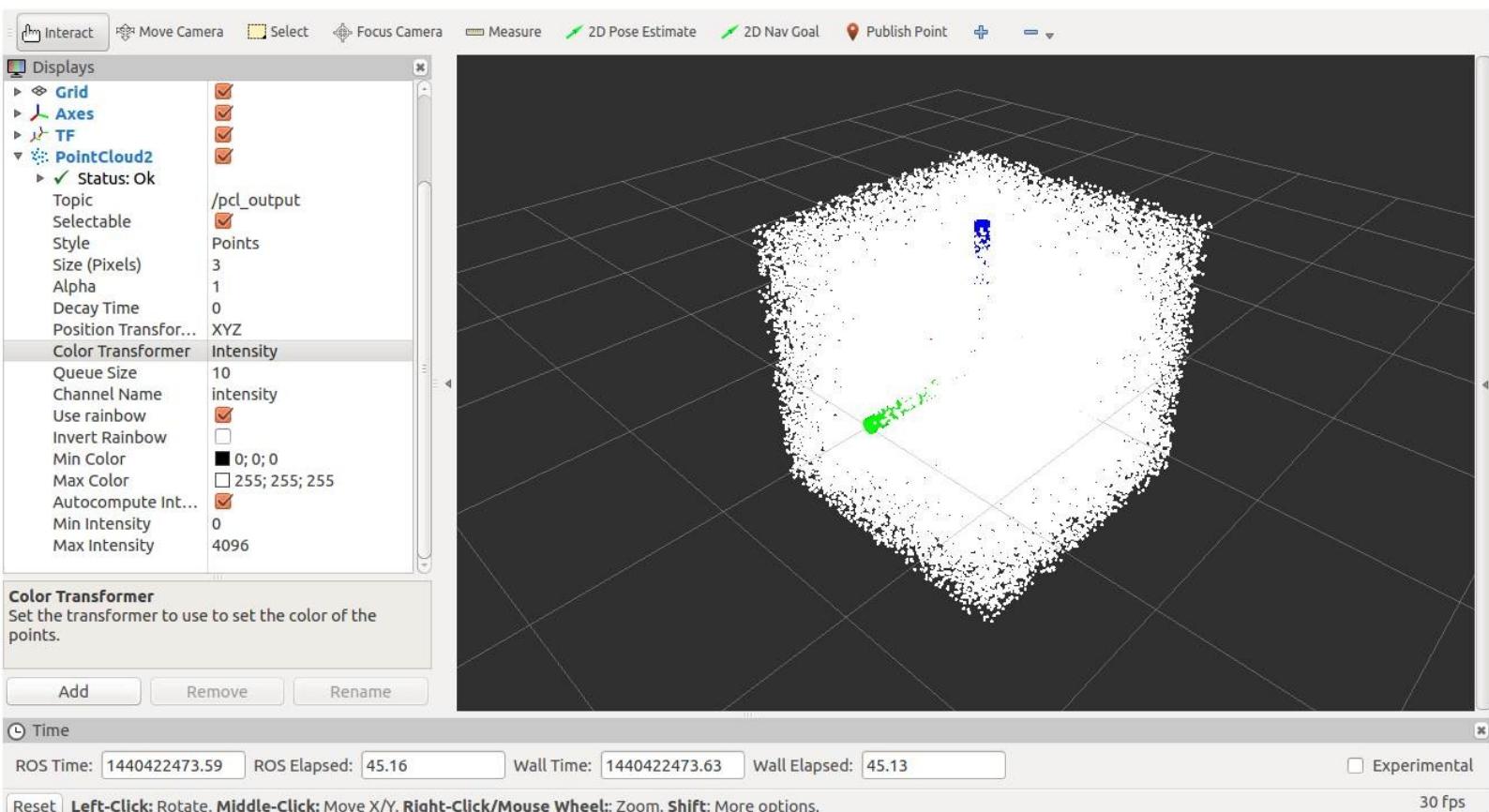


Figure 19: PCL cloud in RViz

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:

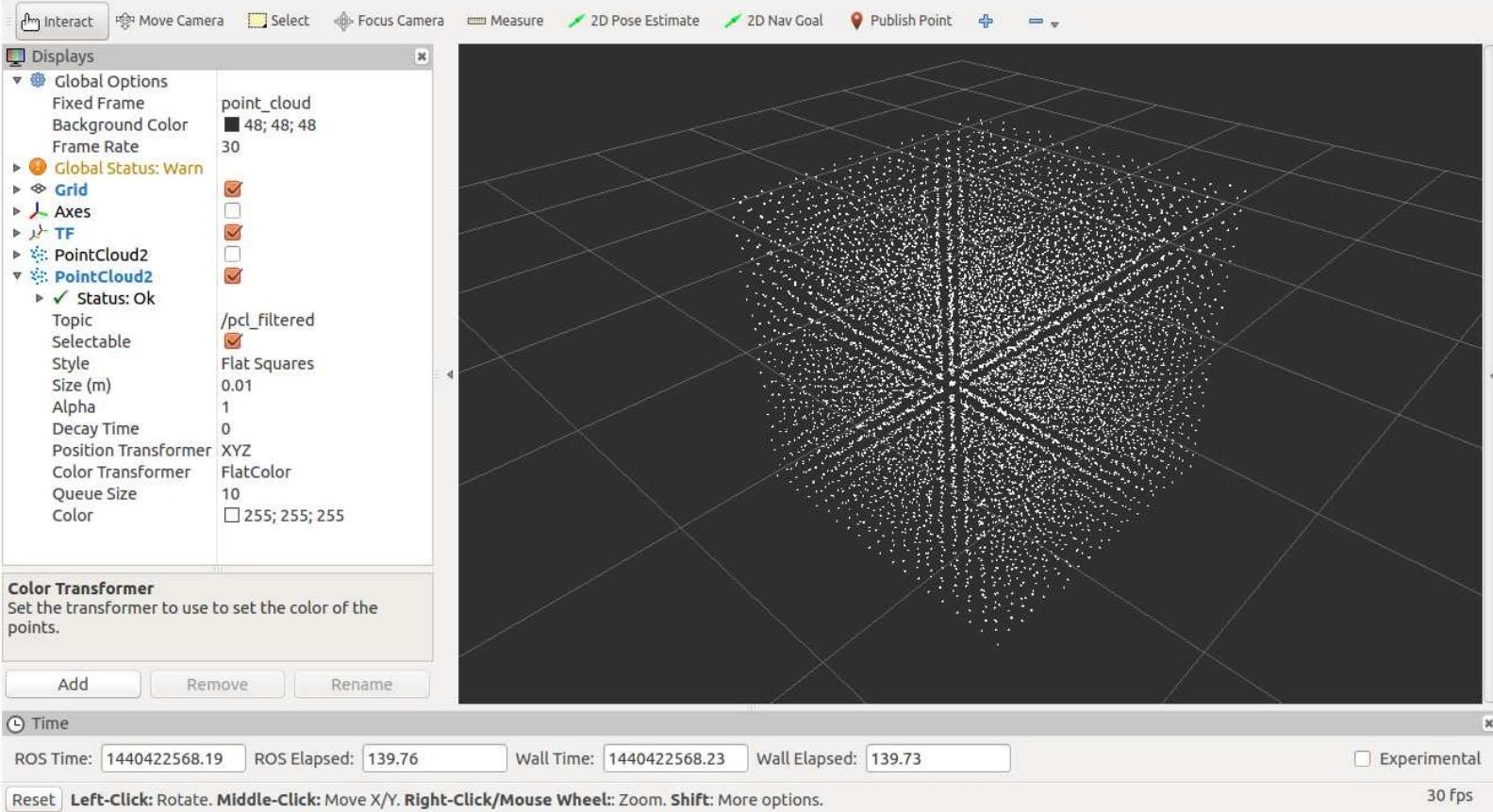


Figure 20 : Filtered PCL cloud in RViz

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:

```
| $ rosrun usb_cam usb_cam-test.launch
```

```

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:

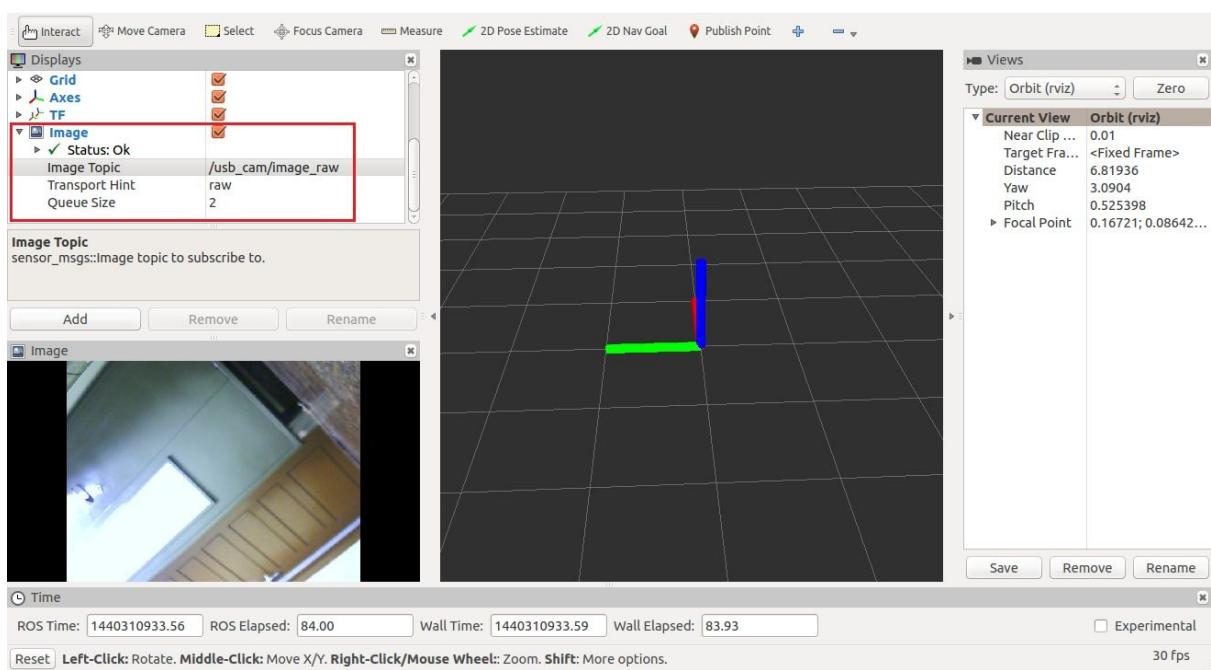


Figure 23 : Odroid camera view in RViz running on PC



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_pc1`. 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.

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 section 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. 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

The following are the mandatory requirements for interfacing a mobile robot with ROS navigation package:

- **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-tm4c123gxl>): 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 an 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:

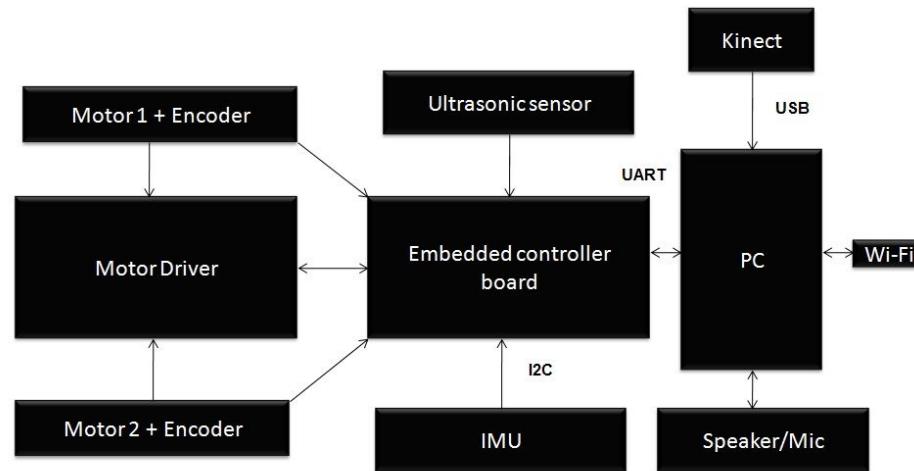


Figure 2: Block diagram of Chefbot

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:

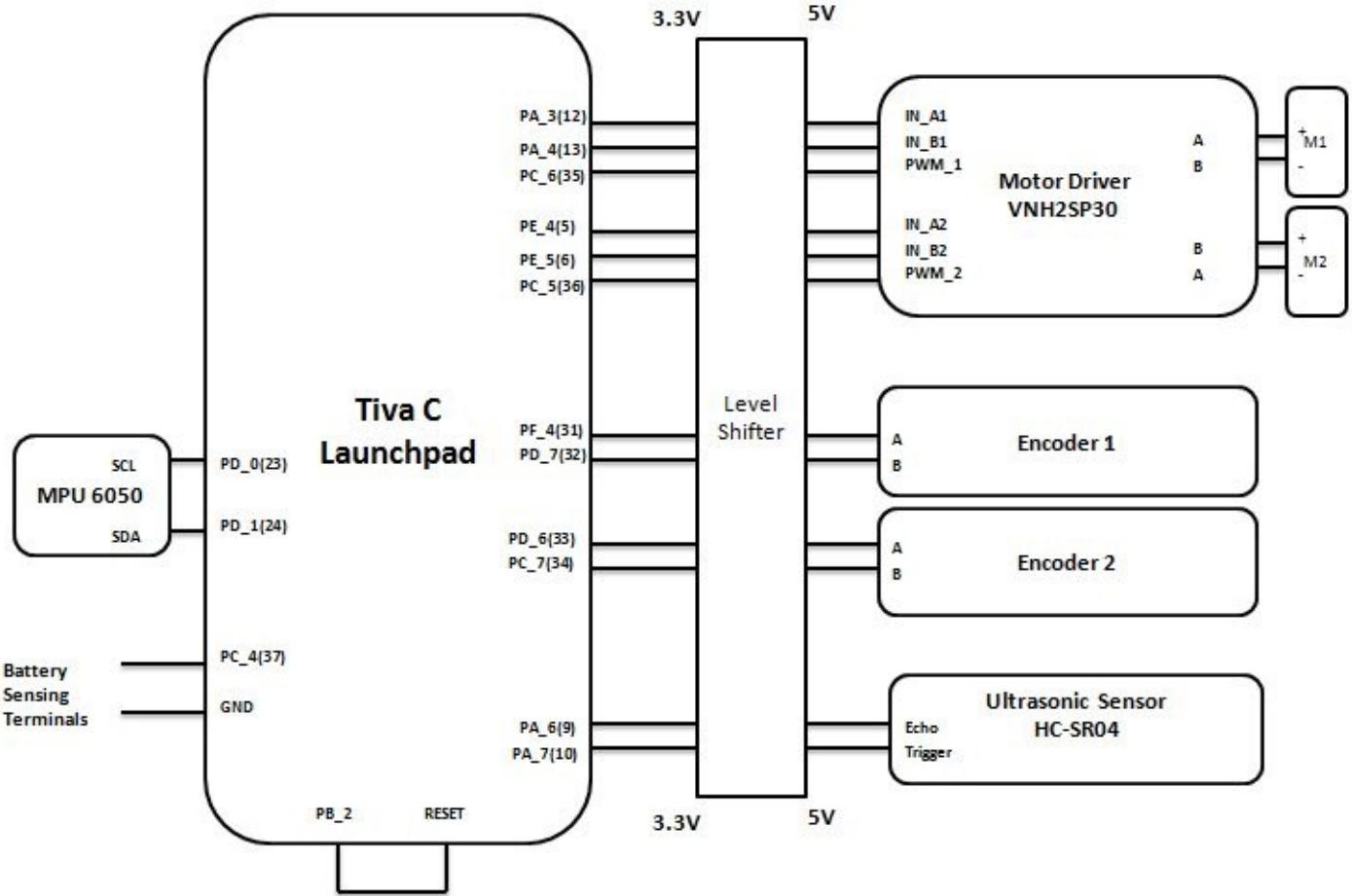


Figure 3: Connection diagram of Chefbot

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>
| &lt;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>&lt;pwm_value_of_motor_1>&lt;space>&lt;pwm_value_of_motor_2>
```

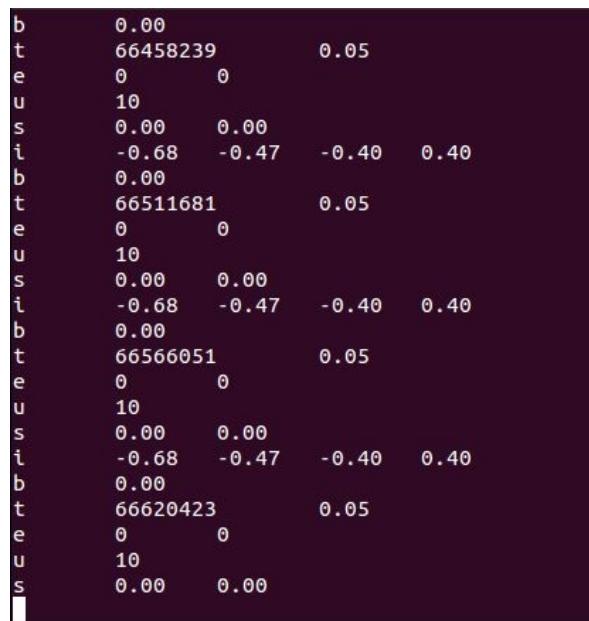
For resetting the device, the protocol will be as follows:

```
| 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:



| Time (t) | PWM Value (b) |
|----------|---------------|
| 66458239 | 0.05 |
| 66458240 | 0.05 |
| 66458241 | 0.05 |
| 66458242 | 0.05 |
| 66458243 | 0.05 |
| 66458244 | 0.05 |
| 66458245 | 0.05 |
| 66458246 | 0.05 |
| 66458247 | 0.05 |
| 66458248 | 0.05 |
| 66458249 | 0.05 |
| 66458250 | 0.05 |
| 66458251 | 0.05 |
| 66458252 | 0.05 |
| 66458253 | 0.05 |
| 66458254 | 0.05 |
| 66458255 | 0.05 |
| 66458256 | 0.05 |
| 66458257 | 0.05 |
| 66458258 | 0.05 |
| 66458259 | 0.05 |
| 66458260 | 0.05 |
| 66458261 | 0.05 |
| 66458262 | 0.05 |
| 66458263 | 0.05 |
| 66458264 | 0.05 |
| 66458265 | 0.05 |
| 66458266 | 0.05 |
| 66458267 | 0.05 |
| 66458268 | 0.05 |
| 66458269 | 0.05 |
| 66458270 | 0.05 |
| 66458271 | 0.05 |
| 66458272 | 0.05 |
| 66458273 | 0.05 |
| 66458274 | 0.05 |
| 66458275 | 0.05 |
| 66458276 | 0.05 |
| 66458277 | 0.05 |
| 66458278 | 0.05 |
| 66458279 | 0.05 |
| 66458280 | 0.05 |
| 66458281 | 0.05 |
| 66458282 | 0.05 |
| 66458283 | 0.05 |
| 66458284 | 0.05 |
| 66458285 | 0.05 |
| 66458286 | 0.05 |
| 66458287 | 0.05 |
| 66458288 | 0.05 |
| 66458289 | 0.05 |
| 66458290 | 0.05 |
| 66458291 | 0.05 |
| 66458292 | 0.05 |
| 66458293 | 0.05 |
| 66458294 | 0.05 |
| 66458295 | 0.05 |
| 66458296 | 0.05 |
| 66458297 | 0.05 |
| 66458298 | 0.05 |
| 66458299 | 0.05 |
| 66458300 | 0.05 |
| 66458301 | 0.05 |
| 66458302 | 0.05 |
| 66458303 | 0.05 |
| 66458304 | 0.05 |
| 66458305 | 0.05 |
| 66458306 | 0.05 |
| 66458307 | 0.05 |
| 66458308 | 0.05 |
| 66458309 | 0.05 |
| 66458310 | 0.05 |
| 66458311 | 0.05 |
| 66458312 | 0.05 |
| 66458313 | 0.05 |
| 66458314 | 0.05 |
| 66458315 | 0.05 |
| 66458316 | 0.05 |
| 66458317 | 0.05 |
| 66458318 | 0.05 |
| 66458319 | 0.05 |
| 66458320 | 0.05 |
| 66458321 | 0.05 |
| 66458322 | 0.05 |
| 66458323 | 0.05 |
| 66458324 | 0.05 |
| 66458325 | 0.05 |
| 66458326 | 0.05 |
| 66458327 | 0.05 |
| 66458328 | 0.05 |
| 66458329 | 0.05 |
| 66458330 | 0.05 |
| 66458331 | 0.05 |
| 66458332 | 0.05 |
| 66458333 | 0.05 |
| 66458334 | 0.05 |
| 66458335 | 0.05 |
| 66458336 | 0.05 |
| 66458337 | 0.05 |
| 66458338 | 0.05 |
| 66458339 | 0.05 |
| 66458340 | 0.05 |
| 66458341 | 0.05 |
| 66458342 | 0.05 |
| 66458343 | 0.05 |
| 66458344 | 0.05 |
| 66458345 | 0.05 |
| 66458346 | 0.05 |
| 66458347 | 0.05 |
| 66458348 | 0.05 |
| 66458349 | 0.05 |
| 66458350 | 0.05 |
| 66458351 | 0.05 |
| 66458352 | 0.05 |
| 66458353 | 0.05 |
| 66458354 | 0.05 |
| 66458355 | 0.05 |
| 66458356 | 0.05 |
| 66458357 | 0.05 |
| 66458358 | 0.05 |
| 66458359 | 0.05 |
| 66458360 | 0.05 |
| 66458361 | 0.05 |
| 66458362 | 0.05 |
| 66458363 | 0.05 |
| 66458364 | 0.05 |
| 66458365 | 0.05 |
| 66458366 | 0.05 |
| 66458367 | 0.05 |
| 66458368 | 0.05 |
| 66458369 | 0.05 |
| 66458370 | 0.05 |
| 66458371 | 0.05 |
| 66458372 | 0.05 |
| 66458373 | 0.05 |
| 66458374 | 0.05 |
| 66458375 | 0.05 |
| 66458376 | 0.05 |
| 66458377 | 0.05 |
| 66458378 | 0.05 |
| 66458379 | 0.05 |
| 66458380 | 0.05 |
| 66458381 | 0.05 |
| 66458382 | 0.05 |
| 66458383 | 0.05 |
| 66458384 | 0.05 |
| 66458385 | 0.05 |
| 66458386 | 0.05 |
| 66458387 | 0.05 |
| 66458388 | 0.05 |
| 66458389 | 0.05 |
| 66458390 | 0.05 |
| 66458391 | 0.05 |
| 66458392 | 0.05 |
| 66458393 | 0.05 |
| 66458394 | 0.05 |
| 66458395 | 0.05 |
| 66458396 | 0.05 |
| 66458397 | 0.05 |
| 66458398 | 0.05 |
| 66458399 | 0.05 |
| 66458400 | 0.05 |
| 66458401 | 0.05 |
| 66458402 | 0.05 |
| 66458403 | 0.05 |
| 66458404 | 0.05 |
| 66458405 | 0.05 |
| 66458406 | 0.05 |
| 66458407 | 0.05 |
| 66458408 | 0.05 |
| 66458409 | 0.05 |
| 66458410 | 0.05 |
| 66458411 | 0.05 |
| 66458412 | 0.05 |
| 66458413 | 0.05 |
| 66458414 | 0.05 |
| 66458415 | 0.05 |
| 66458416 | 0.05 |
| 66458417 | 0.05 |
| 66458418 | 0.05 |
| 66458419 | 0.05 |
| 66458420 | 0.05 |
| 66458421 | 0.05 |
| 66458422 | 0.05 |
| 66458423 | 0.05 |
| 66458424 | 0.05 |
| 66458425 | 0.05 |
| 66458426 | 0.05 |
| 66458427 | 0.05 |
| 66458428 | 0.05 |
| 66458429 | 0.05 |
| 66458430 | 0.05 |
| 66458431 | 0.05 |
| 66458432 | 0.05 |
| 66458433 | 0.05 |
| 66458434 | 0.05 |
| 66458435 | 0.05 |
| 66458436 | 0.05 |
| 66458437 | 0.05 |
| 66458438 | 0.05 |
| 66458439 | 0.05 |
| 66458440 | 0.05 |
| 66458441 | 0.05 |
| 66458442 | 0.05 |
| 66458443 | 0.05 |
| 66458444 | 0.05 |
| 66458445 | 0.05 |
| 66458446 | 0.05 |
| 66458447 | 0.05 |
| 66458448 | 0.05 |
| 66458449 | 0.05 |
| 66458450 | 0.05 |
| 66458451 | 0.05 |
| 66458452 | 0.05 |
| 66458453 | 0.05 |
| 66458454 | 0.05 |
| 66458455 | 0.05 |
| 66458456 | 0.05 |
| 66458457 | 0.05 |
| 66458458 | 0.05 |
| 66458459 | 0.05 |
| 66458460 | 0.05 |
| 66458461 | 0.05 |
| 66458462 | 0.05 |
| 66458463 | 0.05 |
| 66458464 | 0.05 |
| 66458465 | 0.05 |
| 66458466 | 0.05 |
| 66458467 | 0.05 |
| 66458468 | 0.05 |
| 66458469 | 0.05 |
| 66458470 | 0.05 |
| 66458471 | 0.05 |
| 66458472 | 0.05 |
| 66458473 | 0.05 |
| 66458474 | 0.05 |
| 66458475 | 0.05 |
| 66458476 | 0.05 |
| 66458477 | 0.05 |
| 66458478 | 0.05 |
| 66458479 | 0.05 |
| 66458480 | 0.05 |
| 66458481 | 0.05 |
| 66458482 | 0.05 |
| 66458483 | 0.05 |
| 66458484 | 0.05 |
| 66458485 | 0.05 |
| 66458486 | 0.05 |
| 66458487 | 0.05 |
| 66458488 | 0.05 |
| 66458489 | 0.05 |
| 66458490 | 0.05 |
| 66458491 | 0.05 |
| 66458492 | 0.05 |
| 66458493 | 0.05 |
| 66458494 | 0.05 |
| 66458495 | 0.05 |
| 66458496 | 0.05 |
| 66458497 | 0.05 |
| 66458498 | 0.05 |
| 66458499 | 0.05 |
| 66458500 | 0.05 |
| 66458501 | 0.05 |
| 66458502 | 0.05 |
| 66458503 | 0.05 |
| 66458504 | 0.05 |
| 66458505 | 0.05 |
| 66458506 | 0.05 |
| 66458507 | 0.05 |
| 66458508 | 0.05 |
| 66458509 | 0.05 |
| 66458510 | 0.05 |
| 66458511 | 0.05 |
| 66458512 | 0.05 |
| 66458513 | 0.05 |
| 66458514 | 0.05 |
| 66458515 | 0.05 |
| 66458516 | 0.05 |
| 66458517 | 0.05 |
| 66458518 | 0.05 |
| 66458519 | 0.05 |
| 66458520 | 0.05 |
| 66458521 | 0.05 |
| 66458522 | 0.05 |
| 66458523 | 0.05 |
| 66458524 | 0.05 |
| 66458525 | 0.05 |
| 66458526 | 0.05 |
| 66458527 | 0.05 |
| 66458528 | 0.05 |
| 66458529 | 0.05 |
| 66458530 | 0.05 |
| 66458531 | 0.05 |
| 66458532 | 0.05 |
| 66458533 | 0.05 |
| 66458534 | 0.05 |
| 66458535 | 0.05 |
| 66458536 | 0.05 |
| 66458537 | 0.05 |
| 66458538 | 0.05 |
| 66458539 | 0.05 |
| 66458540 | 0.05 |
| 66458541 | 0.05 |
| 66458542 | 0.05 |
| 66458543 | 0.05 |
| 66458544 | 0.05 |
| 66458545 | 0.05 |
| 66458546 | 0.05 |
| 66458547 | 0.05 |
| 66458548 | 0.05 |
| 66458549 | 0.05 |
| 66458550 | 0.05 |
| 66458551 | 0.05 |
| 66458552 | 0.05 |
| 66458553 | 0.05 |
| 66458554 | 0.05 |
| 66458555 | 0.05 |
| 66458556 | 0.05 |
| 66458557 | 0.05 |
| 66458558 | 0.05 |
| 66458559 | 0.05 |
| 66458560 | 0.05 |
| 66458561 | 0.05 |
| 66458562 | 0.05 |
| 66458563 | 0.05 |
| 66458564 | 0.05 |
| 66458565 | 0.05 |
| 66458566 | 0.05 |
| 66458567 | 0.05 |
| 66458568 | 0.05 |
| 66458569 | 0.05 |
| 66458570 | 0.05 |
| 66458571 | 0.05 |
| 66458572 | 0.05 |
| 66458573 | 0.05 |
| 66458574 | 0.05 |
| 66458575 | 0.05 |
| 66458576 | 0.05 |
| 66458577 | 0.05 |
| 66458578 | 0.05 |
| 66458579 | 0.05 |
| 66458580 | 0.05 |
| 66458581 | 0.05 |
| 66458582 | 0.05 |
| 66458583 | 0.05 |
| 66458584 | 0.05 |
| 66458585 | 0.05 |
| 66458586 | 0.05 |
| 66458587 | 0.05 |
| 66458588 | 0.05 |
| 66458589 | 0.05 |
| 66458590 | 0.05 |
| 66458591 | 0.05 |
| 66458592 | 0.05 |
| 66458593 | 0.05 |
| 66458594 | 0.05 |
| 66458595 | 0.05 |
| 66458596 | 0.05 |
| 66458597 | 0.05 |
| 66458598 | 0.05 |
| 66458599 | 0.05 |
| 66458600 | 0.05 |
| 66458601 | 0.05 |
| 66458602 | 0.05 |
| 66458603 | 0.05 |
| 66458604 | 0.05 |
| 66458605 | 0.05 |
| 66458606 | 0.05 |
| 66458607 | 0.05 |
| 66458608 | 0.05 |
| 66458609 | 0.05 |
| 66458610 | 0.05 |
| 66458611 | 0.05 |
| 66458612 | 0.05 |
| 66458613 | 0.05 |
| 66458614 | 0.05 |
| 66458615 | 0.05 |
| 66458616 | 0.05 |
| 66458617 | 0.05 |
| 66458618 | 0.05 |
| 66458619 | 0.05 |
| 66458620 | 0.05 |
| 66458621 | 0.05 |
| 66458622 | 0.05 |
| 66458623 | 0.05 |
| 66458624 | 0.05 |
| 66458625 | 0.05 |
| 66458626 | 0.05 |
| 66458627 | 0.05 |
| 66458628 | 0.05 |
| 66458629 | 0.05 |
| 66458630 | 0.05 |
| 66458631 | 0.05 |
| 66458632 | 0.05 |
| 66458633 | 0.05 |
| 66458634 | 0.05 |
| 66458635 | 0.05 |
| 66458636 | 0.05 |
| 66458637 | 0.05 |
| 66458638 | 0.05 |
| 66458639 | 0.05 |
| 66458640 | 0.05 |
| 66458641 | 0.05 |
| 66458642 | 0.05 |
| 66458643 | 0.05 |
| 66458644 | 0.05 |
| 66458645 | 0.05 |
| 66458646 | 0.05 |
| 66458647 | 0.05 |
| 66458648 | 0.05 |
| 66458649 | 0.05 |
| 66458650 | 0.05 |
| 66458651 | 0.05 |
| 66458652 | 0.05 |
| 66458653 | 0.05 |
| | |

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:

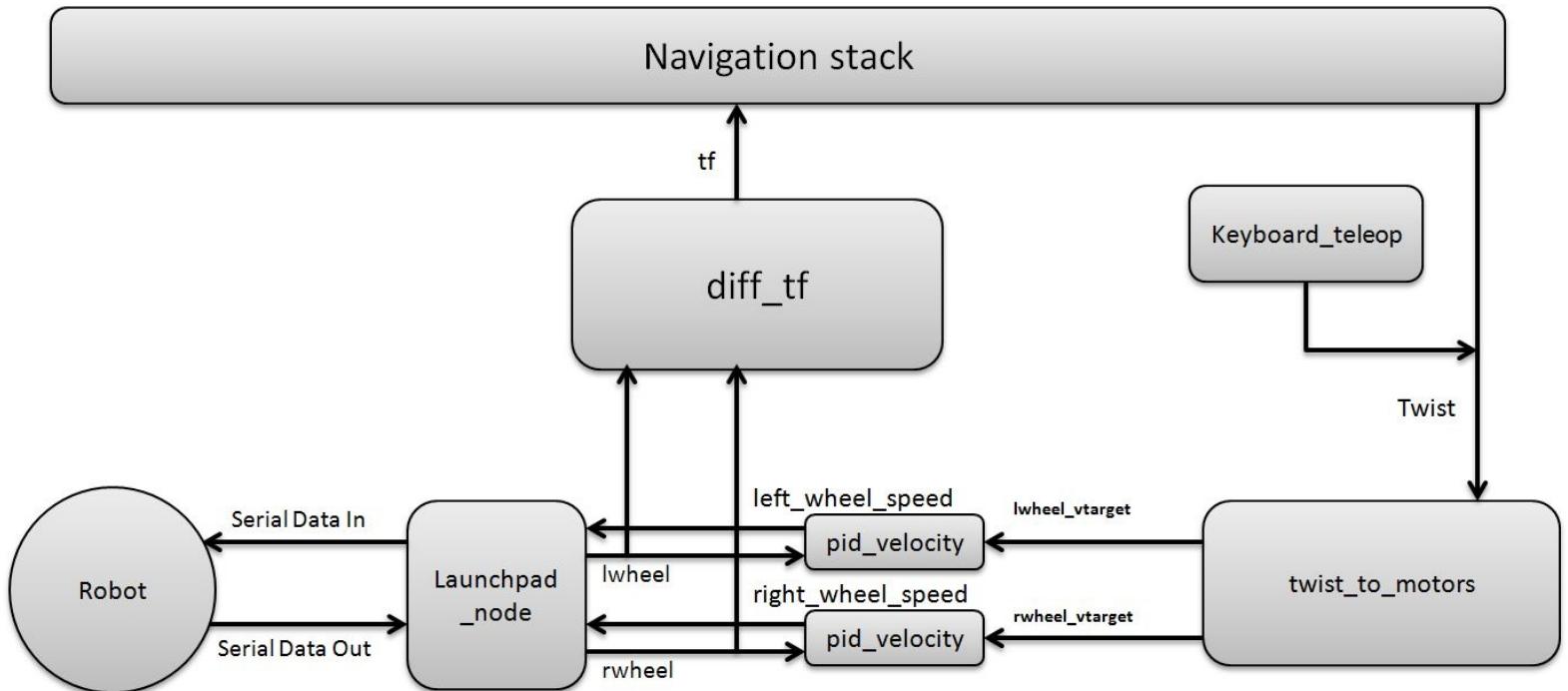


Figure 5: Interconnection of each nodes in Chefbot

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:

```
&lt;launch>
  &lt;arg name="simulation" default="$(optenv TURTLEBOT_SIMULATION false)" />
  &lt;param name="/use_sim_time" value="$(arg simulation)" />

  &lt;!-- URDF robot model -->
  &lt;arg name="urdf_file" default="$(find xacro)/xacro.py '$(find chefbot_description)/urdf/chefbot_base.xacro'" />
  &lt;param name="robot_description" command="$(arg urdf_file)" />

  &lt;!-- important generally, but specifically utilised by the current app manager -->
  &lt;param name="robot/name" value="$(optenv ROBOT turtlebot)" />
  &lt;param name="robot/type" value="turtlebot" />

  &lt;!-- Starting robot state publisher -->
  &lt;node pkg="robot_state_publisher" type="robot_state_publisher" name="robot_state_publisher">
    &lt;param name="publish_frequency" type="double" value="5.0" />
  &lt;/node>

  &lt;!-- Robot parameters -->
  &lt;rosparam param="base_width">0.3</rosparam>
  &lt;rosparam param="ticks_meter">14865</rosparam>

  &lt;!-- Starting launchpad_node -->
  &lt;node name="launchpad_node" pkg="chefbot Bringup" type="launchpad_node.py">
    &lt;rosparam file="$(find chefbot Bringup)/param/serial.yaml" command="load" />
  &lt;/node>

  &lt;!-- PID node for left motor , setting PID parameters -->
  &lt;node name="lpid_velocity" pkg="chefbot Bringup" type="pid_velocity.py" output="screen">
    &lt;remap from="wheel" to="lwheel"/>
    &lt;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:

```
$ roslaunch chefbot Bringup view_robot.launch
```

We will see the robot model as shown in this next screenshot:

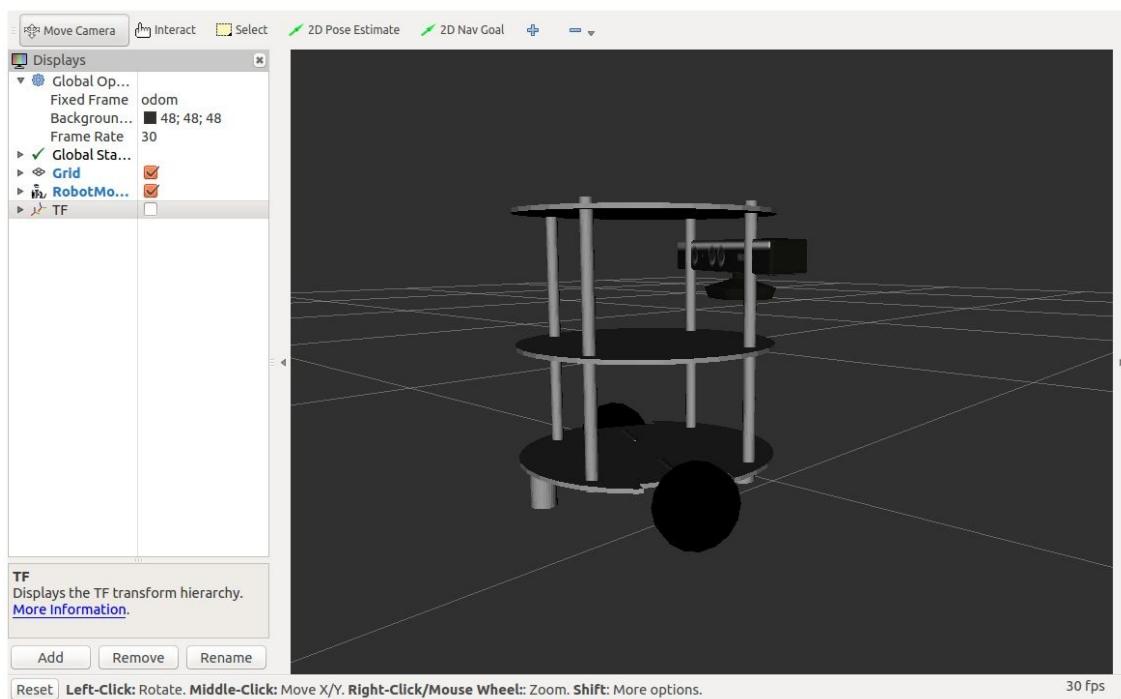


Figure 7: Visualization of robot model using real robot values.

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:

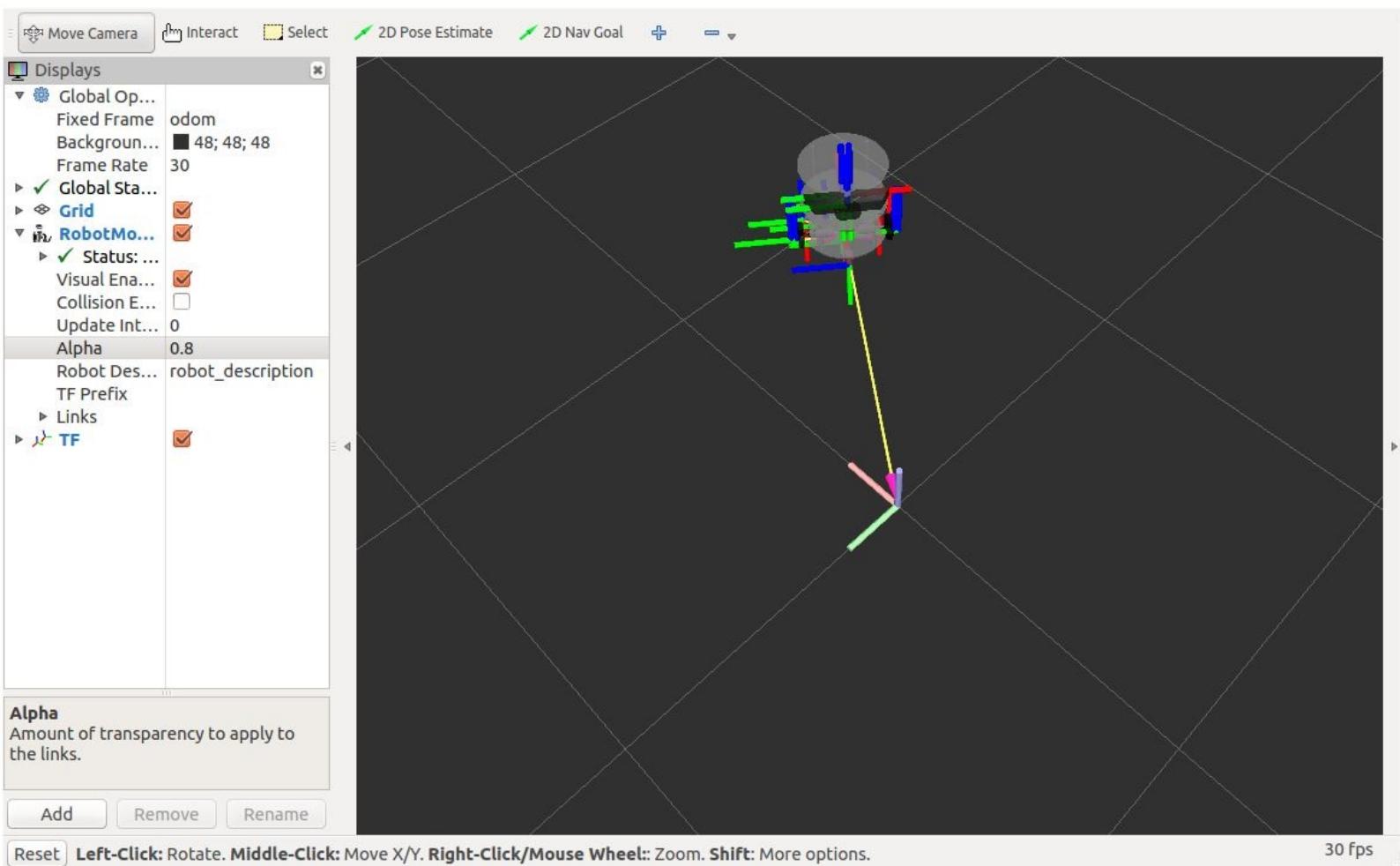


Figure 8: Visualizing robot odometry

The graph of the connection between each node is given next. We can view it using the `rqt_graph` tool.

```
| $ rqt_graph
```

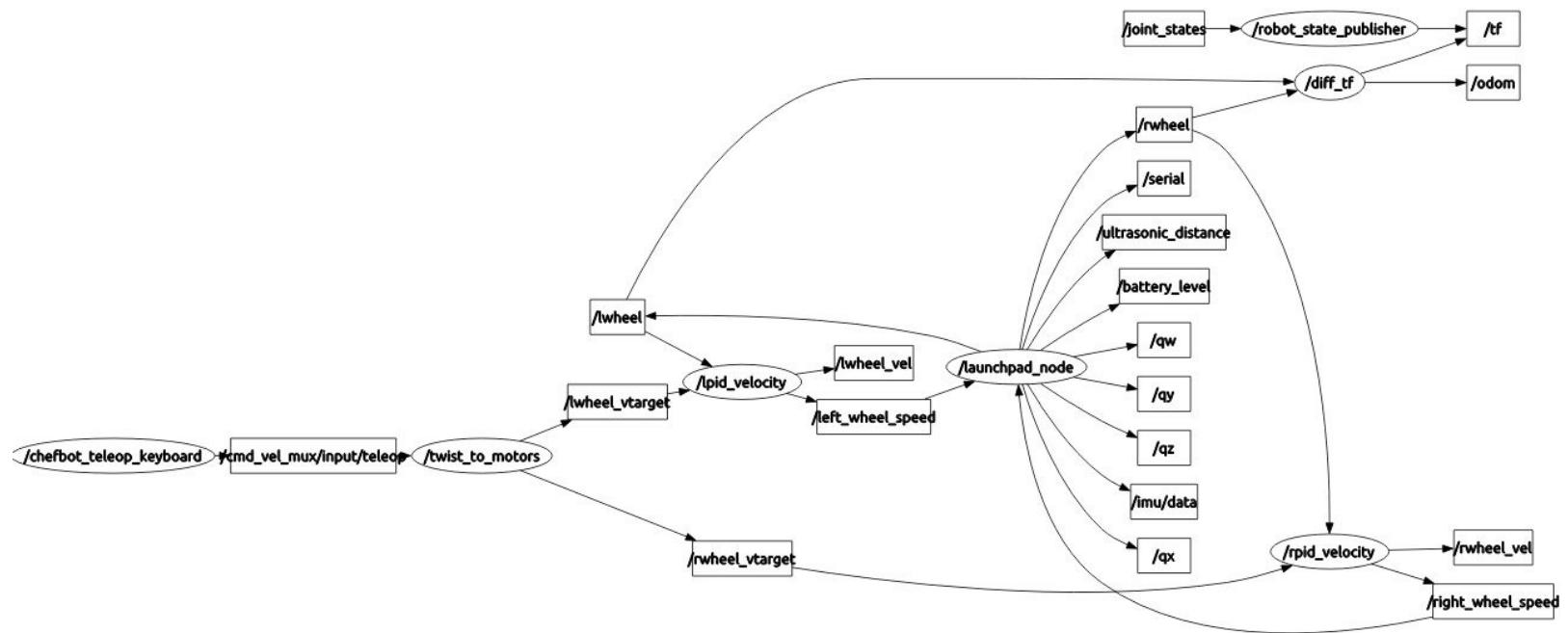


Figure 9: Interconnection of nodes in Chefbot

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.

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:

```
&lt;launch>
  &lt;arg name="scan_topic" default="scan" />

  &lt;!-- Starting gmapping node -->
  &lt;node pkg="gmapping" type="slam_gmapping" name="slam_gmapping" output="screen">

  &lt;!-- Frame of mobile base -->
  &lt;param name="base_frame" value="base_footprint"/>
  &lt;param name="odom_frame" value="odom"/>
  &lt;!-- The interval of map updation, reducing this value will speed of map generation but increase computation load -->
  &lt;param name="map_update_interval" value="5.0"/>
  &lt;!-- Maximum usable range of laser/kinect -->
  &lt;param name="maxUrange" value="6.0"/>
  &lt;!-- Maximum range of sensor, max range should be > maxUrange -->
  &lt;param name="maxRange" value="8.0"/>
  &lt;param name="sigma" value="0.05"/>
  &lt;param name="kernelSize" value="1"/>
&lt;/node>
&lt;/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.

```
&lt;launch>
  &lt;!-- Launches 3D sensor nodes -->
  &lt;include file="$(find chefbot_bringup)/launch/3dsensor.launch">
    &lt;arg name="rgb_processing" value="false" />
    &lt;arg name="depth_registration" value="false" />
    &lt;arg name="depth_processing" value="false" />
    &lt;arg name="scan_topic" value="/scan" />
  &lt;/include>
```

```
&lt;!-- Start gmapping nodes and its configurations -->
&lt;include file="$(find chefbot Bringup)/launch/includes/gmapping.launch.xml"/>

&lt;!-- Start move_base node and its configuration -->
&lt;include file="$(find chefbot Bringup)/launch/includes/move_base.launch.xml"/>
&lt;/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`:

```
&lt;launch>
  &lt;arg name="odom_topic" default="odom" />
  &lt;!-- Starting move_base node -->
  &lt;node pkg="move_base" type="move_base" respawn="false" name="move_base" output="screen">

  &lt;!-- common parameters of global costmap -->
  &lt;rosparam file="$(find chefbot_bringup)/param/costmap_common_params.yaml" command="load" ns="global_costmap"/>

  &lt;!-- common parameters of local costmap -->
  &lt;rosparam file="$(find chefbot_bringup)/param/costmap_common_params.yaml" command="load" ns="local_costmap"/>

  &lt;!-- local cost map parameters -->
  &lt;rosparam file="$(find chefbot_bringup)/param/local_costmap_params.yaml" command="load" />

  &lt;!-- global cost map parameters -->
  &lt;rosparam file="$(find chefbot_bringup)/param/global_costmap_params.yaml" command="load" />

  &lt;!-- base local planner parameters -->
  &lt;rosparam file="$(find chefbot_bringup)/param/base_local_planner_params.yaml" command="load" />

  &lt;!-- dwa local planner parameters -->
  &lt;rosparam file="$(find chefbot_bringup)/param/dwa_local_planner_params.yaml" command="load" />

  &lt;!-- move_base node parameters -->
  &lt;rosparam file="$(find chefbot_bringup)/param/move_base_params.yaml" command="load" />

    &lt;remap from="cmd_vel" to="/cmd_vel_mux/input/navi"/>
    &lt;remap from="odom" to="$(arg odom_topic)"/>
  &lt;/node>
&lt;/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.,-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:

```
| $ roslaunch 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:

```
| $ roslaunch chefbot_bringup keyboard_teleop.launch
```

We can see the map building in RViz, which can be invoked using the following command:

```
| $ roslaunch 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:

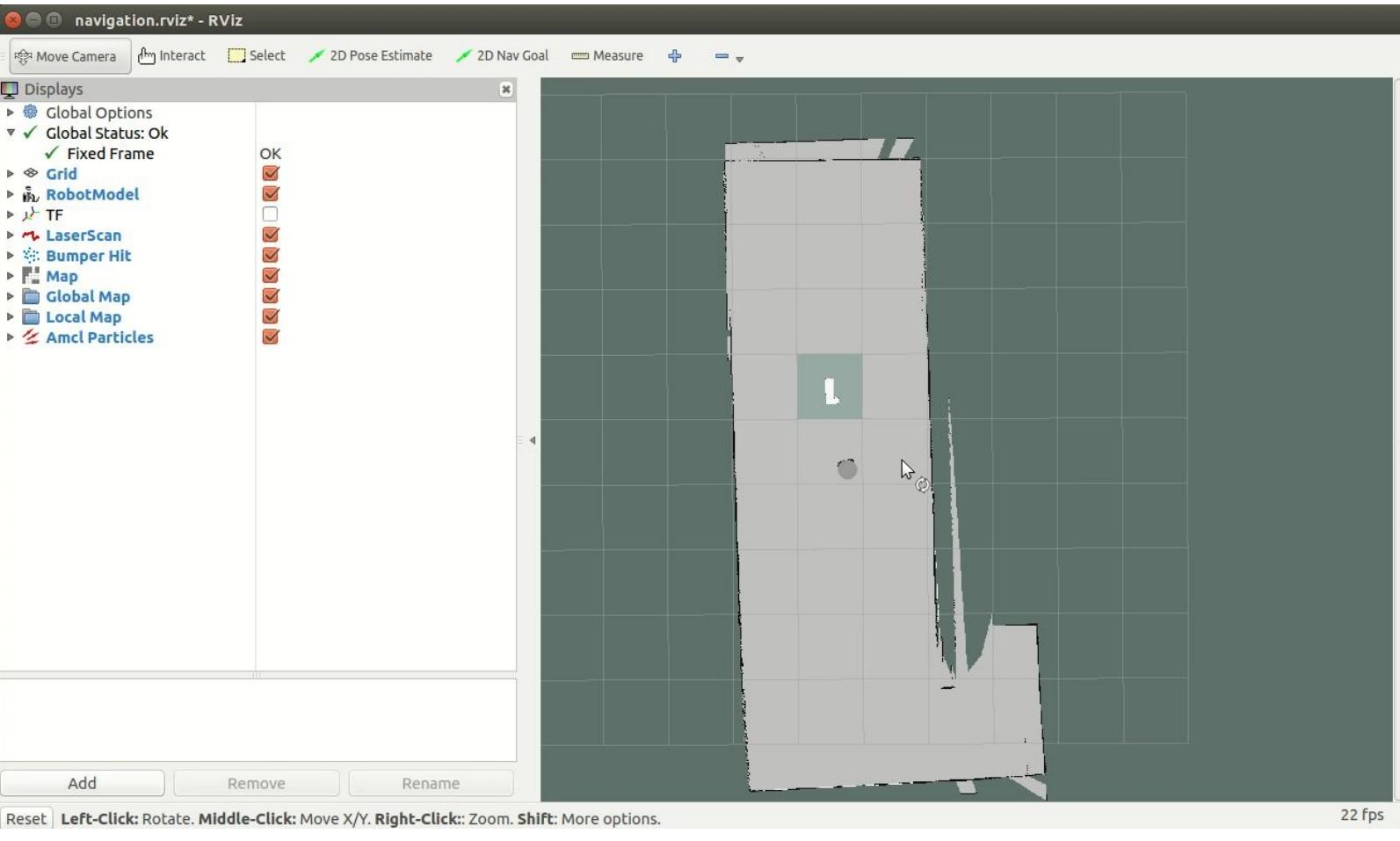


Figure 10: Creating a map using gmapping is shown in RViz

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:~$ rosrun 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. 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`:

```
&lt;launch>
  &lt;rosparam command="delete" ns="move_base" />
  &lt;include file="$(find chefbot_bringup)/launch/3dsensor.launch">
    &lt;arg name="rgb_processing" value="false" />
    &lt;arg name="depth_registration" value="false" />
    &lt;arg name="depth_processing" value="false" />

    &lt;!-- We must specify an absolute topic name because if not it will be prefixed by "$(arg camera)". -->
    &lt;arg name="scan_topic" value="/scan" />
  &lt;/include>

  &lt;!-- Map server -->
  &lt;arg name="map_file" default="$(find turtlebot_navigation)/maps/willow-2010-02-18-0.10.yaml"/>
  &lt;node name="map_server" pkg="map_server" type="map_server" args="$(arg map_file)" />

  &lt;arg name="initial_pose_x" default="0.0"/> &lt;!-- Use 17.0 for willow's map in simulation -->
  &lt;arg name="initial_pose_y" default="0.0"/> &lt;!-- Use 17.0 for willow's map in simulation -->
  &lt;arg name="initial_pose_a" default="0.0"/>

  &lt;include file="$(find chefbot_bringup)/launch/includes/amcl.launch.xml">
    &lt;arg name="initial_pose_x" value="$(arg initial_pose_x)"/>
    &lt;arg name="initial_pose_y" value="$(arg initial_pose_y)"/>
    &lt;arg name="initial_pose_a" value="$(arg initial_pose_a)"/>
  &lt;/include>

  &lt;include file="$(find chefbot_bringup)/launch/includes/move_base.launch.xml"/>
&lt;/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:

```
&lt;launch>
  &lt;arg name="use_map_topic"  default="false"/>
  &lt;arg name="scan_topic"    default="scan"/>
  &lt;arg name="initial_pose_x" default="0.0"/>
  &lt;arg name="initial_pose_y" default="0.0"/>
  &lt;arg name="initial_pose_a" default="0.0"/>

  &lt;node pkg="amcl" type="amcl" name="amcl">
    &lt;param name="use_map_topic"           value="$(arg use_map_topic)"/>
    .....
    .....

    &lt;!-- Increase tolerance because the computer can get quite busy -->
    &lt;param name="transform_tolerance"      value="1.0"/>
    &lt;param name="recovery_alpha_slow"     value="0.0"/>
    &lt;param name="recovery_alpha_fast"     value="0.0"/>
    &lt;param name="initial_pose_x"          value="$(arg initial_pose_x)"/>
    &lt;param name="initial_pose_y"          value="$(arg initial_pose_y)"/>
    &lt;param name="initial_pose_a"          value="$(arg initial_pose_a)"/>
    &lt;remap from="scan"                  to="$(arg scan_topic)"/>
  &lt;/node>
&lt;/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:

```
| $ roslaunch chefbot_bringup robot_standalone.launch
```

Run the `amcl` launch file using the following command:

```
| $ roslaunch 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:

```
| $ roslaunch chefbot_bringup view_navigation.launch
```

The following is the screenshot of RViz:

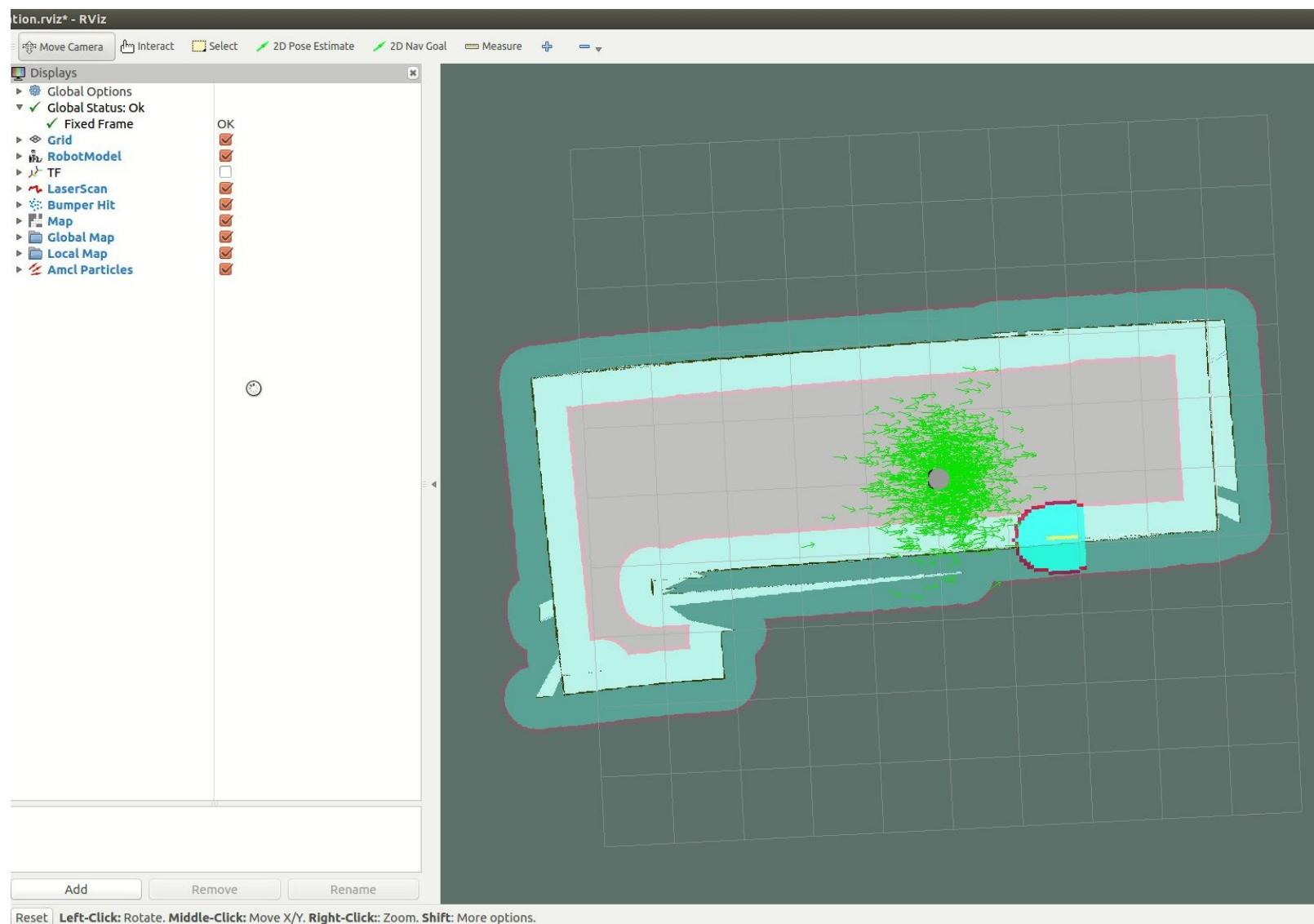


Figure 12: Robot autonomous navigation using AMCL

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:

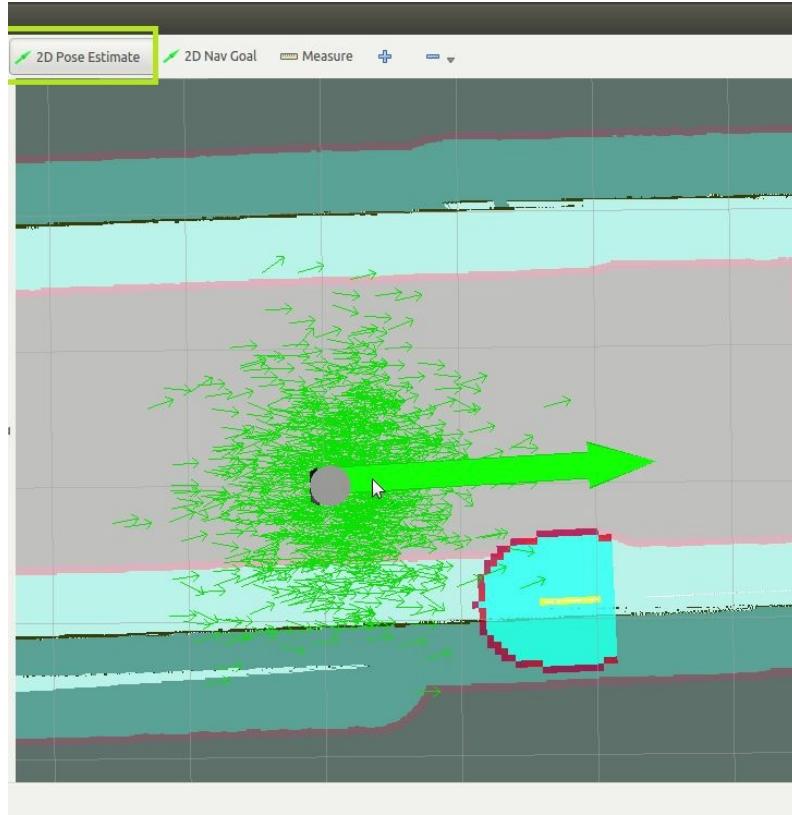


Figure 13: RViz 2D Pose Estimate button

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 `amc1`. 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`

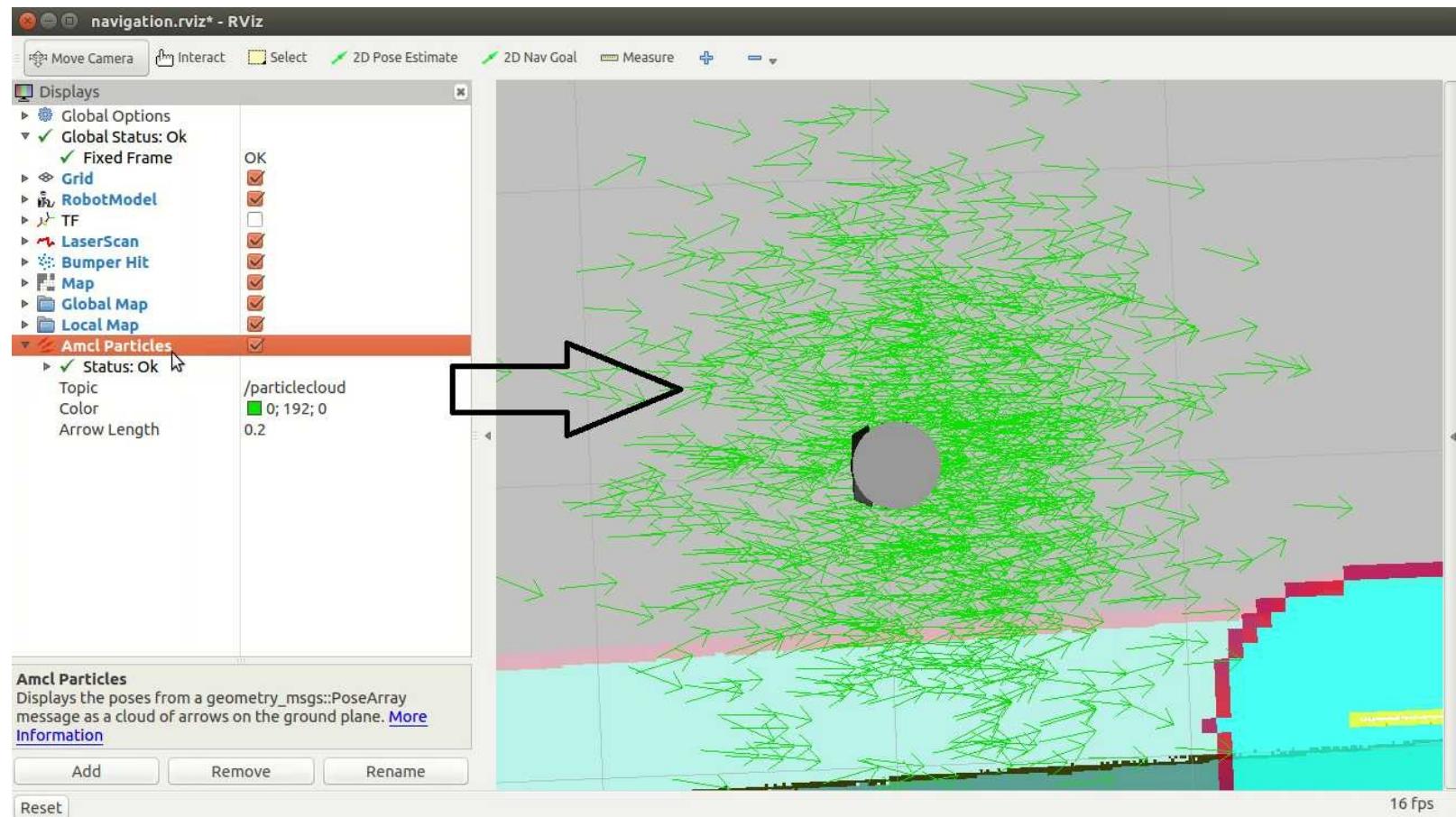


Figure 14: Visualizing AMCL particles

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`

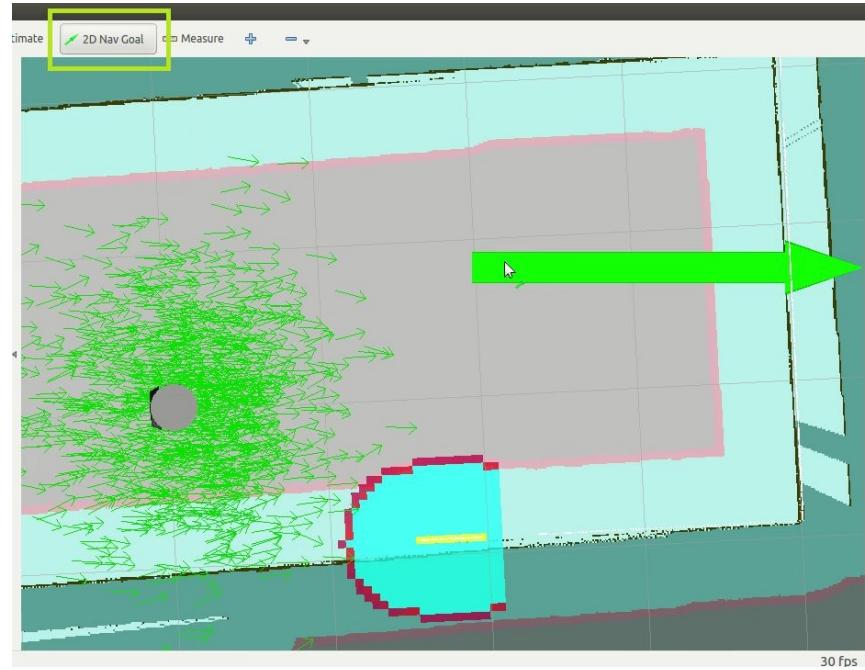


Figure 15 []:Setting robot goal position in RViz using 2D Nav Goal

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:

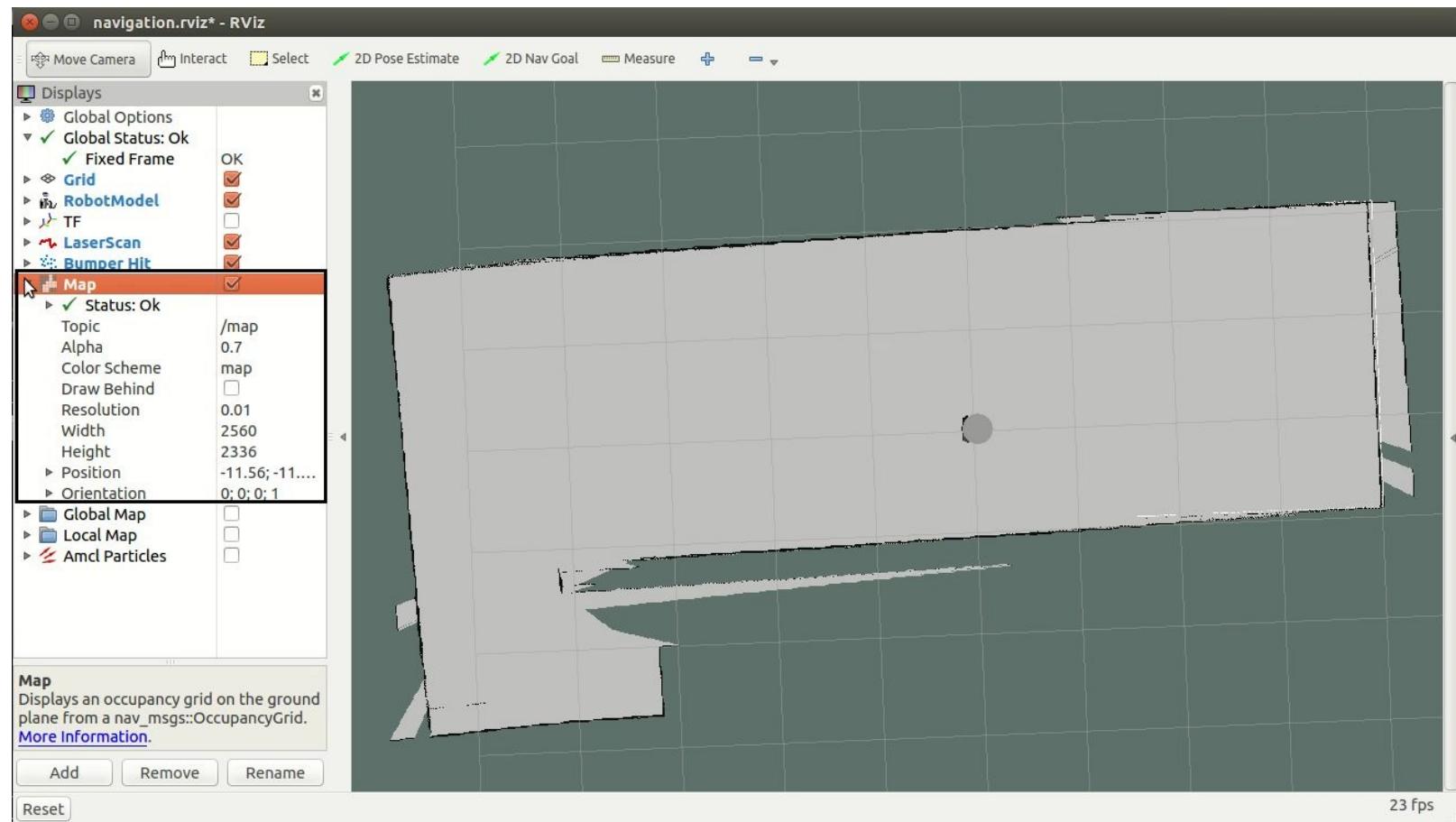


Figure 16: Visualizing static map in RViz

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`

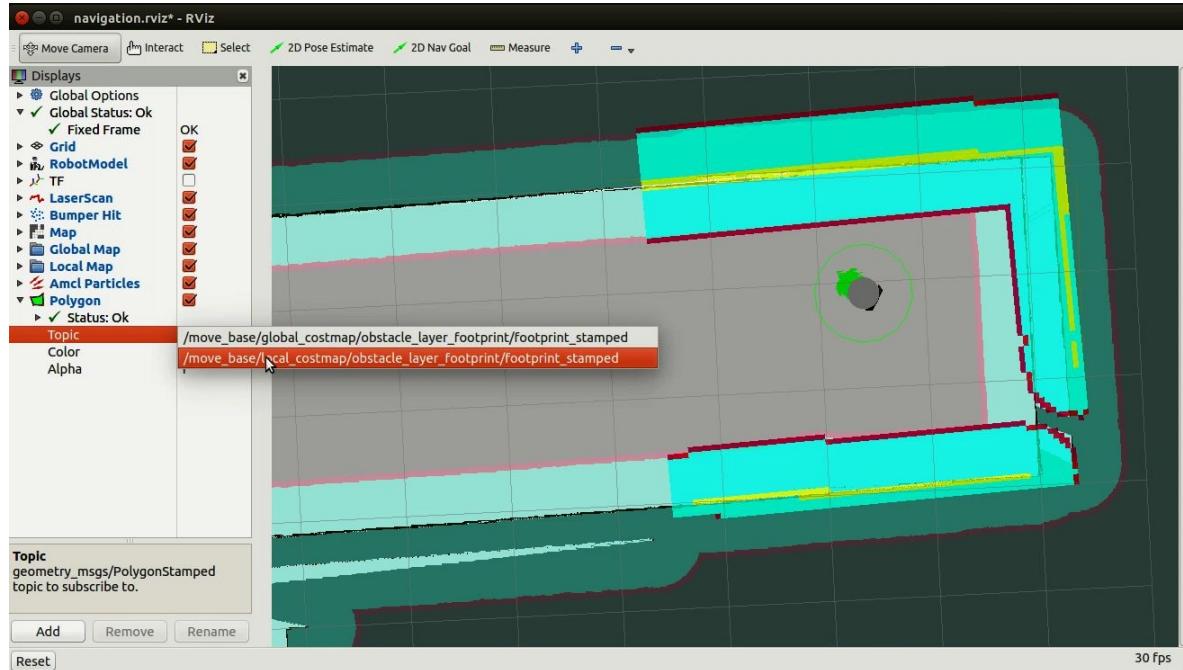


Figure 17: global and local robot footprint in RViz

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`

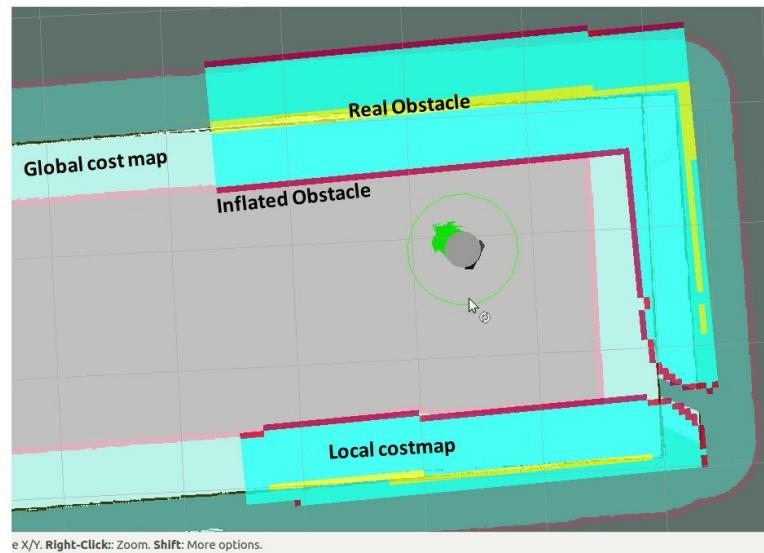


Figure 18 : Visualizing global and local map, and real and inflated obstacle in RViz

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`

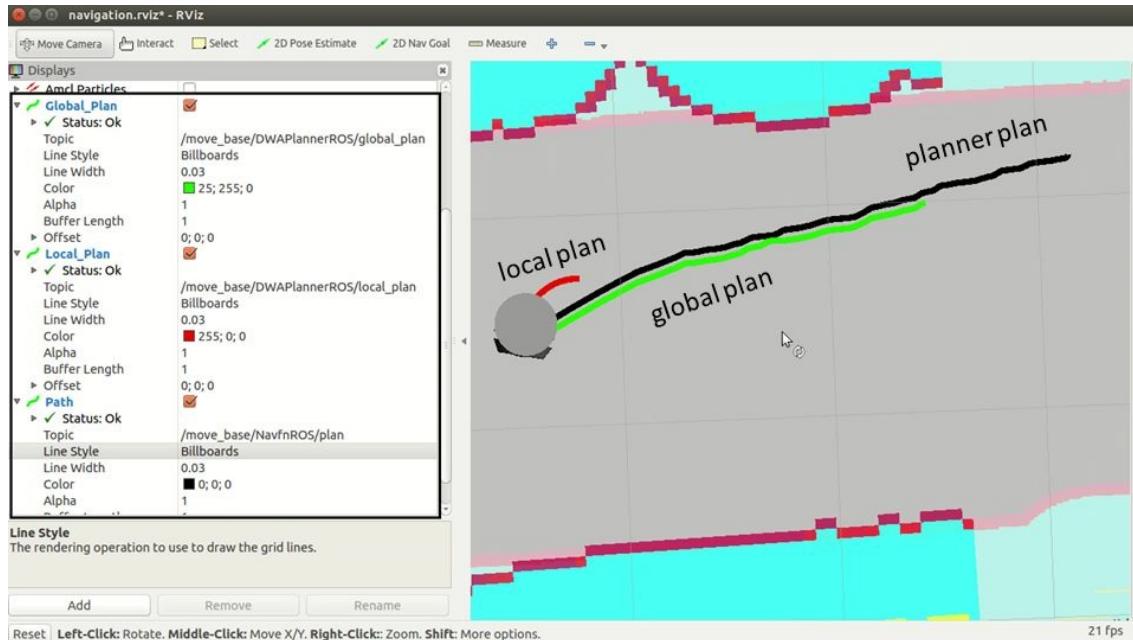


Figure 19: Visualizing global, local, and planner plan in RViz

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

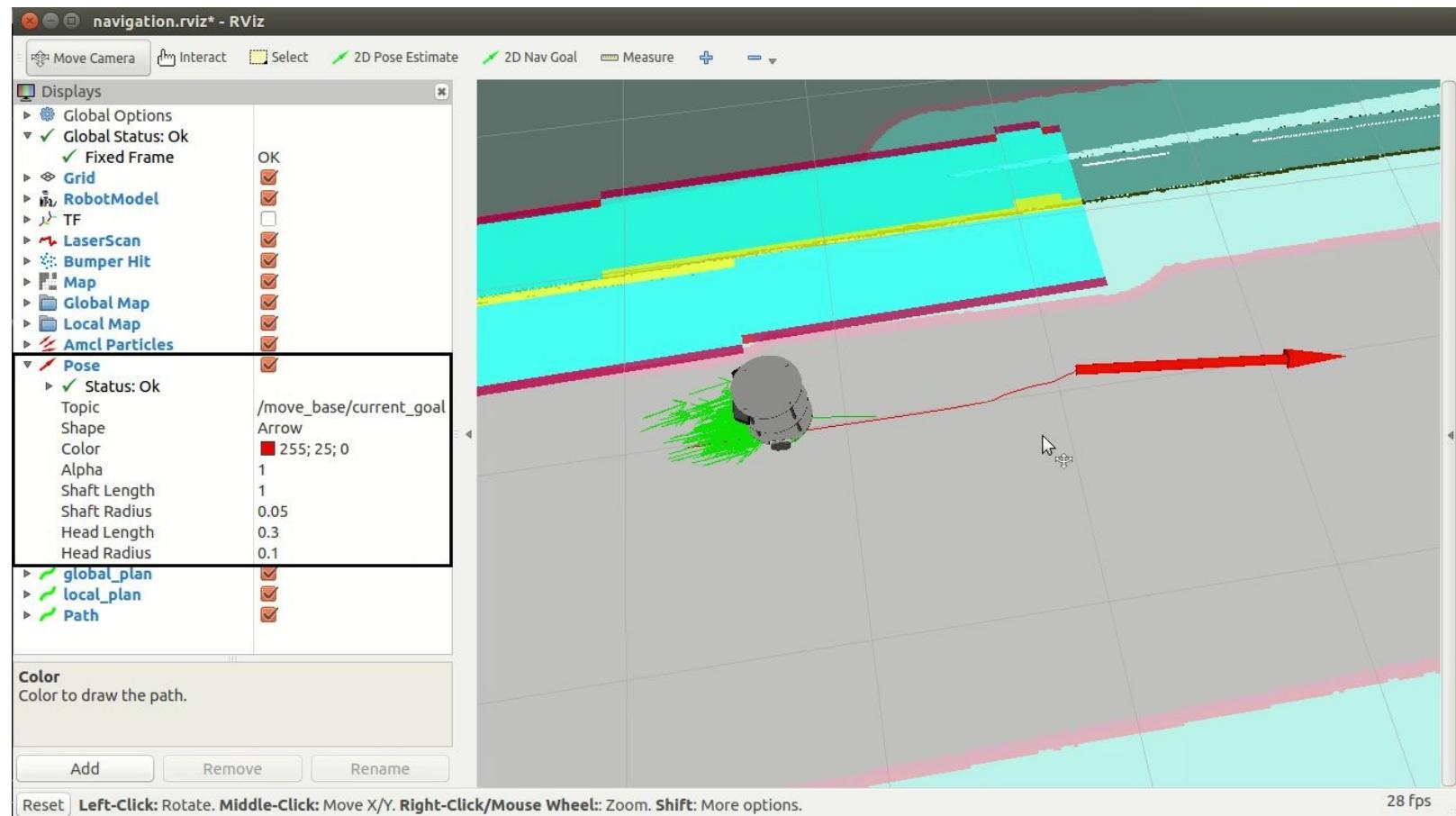


Figure 20: Visualizing robot goal position

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.

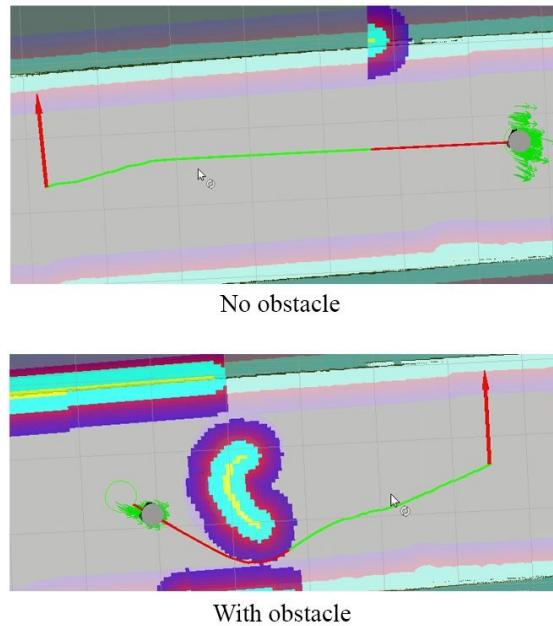


Figure 21: Visualizing obstacle avoidance capabilities in RViz

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.

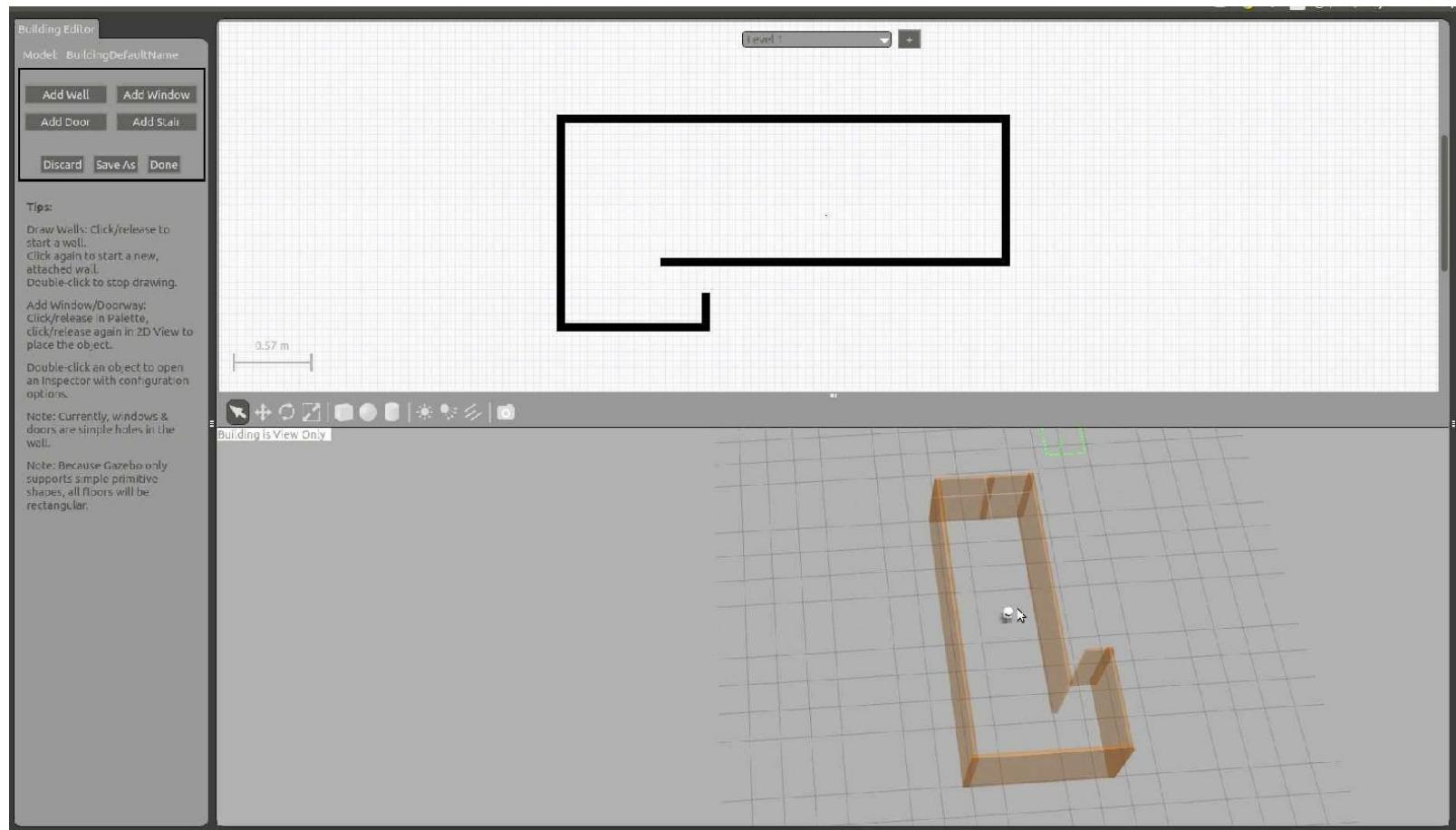


Figure 22: Building walls in Gazebo

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:

```
&lt;?xml version="1.0"?>

&lt;model>
&lt;!-- Name of model which is displaying in Gazebo -->
  &lt;name>Test Room&lt;/name>
  &lt;version>1.0&lt;/version>
&lt;!-- Model file name -->
  &lt;sdf version="1.2">final_room.sdf&lt;/sdf>

  &lt;author>
    &lt;name>Lentin Joseph&lt;/name>
    &lt;email>qboticslabs@gmail.com&lt;/email>
  &lt;/author>

  &lt;description>
    A test room for performing SLAM
  &lt;/description>
&lt;/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:

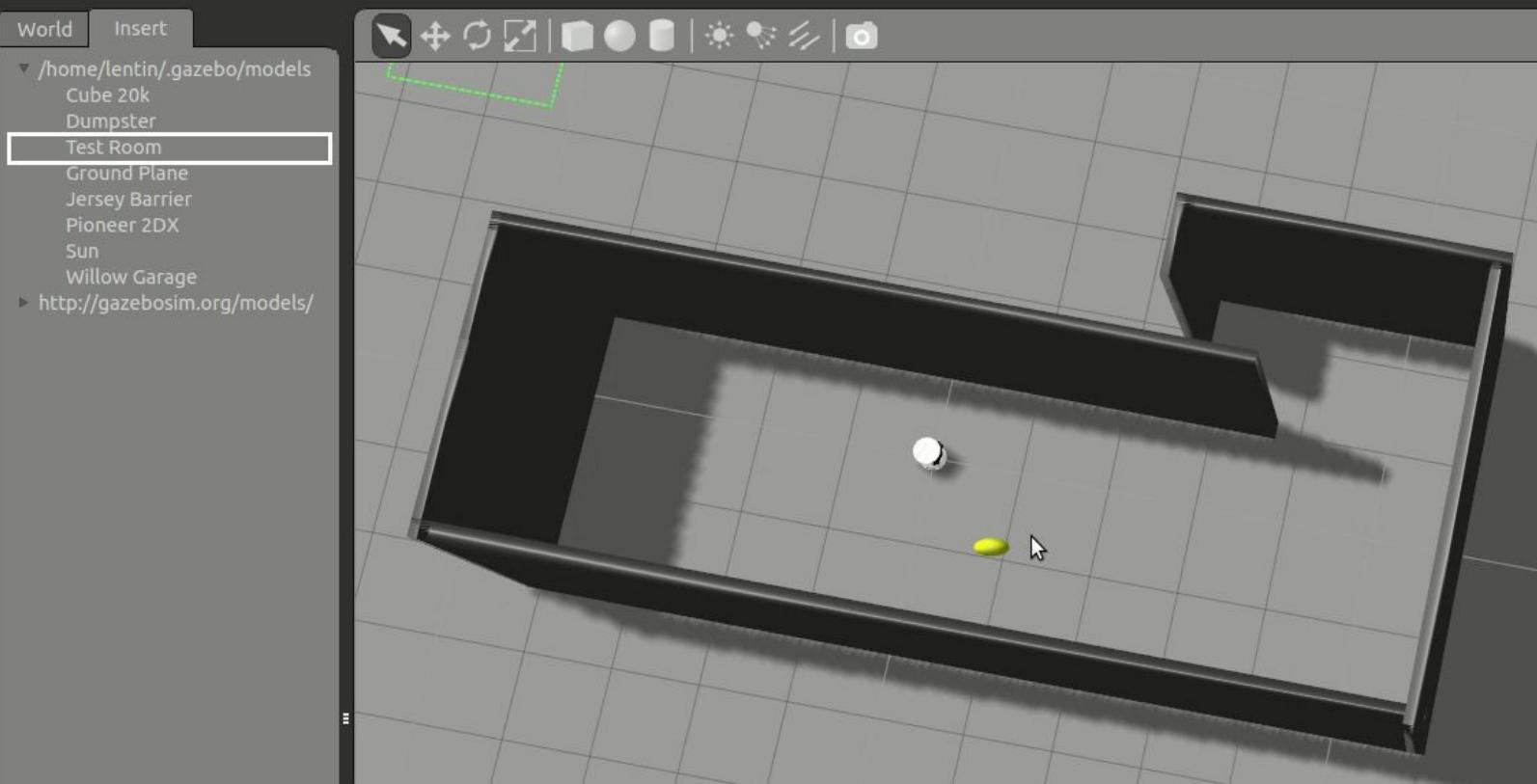


Figure 23: Inserting the walls in Chefbot simulation

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:

```
&lt;include file="$(find gazebo_ros)/launch/empty_world.launch">
  &lt;arg name="use_sim_time" value="true"/>
  &lt;arg name="debug" value="false"/>

  &lt;!-- Adding world test_room.sdf as argument -->
  &lt;arg name="world_name" value="$(find chefbot_gazebo)/worlds/test_room.sdf"/>
&lt;/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.

```
| $ rosrun 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_brinup 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 <iostream>
#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.

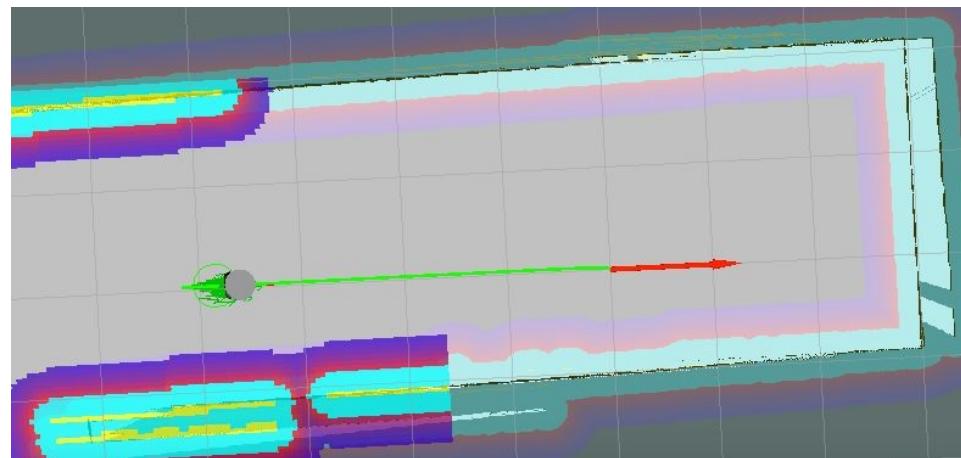


Figure 24: Sending a goal to move_base node from C++ APIs

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`.

Exploring the Advanced Capabilities of ROS-MoveIt!

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 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 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 rospp 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:

```
| $ roslaunch 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:

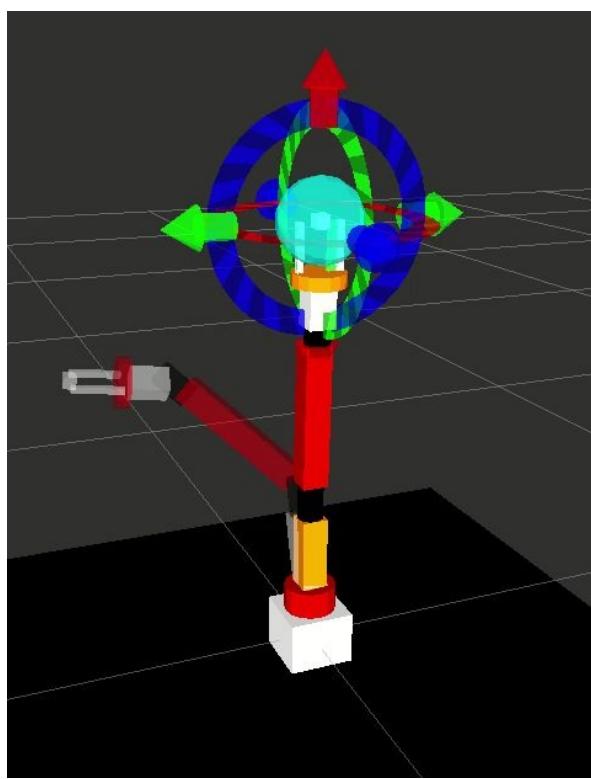


Figure 1: Random motion planning using move_group APIs

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`:

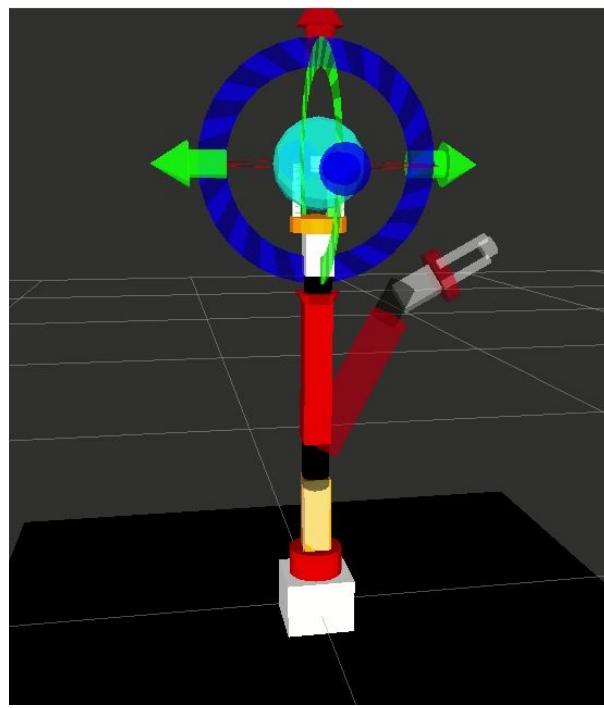


Figure 2: Custom motion planning using MoveIt! C++ APIs

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:

In the node `add_collision_objt.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);
```

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";
```

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;
```

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;
```

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;
```

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);
```

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_objct src/add_collision_objct.cpp)  
| add_dependencies(add_collision_objct  
|   seven_dof_arm_test_generate_messages_cpp)  
| target_link_libraries(add_collision_objct  
| ${catkin_LIBRARIES} )
```

Let's see how this node works in RViz with MoveIt! motion planning Plugin:

We will start `demo.launch` inside the `seven_dof_arm_config` package for testing this node:

```
| $ roslaunch seven_dof_arm_config demo.launch
```

Next, add the following collision object:

```
| $ rosrun seven_dof_arm_test add_collision_objct
```

When we run the `add_collision_objct` 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:

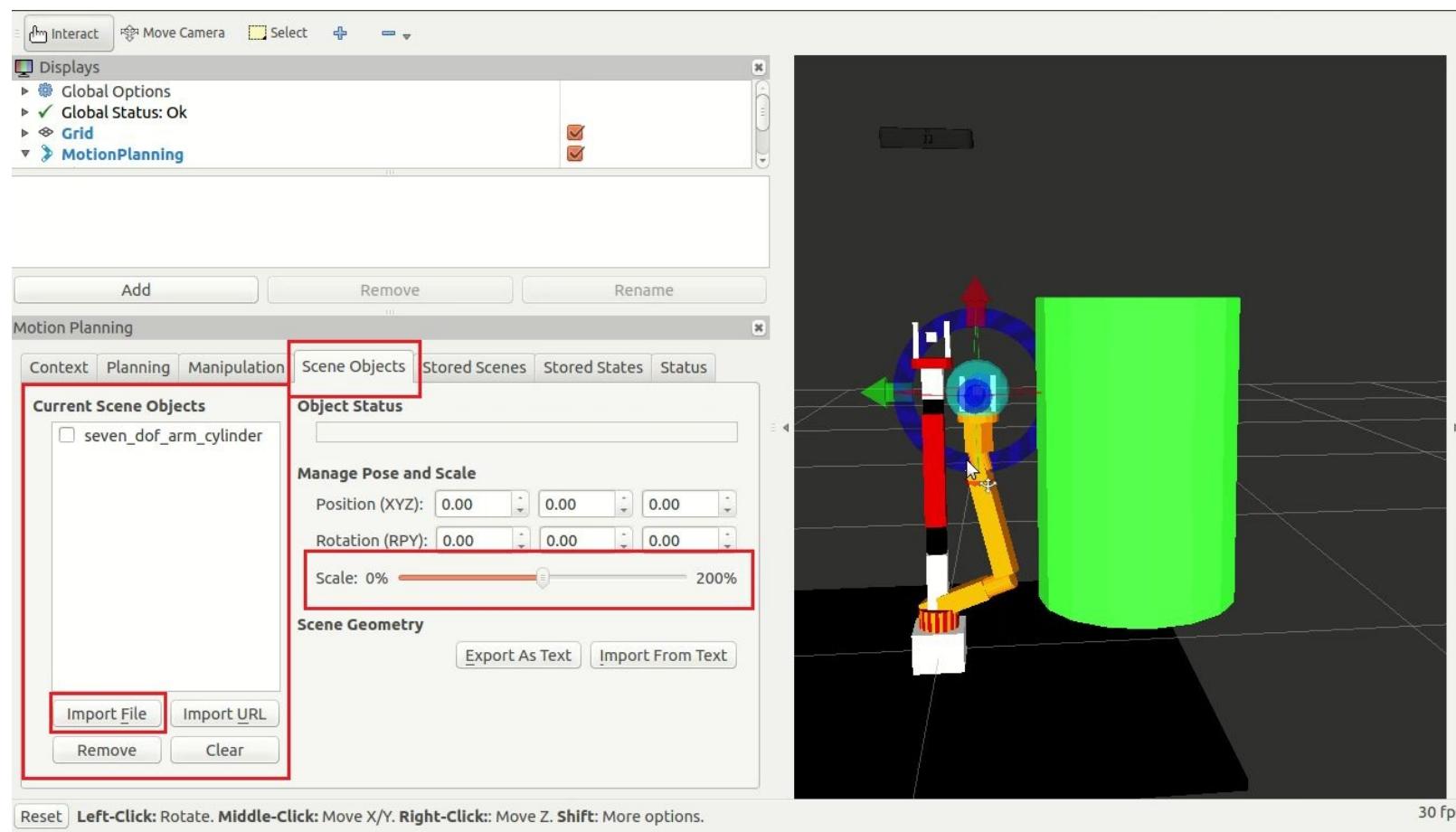


Figure 3 : Adding collision objects to RViz using MoveIt!! C++ APIs

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:

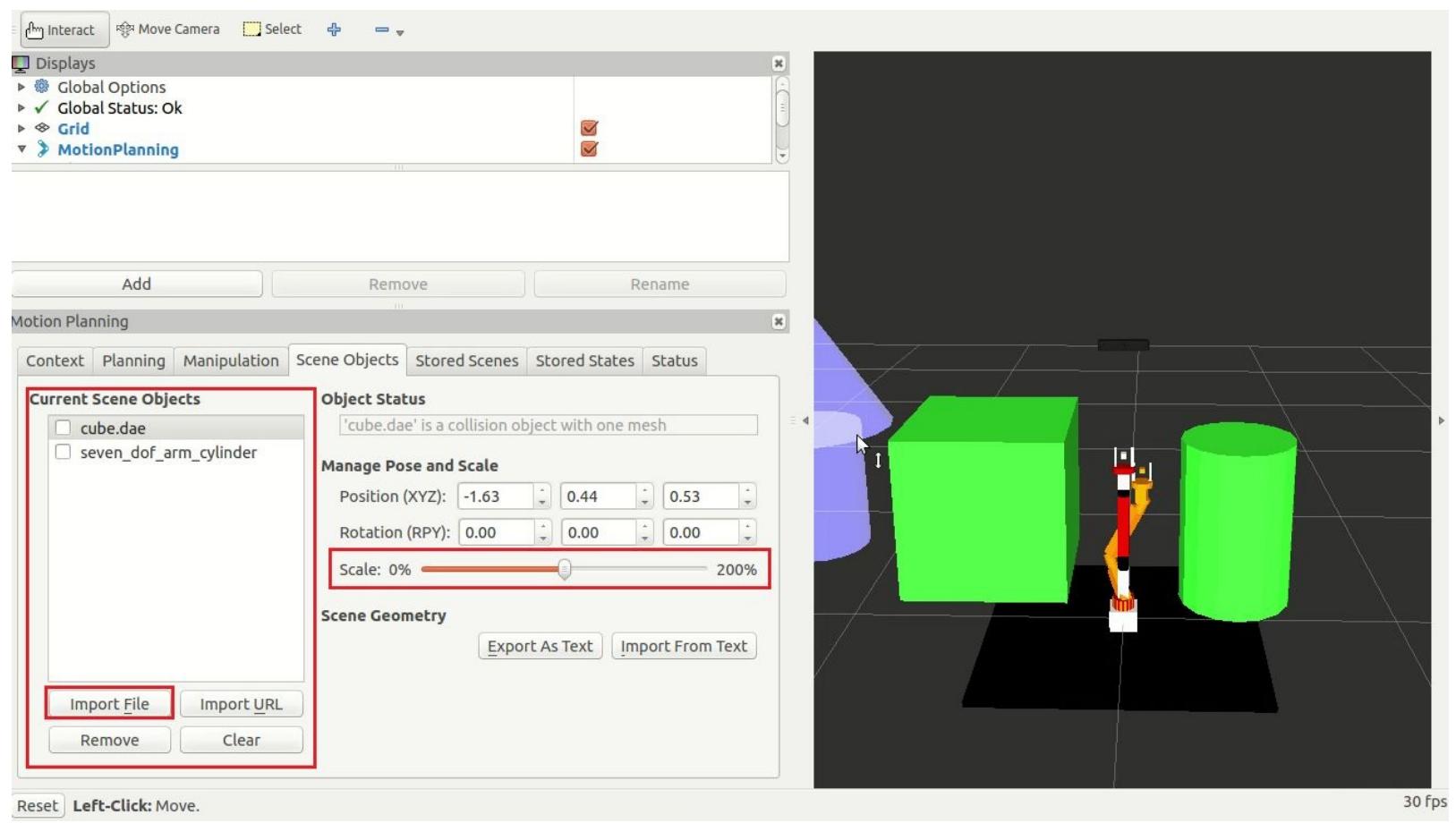


Figure 4: Adding collision objects by importing meshes

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:

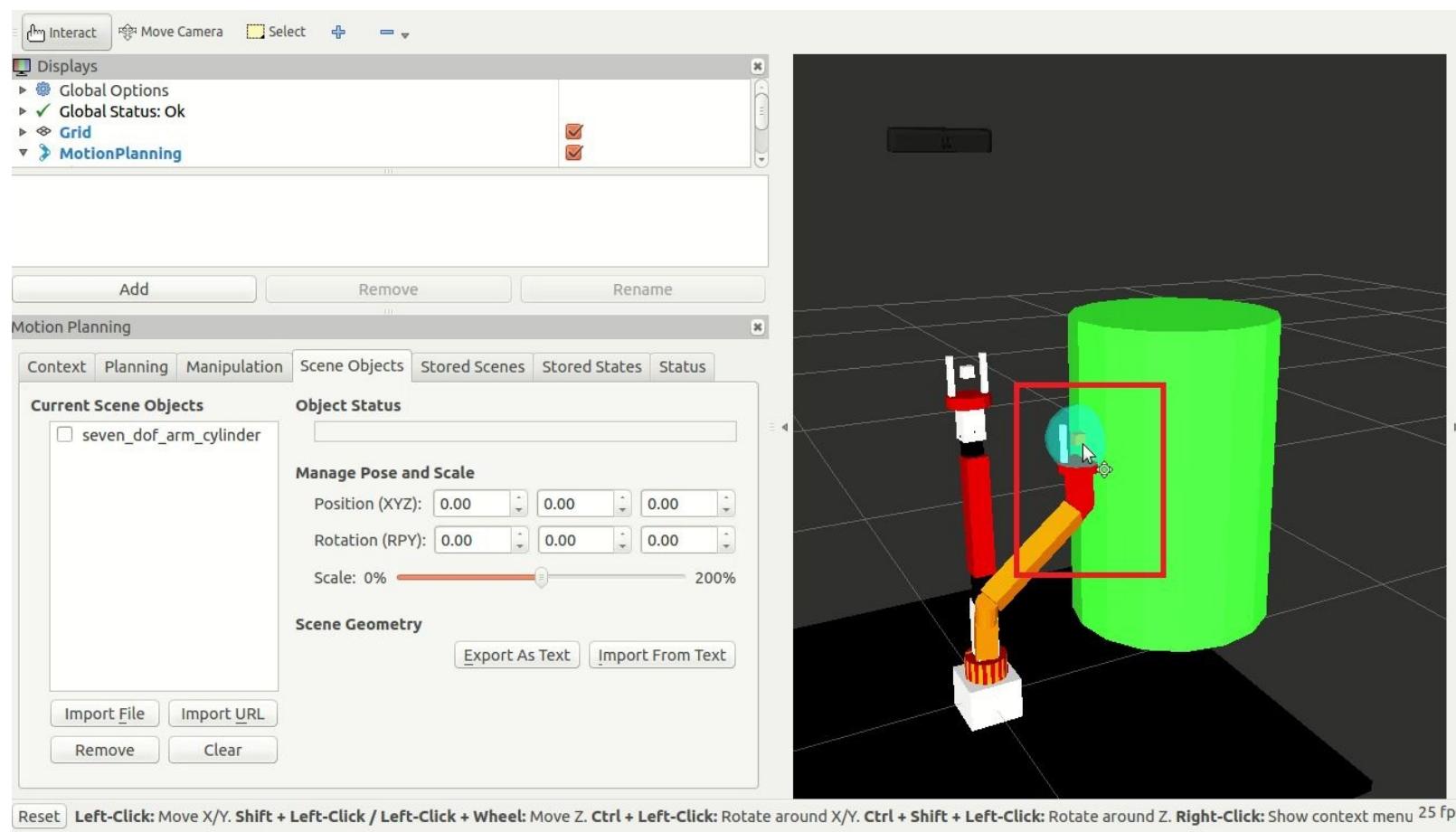


Figure 5: Visualizing collided link.

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: "<&lt;  
(collision_result.collision ? "in" : "not in")  
&lt;&lt; " 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): "<&lt;  
(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: "<&lt;  
(collision_result.collision ? "in" : "not in"))
```

```
| &lt; " 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:

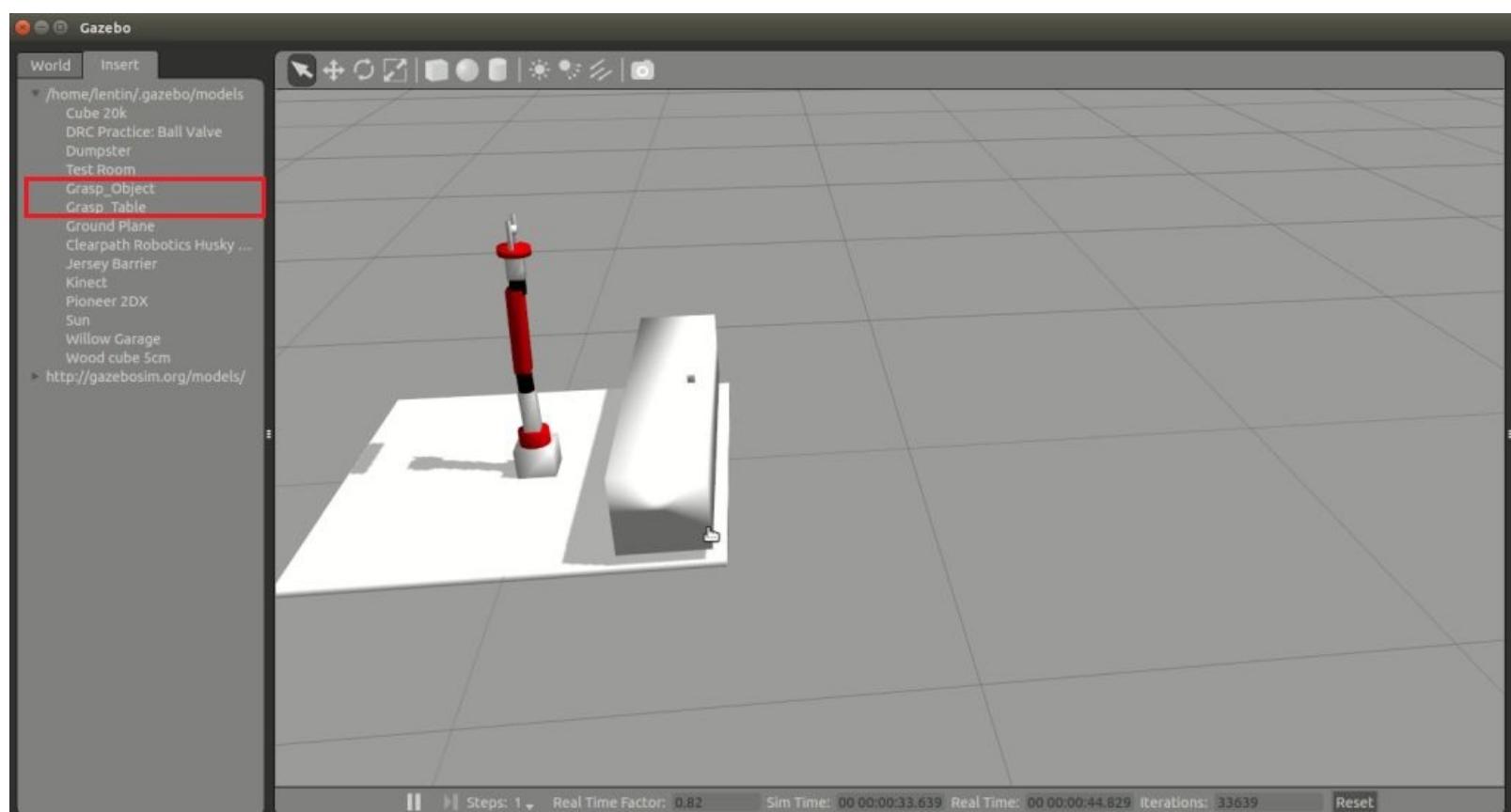


Figure 7: Robot arm with grasp table and object in Gazebo

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:

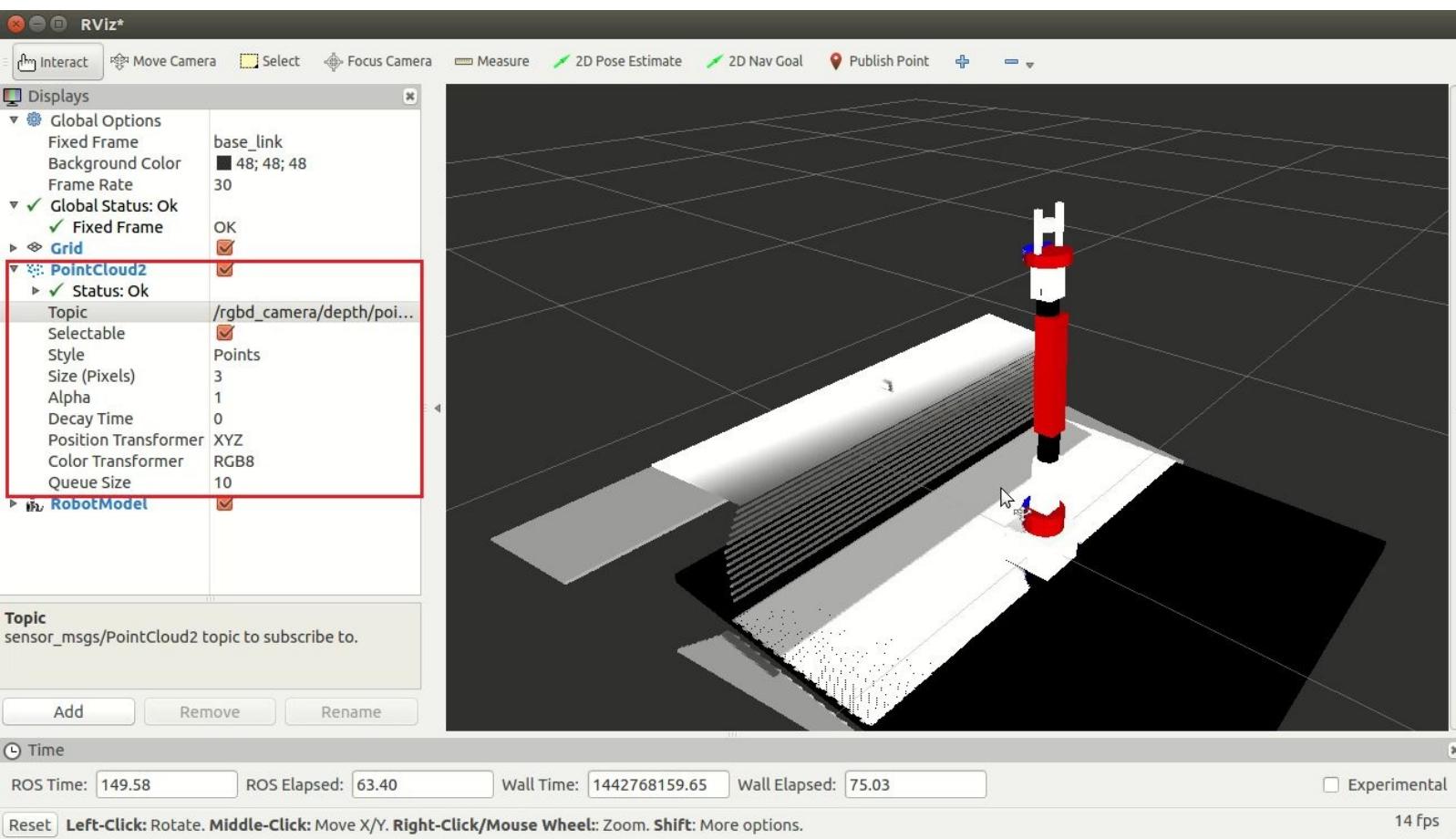


Figure 9: Visualizing point cloud data in RViz

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 **Occupancy 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:

```
&lt;launch>
  &lt;rosparam command="load" file="$(find seven_dof_arm_config)/config/sensors_rgbd.yaml" />
&lt;/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:

```
&lt;launch>
  &lt;!-- This file makes it easy to include the settings for sensor managers -->

  &lt;!-- Params for the octomap monitor -->
  &lt;!--   &lt;param name="octomap_frame" type="string" value="some frame in which the robot moves" /> -->
  &lt;param name="octomap_resolution" type="double" value="0.015" />
  &lt;param name="max_range" type="double" value="5.0" />

  &lt;!-- Load the robot specific sensor manager; this sets the moveit_sensor_manager ROS parameter -->

  &lt;arg name="moveit_sensor_manager" default="seven_dof_arm" />
  &lt;include file="$(find seven_dof_arm_config)/launch/$(arg
moveit_sensor_manager)_moveit_sensor_manager.launch.xml" />

&lt;/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:

```
&lt;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.

```
&lt;param name="octomap_resolution" type="double" value="0.015" />
&lt;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_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:

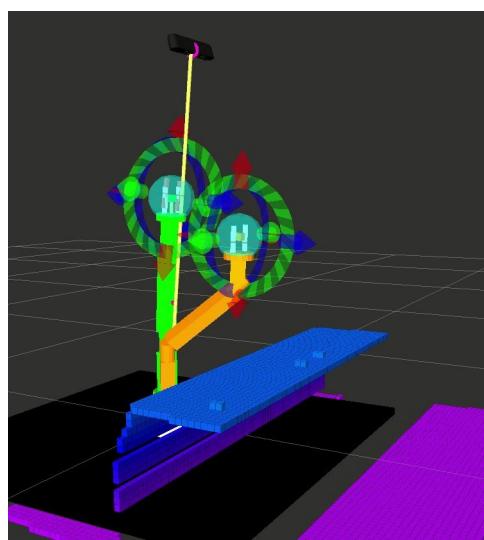


Figure 10 : Visualizing octomap in RViz

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:

```
&lt;launch>
  &lt;arg name="robot" default="seven_dof_arm"/>
  &lt;arg name="group"      default="arm"/>
  &lt;arg name="end_effector" default="gripper"/>

  &lt;node pkg="moveit_simple_grasps" type="moveit_simple_grasps_server" name="moveit_simple_grasps_server">
    &lt;param name="group" value="$(arg group)"/>
    &lt;param name="end_effector" value="$(arg end_effector)"/>

    &lt;rosparam command="load" file="$(find seven_dof_arm_gazebo)/config/$(arg robot)_grasp_data.yaml"/>

  &lt;/node>

&lt;/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:

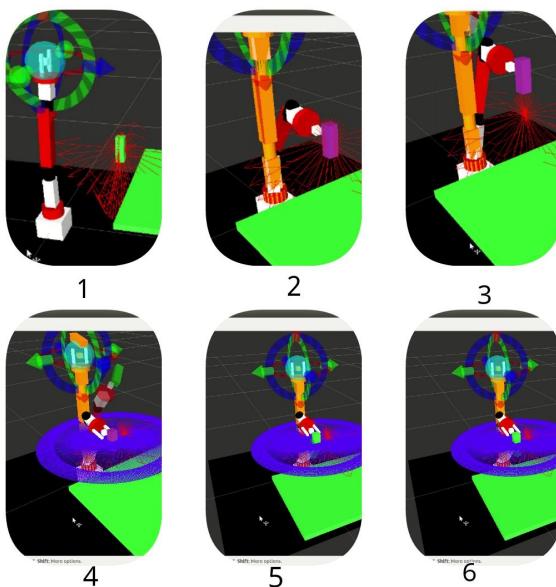


Figure 12 : Pick and place sequences using MoveIt!.

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 `def _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:

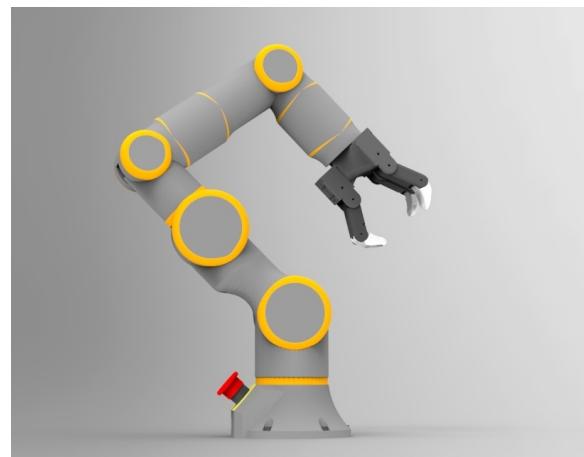


Figure 15 : COOL Arm illustration.

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 section 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>

```

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`:

```
&lt;launch>
&lt;!--
 Set the param that trajectory_execution_manager needs to find the controller plugin
-->
&lt;arg name="moveit_controller_manager" default="MoveIt_simple_controller_manager/MoveItSimpleControllerManager"/>

&lt;param name="MoveIt_controller_manager" value="$(arg MoveIt_controller_manager)"/>

&lt;!-- load controller_list -->

&lt;rosparam file="$(find cool5000_moveit_config)/config/controllers.yaml"/>
&lt;/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

- What is the role of the FCL library in MoveIt!?
- How does MoveIt! build OctoMap of the environment?
- What is the main function of grasp server?
- 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.

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>).

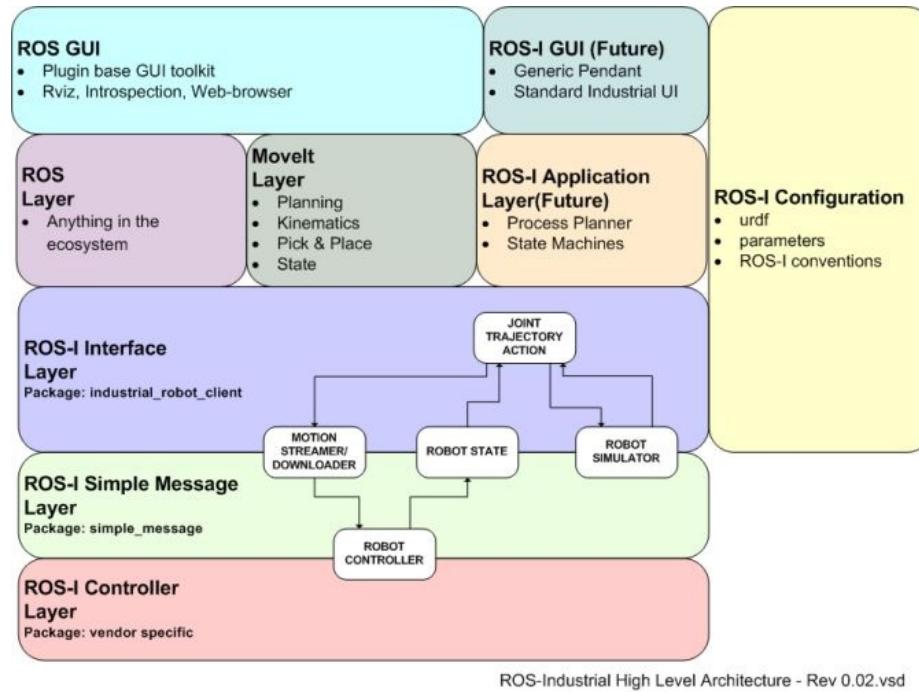


Figure 2: The block diagram of ROS-Industrial

- **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:

```
| $ roslaunch 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 $\sim 80,000$. This value can increase the collision checking in the arm
- Add a Virtual Jointmatrix as shown in the following screenshot. Here the virtual and parent frame names are arbitrary.

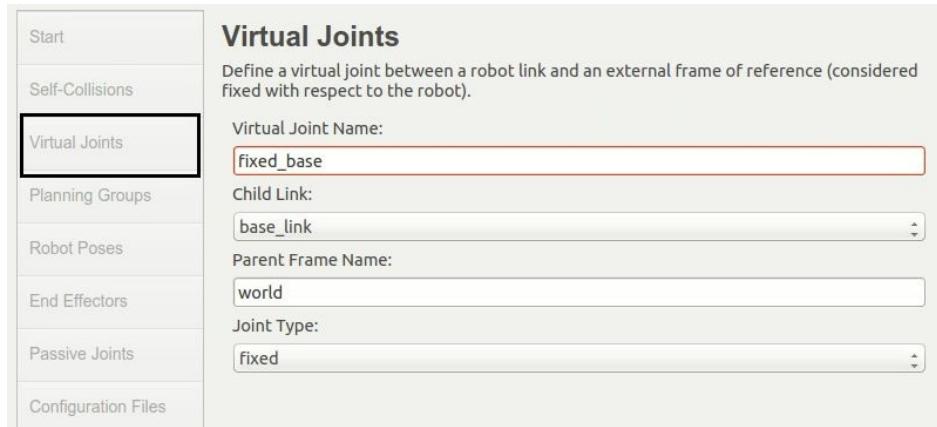


Figure 3: Adding MoveIt! - Virtual joints

- 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 manger, 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:

```
| $ roslaunch <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:

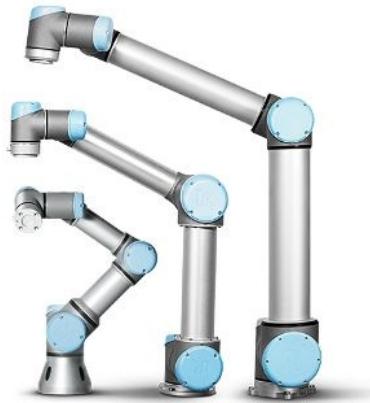


Figure 7: UR-3, UR-5, and UR-10 robots

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
```

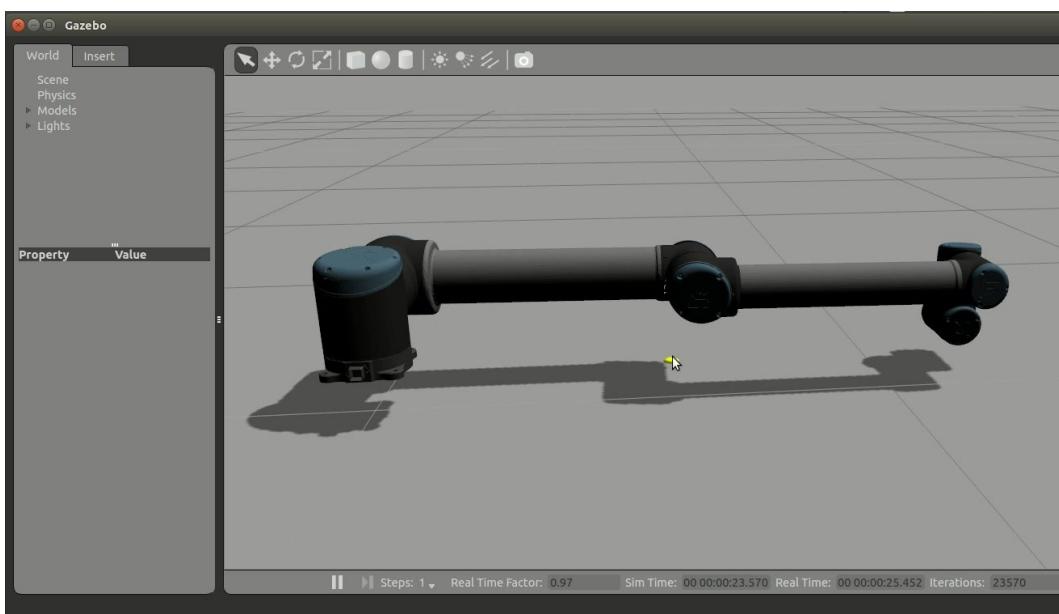


Figure 8: Universal robot, UR-10 model simulation in Gazebo

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:

```
$ rosrun ur10_moveit_config ur10_moveit_planning_execution.launch sim:=true
```

3. Launch RViz with the MoveIt! visualization plugin:

```
$ rosrun ur10_moveit_config moveit_rviz.launch config:=true
```

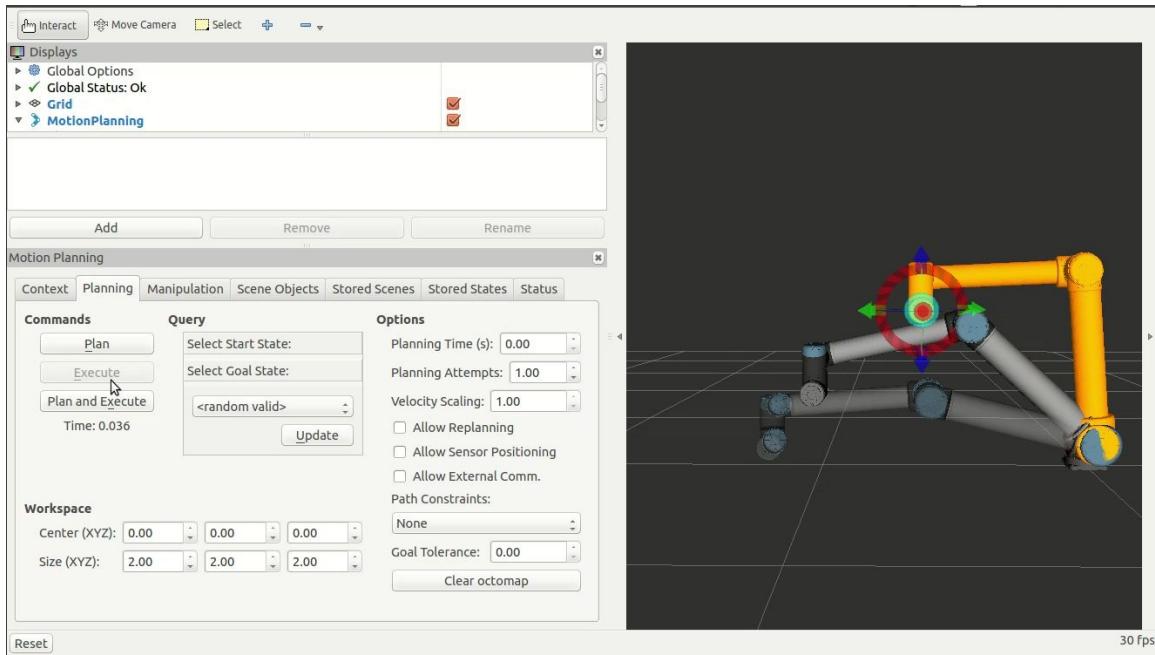


Figure 9: Motion planning in UR-10 model in RViz

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.

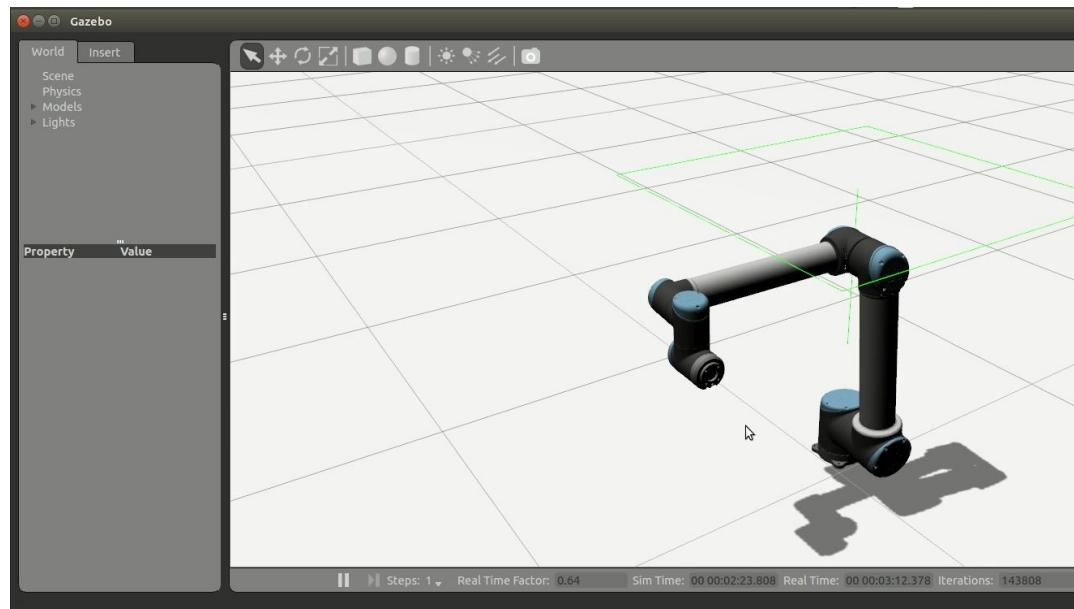


Figure 10 : Motion planned trajectory from MoveIt! executing in Gazebo

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:

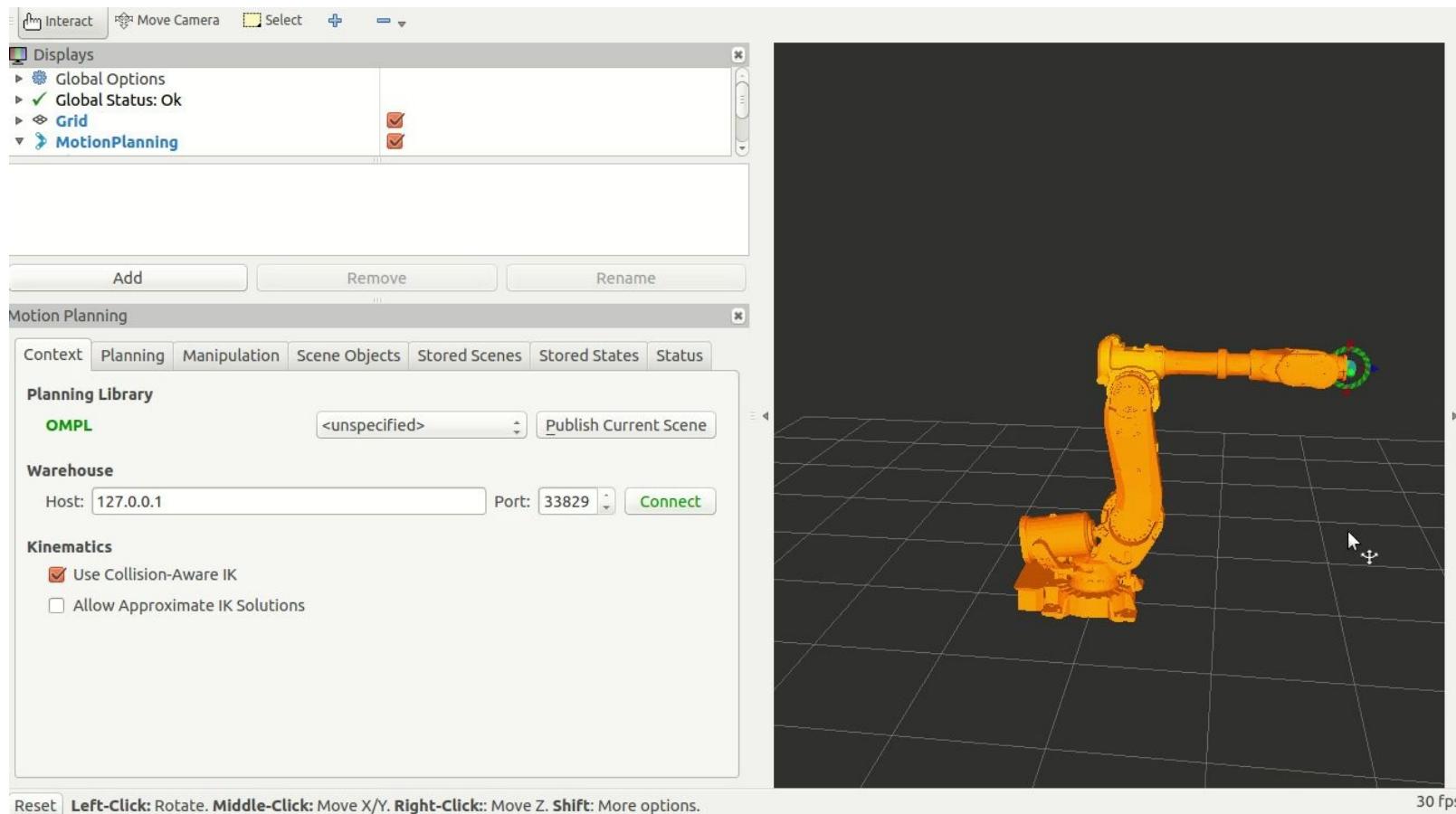


Figure 12: Motion planning of ABB IRB 6640

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:

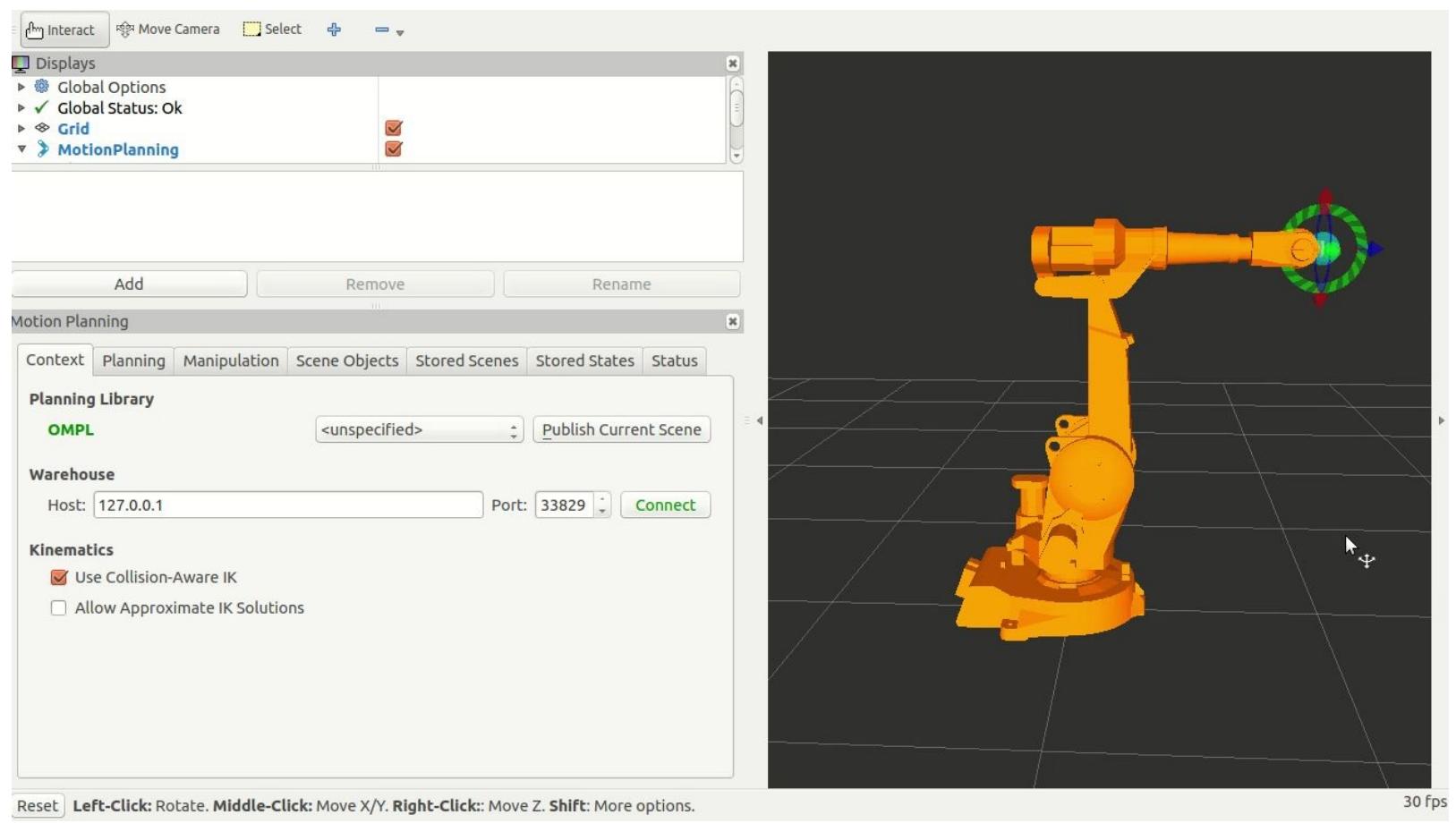


Figure 13: Motion planning of ABB IRB 2400

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:

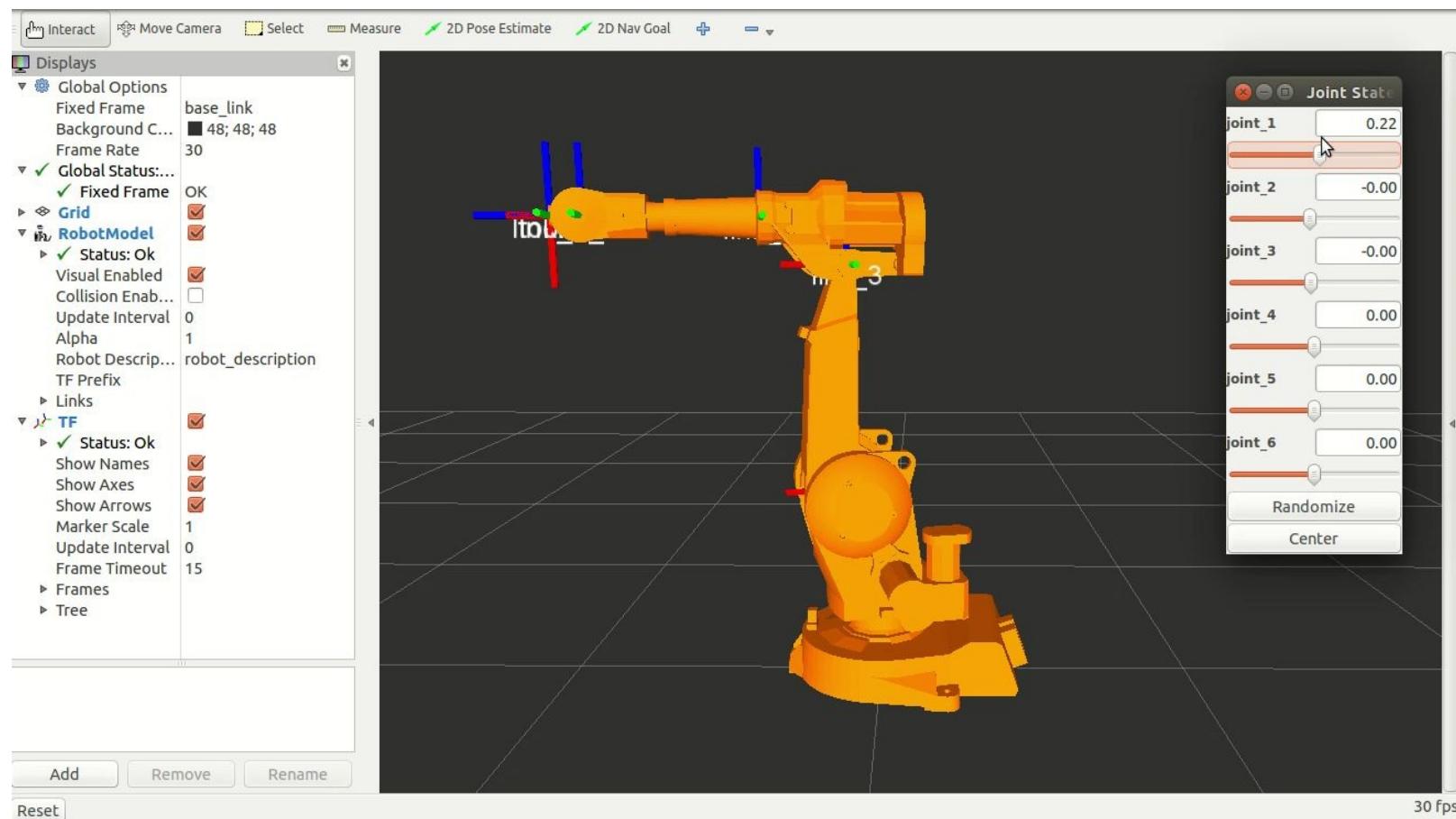


Figure 14: ABB IRB 2400 with joint state publisher on RViz

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:

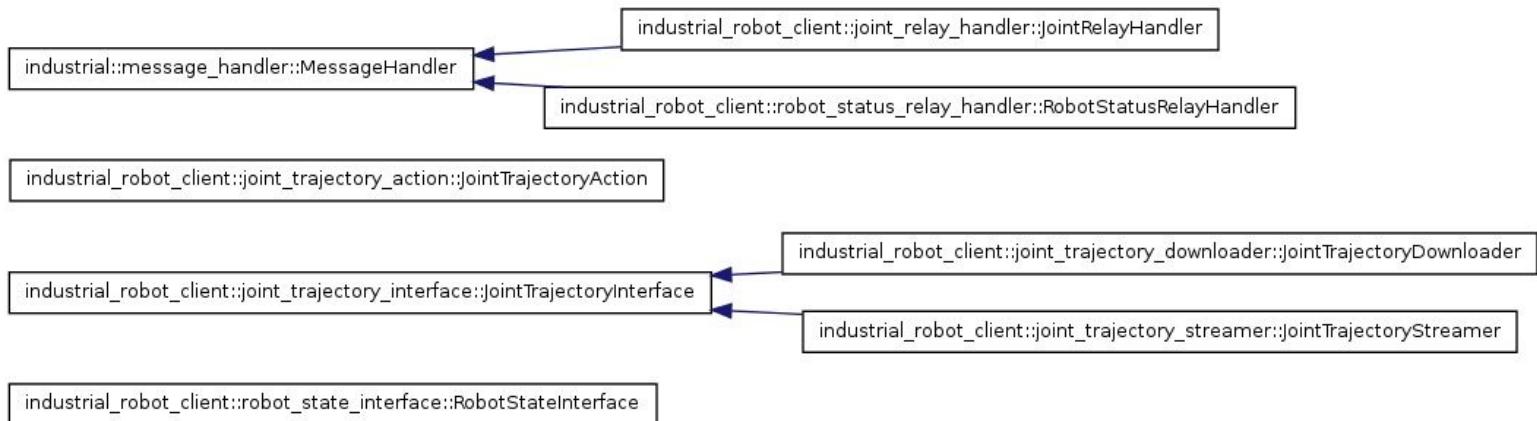


Figure 15: A list of the industrial robot client APIs

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.

- Create a `build` folder inside the OpenRave cloned folder and open `cmake-gui` for configuring and building Makefiles.
- Browse the source code and the `build` folder, as shown in the following screenshot, and after configuring uncheck support for Matlab and Octave interfaces:

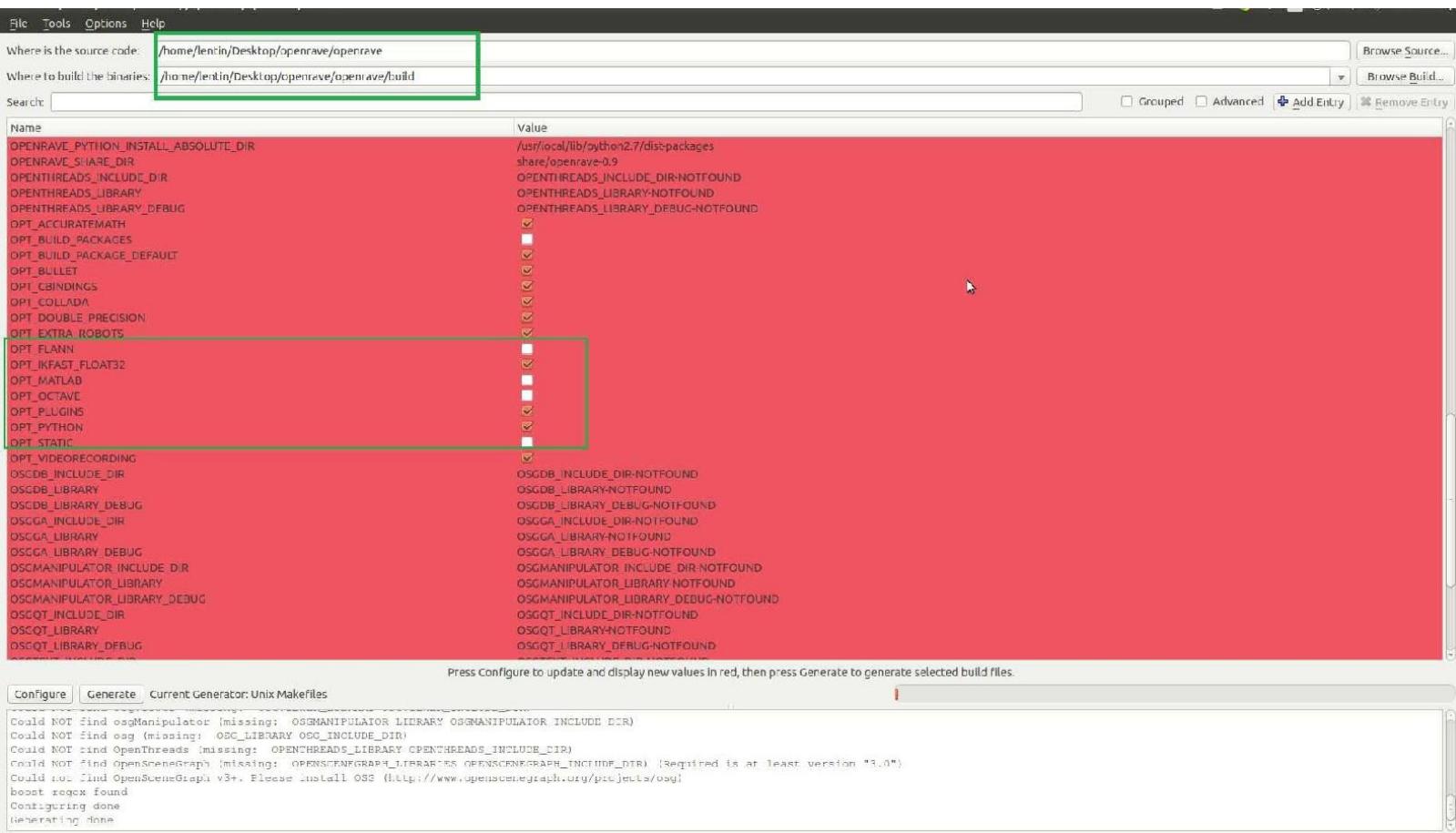


Figure 15: Configuring OpenRave with cmake-gui

- Click on the Generate button to generate the Makefiles in the `build` folder.
- Switch to the `build` folder and build the code and install using the following command:

```
$ make
$ sudo make install
```

- 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:

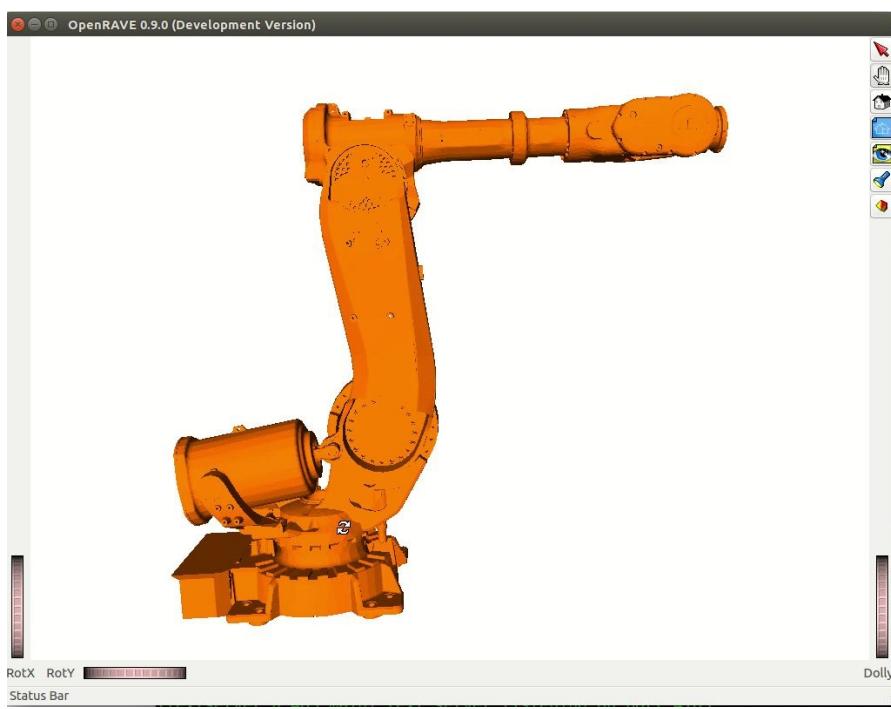


Figure 16: Viewing the ABB 6640 model on OpenRave

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/ikfast_demo/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_moveit_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/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:

```
$ roslaunch abb_irb6640_moveit_config demo.launch
```

We will get the planning window with a ne IK solver as follows:

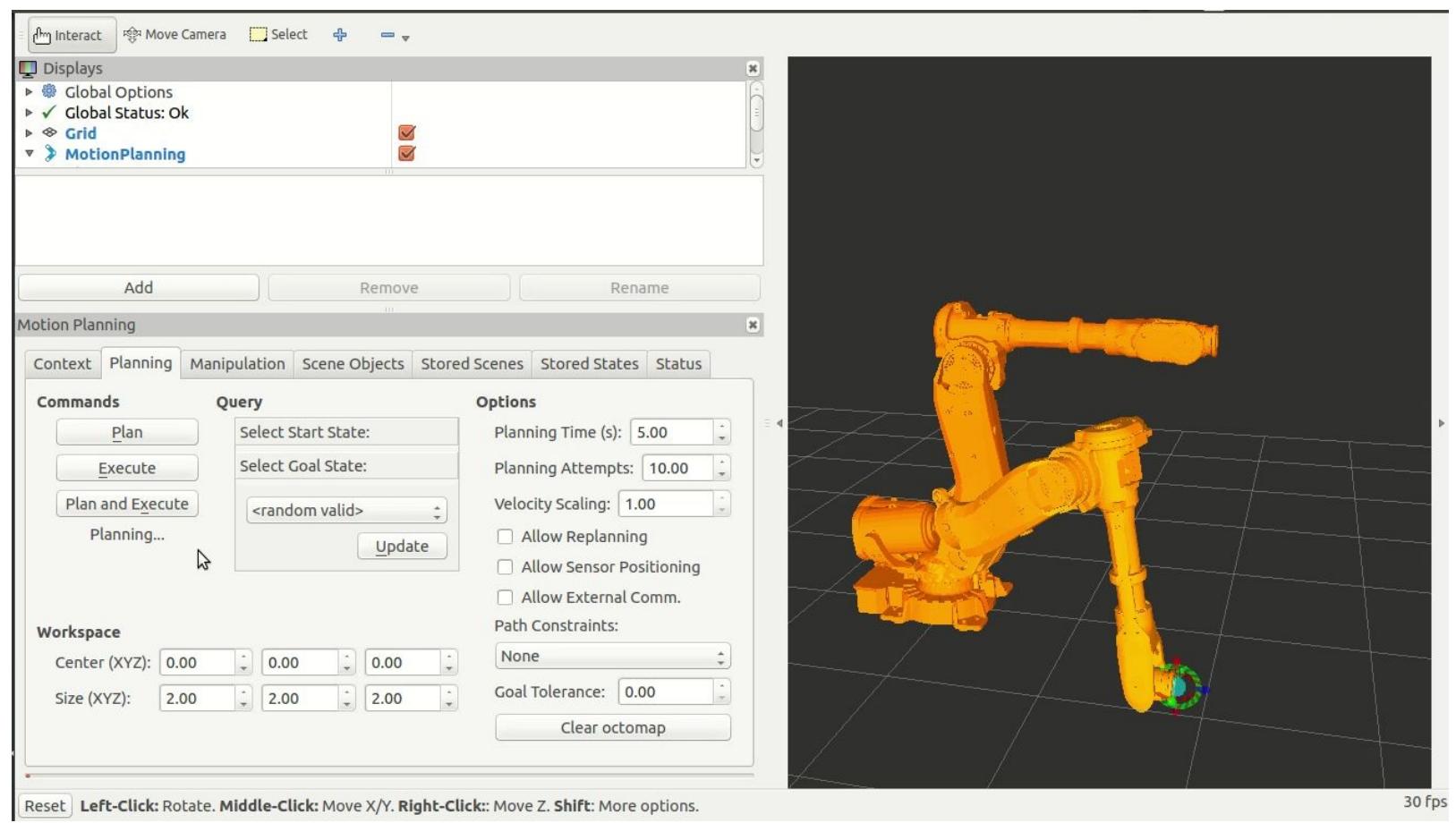


Figure 18: Motion planning of ABB 6640 using custom IKFast Plugin

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.

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 module, 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>.

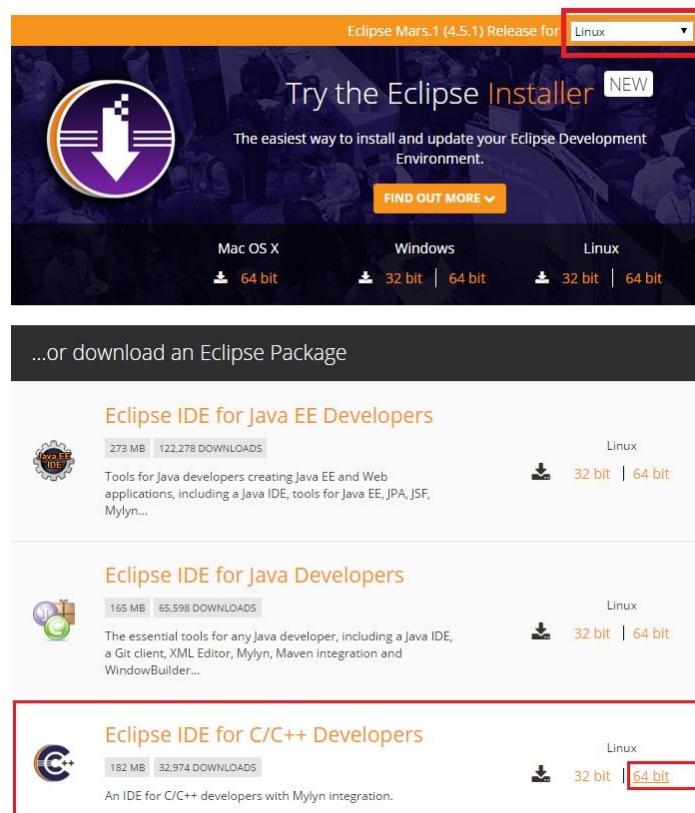


Figure 1: Eclipse IDE download page

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 -xvf 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:

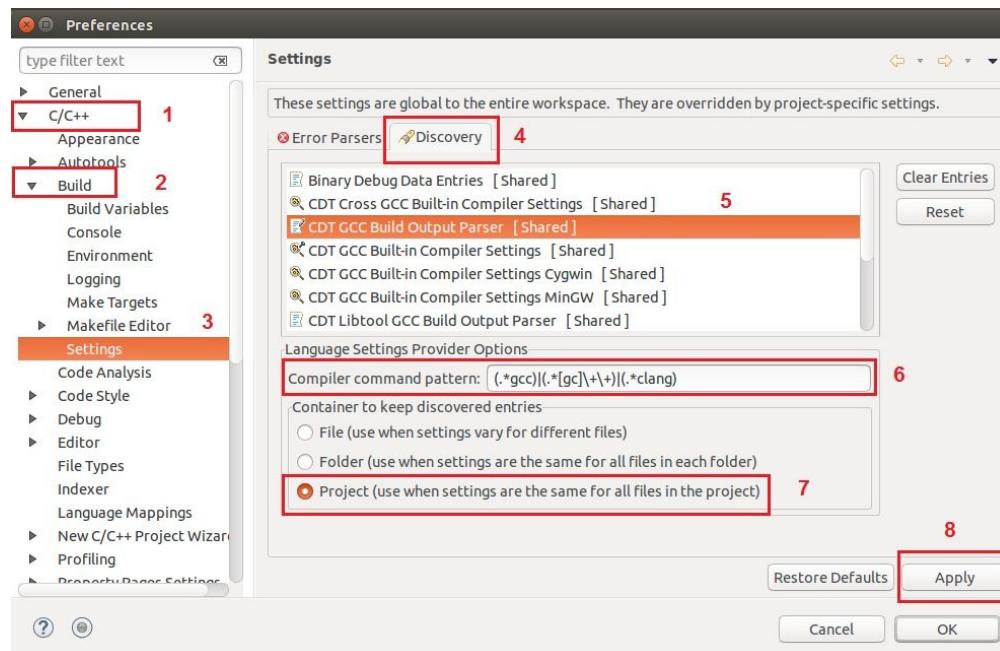


Figure 2: Settings inside Eclipse Preferences

- 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}"`.

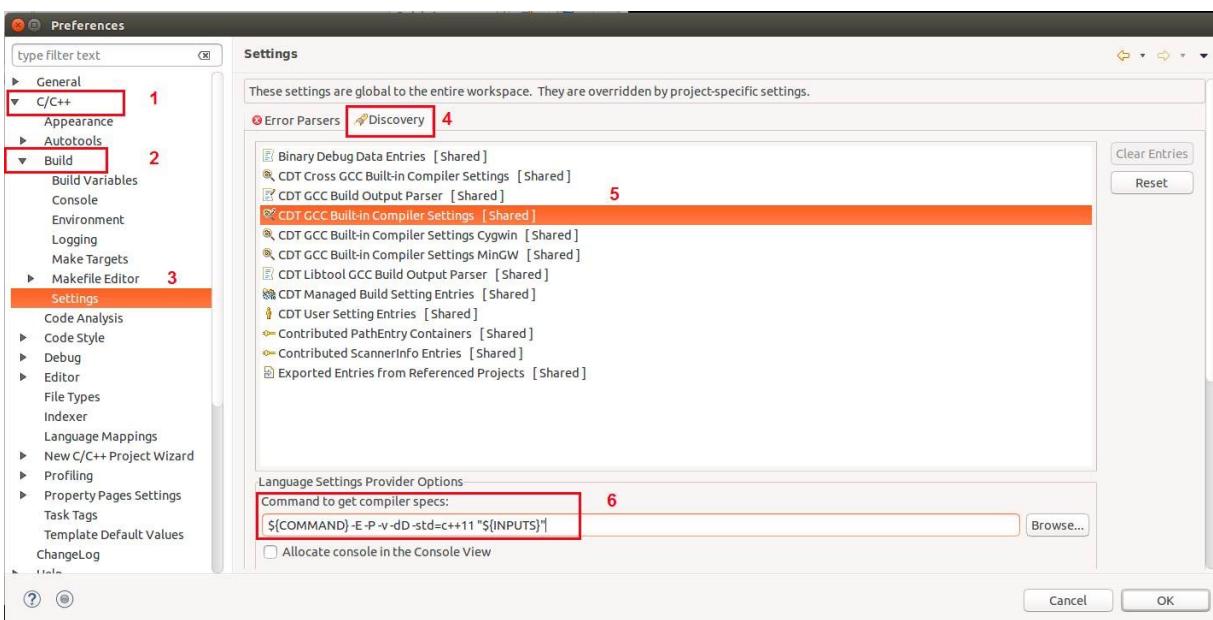


Figure 3: Eclipse Compiler settings

ROS compile script for Eclipse IDE

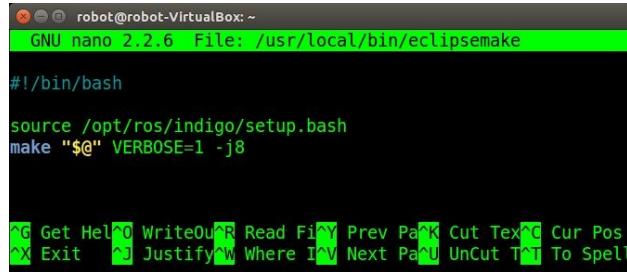
Compiling ROS nodes needs the ROS environment. We have to source `/opt/ros/indigo/setup.bash` to access the ROS environment in the current terminal. We can work from the system terminal because we already added this line to the `.bashrc` file, but when we work using Eclipse, we have to make a script 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
```



The screenshot shows a terminal window titled "robot@robot-VirtualBox: ~". The title bar also displays "GNU nano 2.2.6 File: /usr/local/bin/eclipsemake". The main area of the terminal contains the following text:

```
#!/bin/bash  
source /opt/ros/indigo/setup.bash  
make "$@" VERBOSE=1 -j8
```

At the bottom of the terminal window, there is a menu bar with various keyboard shortcuts for text editing.

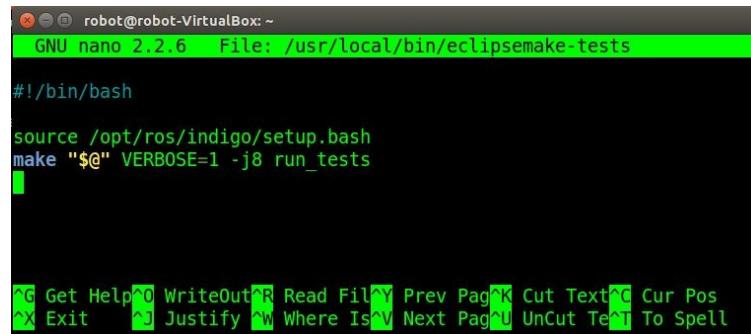
Figure 4: Script to source the ROS environment

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
```



The screenshot shows a terminal window titled "robot@robot-VirtualBox: ~". The title bar also displays "GNU nano 2.2.6 File: /usr/local/bin/eclipsemake-tests". The main area of the terminal contains the following text:

```
#!/bin/bash  
source /opt/ros/indigo/setup.bash  
make "$@" VERBOSE=1 -j8 run_tests
```

At the bottom of the terminal window, there is a menu bar with various keyboard shortcuts for text editing.

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++ | MakeFileProject with Existing Code:

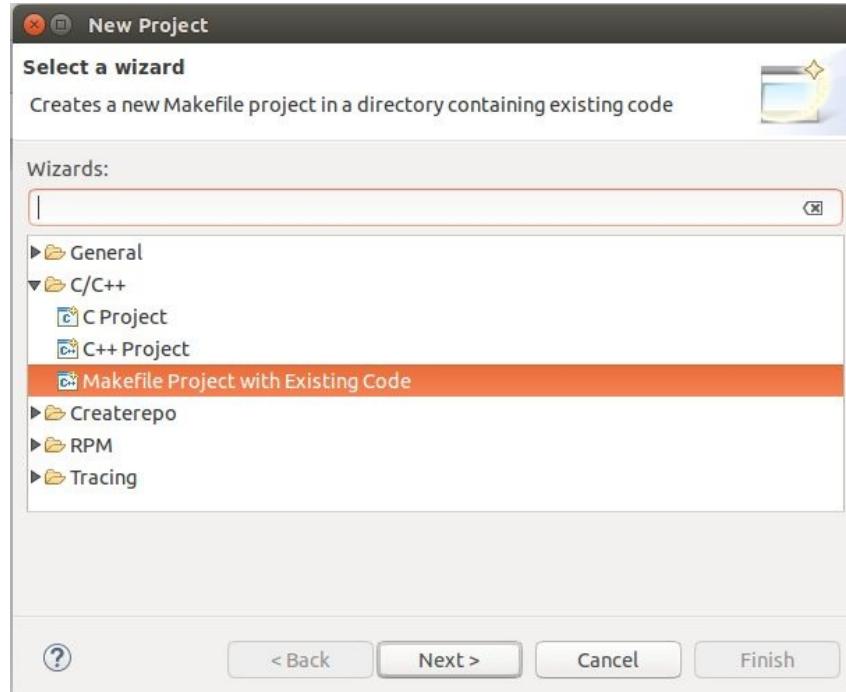


Figure 6: Open the ROS package from the catkin workspace

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:

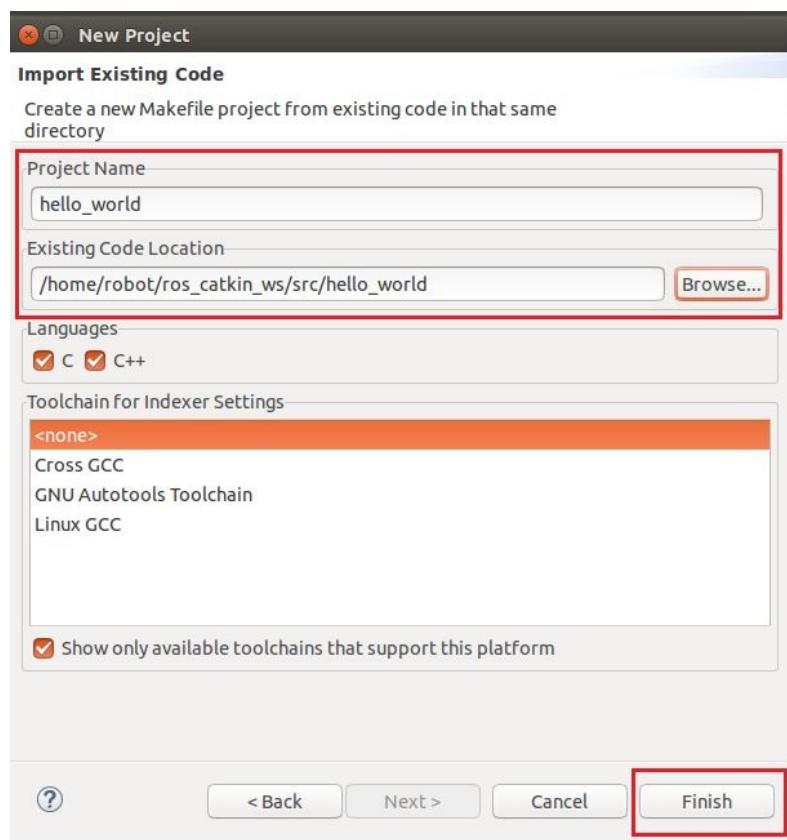


Figure 7: Giving project and location

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:

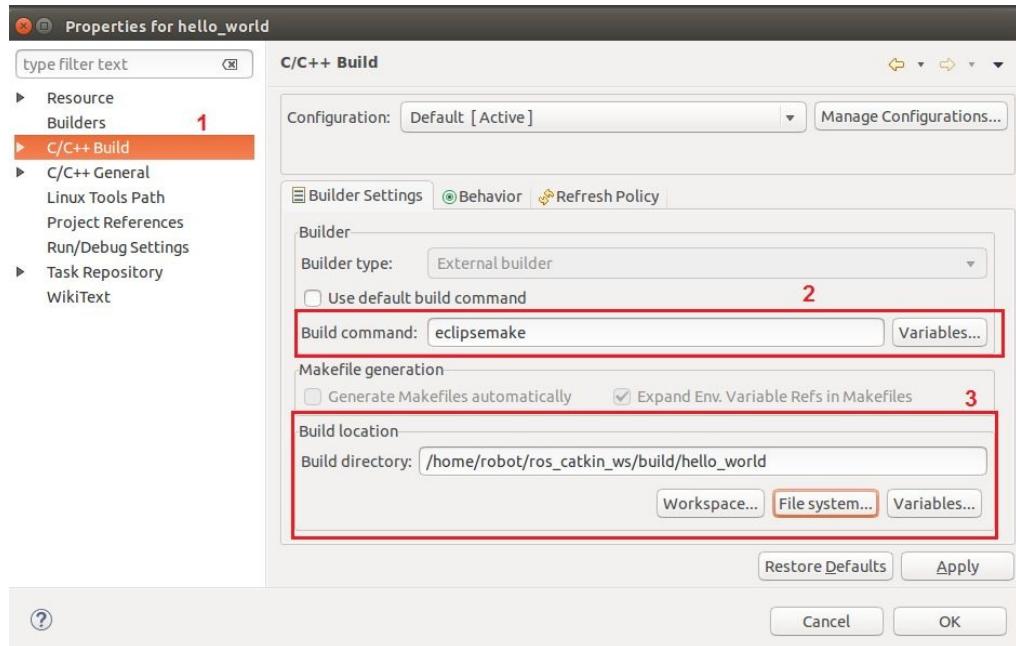


Figure 8: Eclipse build settings for ROS

In C/C++ Build | Environment, add a new variable called `VERBOSE` and set the value as `1`, as shown in the following screenshot:

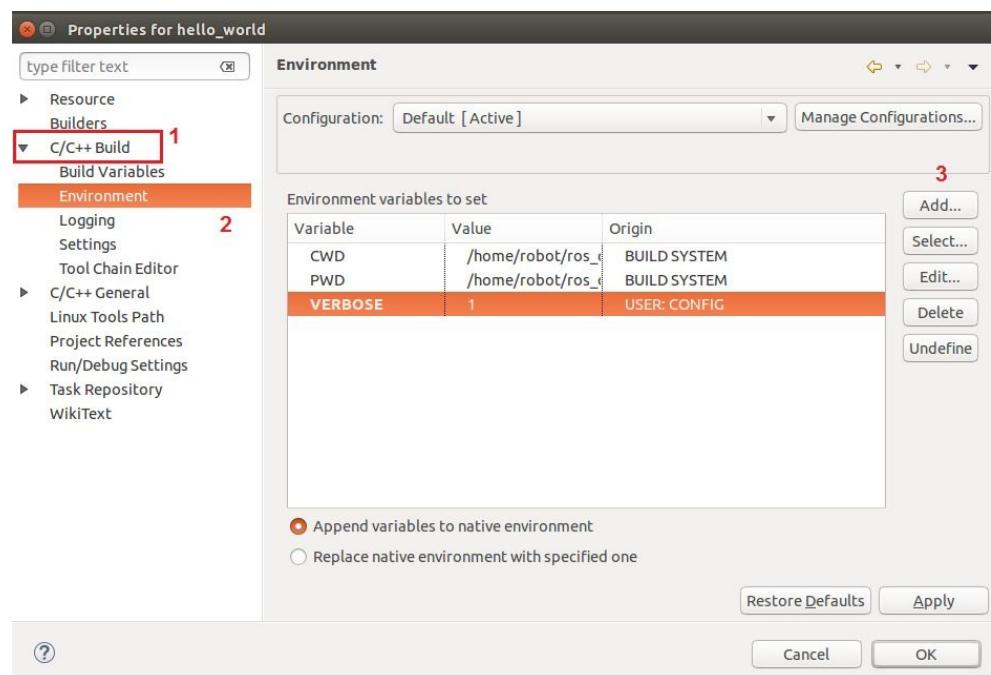


Figure 9: Setting VERBOSE in ROS project properties

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:

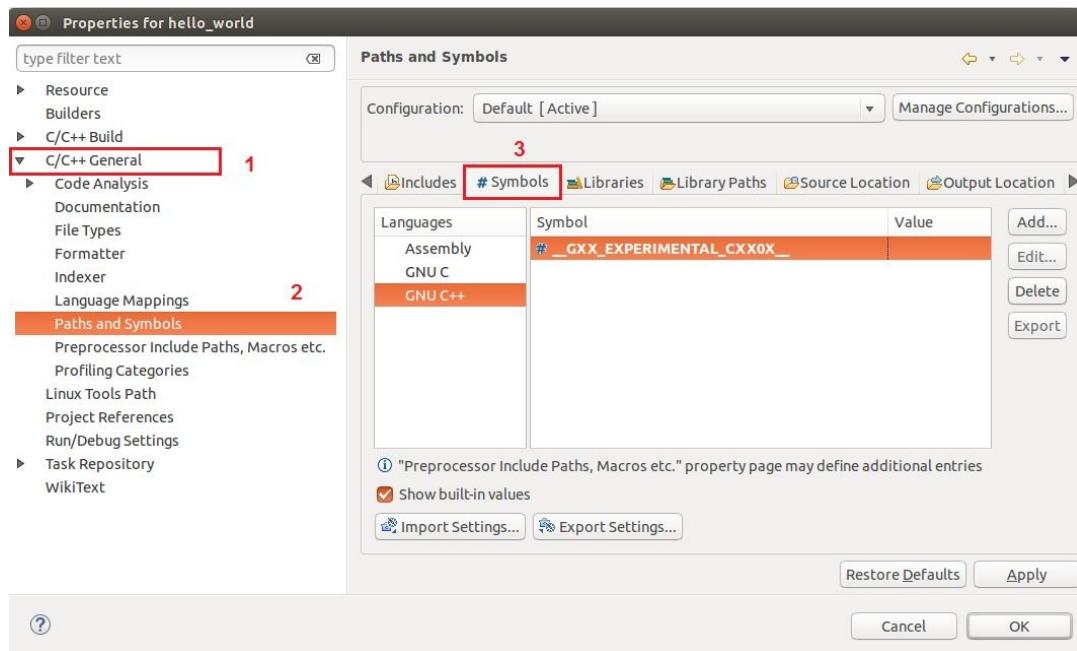


Figure 10: Setting path and symbols for the ROS project

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:

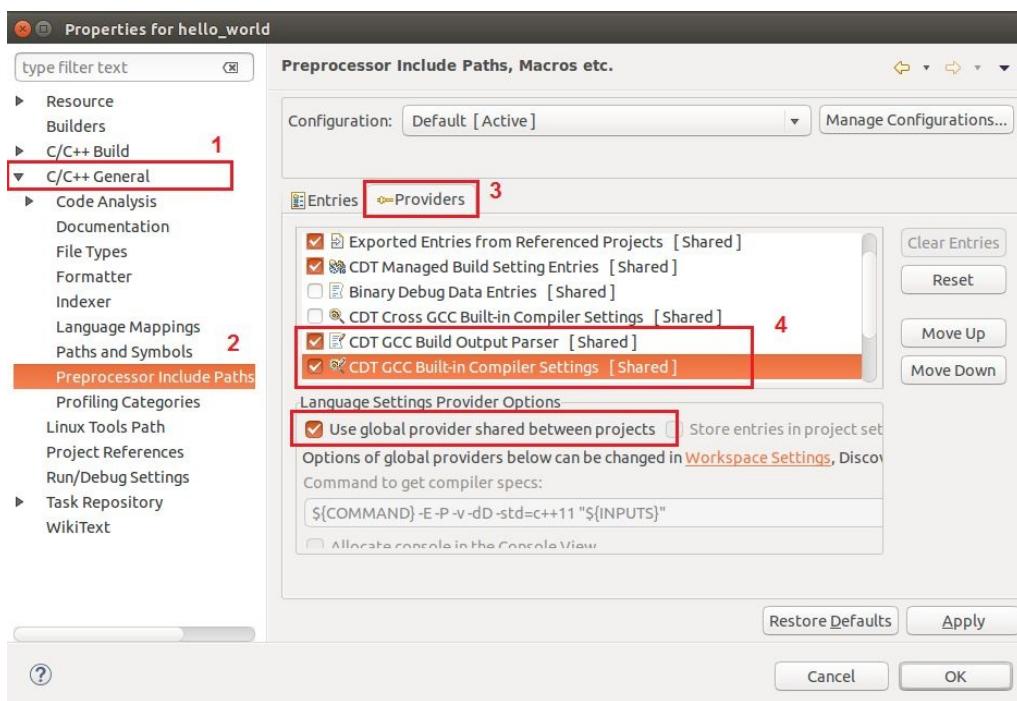


Figure 11: Setting Pre-processor in the project properties

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:

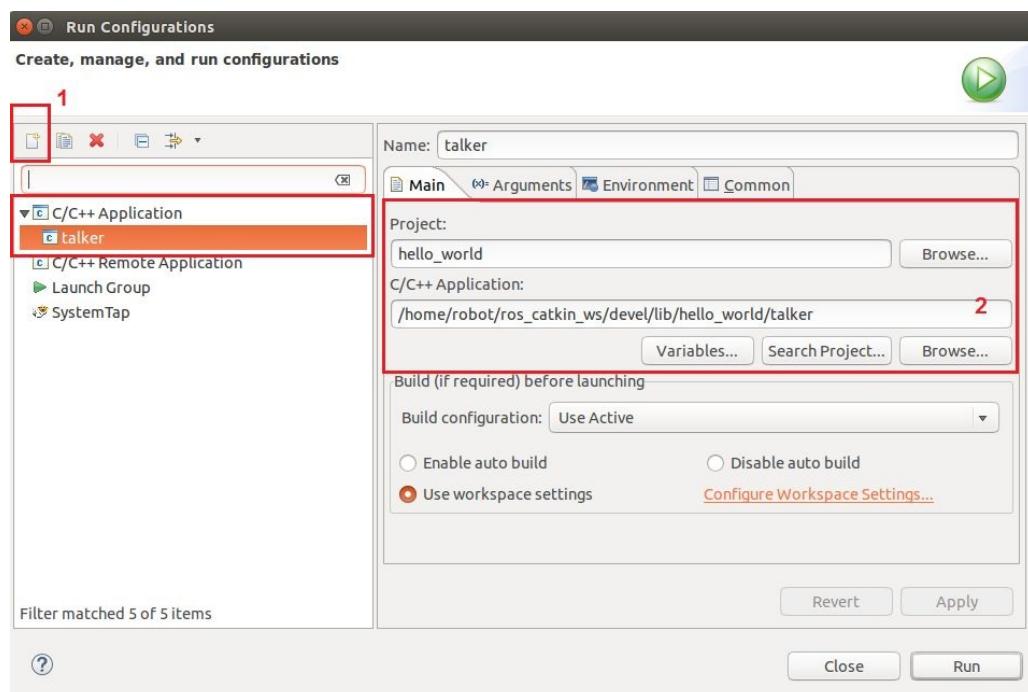


Figure 12: Creating the launcher for the talker node

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
```

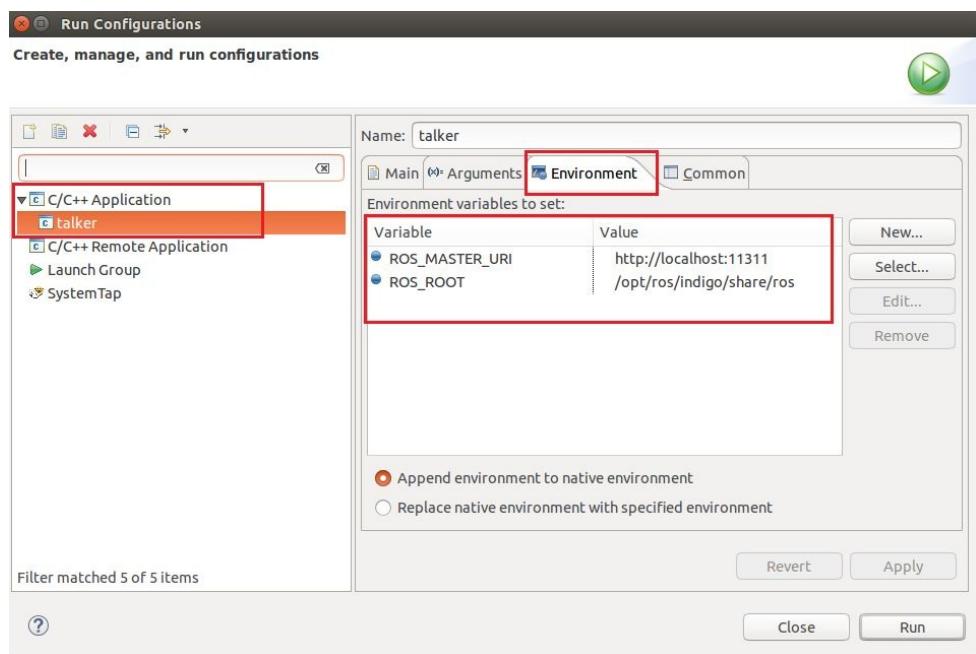


Figure 13: Setting the ROS environment variable inside the launcher configuration

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:

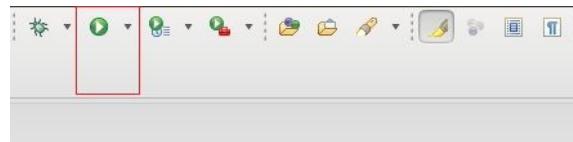


Figure 14: Launching the talker node

We can see the output in the Eclipse console, as shown in the following screenshot:

```
[0m[ INFO] [1444376599.098332095]: hello world 363[0m[0m
[0m[ INFO] [1444376599.198849951]: hello world 364[0m[0m
[0m[ INFO] [1444376599.298491738]: hello world 365[0m[0m
[0m[ INFO] [1444376599.398298238]: hello world 366[0m[0m
[0m[ INFO] [1444376599.497785021]: hello world 367[0m[0m
[0m[ INFO] [1444376599.597852038]: hello world 368[0m[0m
[0m[ INFO] [1444376599.698617732]: hello world 369[0m[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 `roswtf`. `roswtf`, 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 roswhf

We can check the issues inside a ROS package by simply entering the package and entering `roswhf`. We can also check issues in the launch file using the following command:

```
$ roswhf <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: roswhf command output for a ROS package

The wiki page of `roswhf` is available at <http://wiki.ros.org/roswtf>.

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:~$ rosrun roscpp_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': '992ce8a1687cec8c8bd883ec73ca41d1', '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::st
ring const&, std::map<std::string, std::string, std::less<std::string>, std::allocat
or<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::initializeLogLocatio
n(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::checkLogLocationEnab
```

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.

ROS Robotics Projects

Build a variety of awesome robots that can see, sense, move, and do a lot more using the powerful Robot Operating System

Face Detection and Tracking Using ROS, OpenCV and Dynamixel Servos

One of the capabilities of most service and social robots is face detection and tracking. These robots can identify faces and can move their heads according to the human face that moves around it. There are numerous implementations of face detection and tracking systems on the Web. Most trackers have a pan-and-tilt mechanism, and a camera is mounted on the top of the servos. In this chapter, we will see a simple tracker that only has a pan mechanism. We are going to use a USB webcam mounted on an AX-12 Dynamixel servo. The controlling of Dynamixel servo and image processing are done in ROS.

The following topics will be covered in this chapter:

- An overview of the project
- Hardware and software prerequisites
- Configuring Dynamixel AX-12 servos
- The connection diagram of the project
- Interfacing Dynamixel with ROS
- Creating ROS packages for a face tracker and controller
- The ROS-OpenCV interface
- Implementing a face tracker and face tracker controller
- The final run

Overview of the project

The aim of the project is to build a simple face tracker that can track face only along the horizontal axis of the camera. The face tracker hardware consists of a webcam, Dynamixel servo called AX-12, and a supporting bracket to mount the camera on the servo. The servo tracker will follow the face until it aligns to the center of the image from the webcam. Once it reaches the center, it will stop and wait for face movement. The face detection is done using an OpenCV and ROS interface, and the controlling of the servo is done using a Dynamixel motor driver in ROS.

We are going to create two ROS packages for this complete tracking system; one is for face detection and finding the centroid of the face, and the other is for sending commands to the servo to track the face using the centroid values.

Okay! Let's start discussing the hardware and software prerequisites of this project.



*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/ros_robotics_projects*

Hardware and software prerequisites

The following is a table of the hardware components that can be used for building this project. You can also see the rough price and a purchase link for each component.

List of hardware components:

| No | Component name | Estimated price (USD) | Purchase link |
|----|--|-----------------------|---|
| 1 | Webcam | 32 | https://amzn.com/B003LVZO8S |
| 2 | Dynamixel AX-12A servo with mounting bracket | 76 | https://amzn.com/B0051OXJXU |
| 3 | USB-to-Dynamixel Adapter | 50 | http://www.robotshop.com/en/robotis-usb-to-dynamixel-adapter.html |
| 4 | Extra 3-pin cables for AX-12 servos | 12 | http://www.trossenrobotics.com/p/100mm-3-Pin-DYNAMIXEL-Compatible-Cable-10-Pack |
| 5 | Power adapter | 5 | https://amzn.com/B005JRGOCM |
| 6 | 6-port AX/MX power hub | 5 | http://www.trossenrobotics.com/6-port-ax-mx-power-hub |
| 7 | USB extension cable | 1 | https://amzn.com/B00YBKA5Z0 |
| | Total cost with shipping and tax | Around 190-200 | |



The URLs and prices can vary. If the links are not available, a Google search might do the job. The shipping charges and tax are excluded from the prices.

If you are thinking that the total cost is not affordable, then there are cheap alternatives to do this project too. The main heart of this project is the Dynamixel servo. We can replace this servo with RC servos, which only cost around \$10, and an Arduino board costing around \$20 can be used to control the servo too. The ROS and Arduino interfacing will be discussed in the upcoming chapters, so you can think about porting the face tracker project using an Arduino and RC servo.

Okay, let's look at the software prerequisites of the project. The prerequisites include the ROS framework, OS version, and ROS packages:

| No | Name of the software | Estimated price (USD) | Download link |
|----|----------------------|-----------------------|---|
| 1 | Ubuntu 16.04 LTS | Free | http://releases.ubuntu.com/16.04/ |
| 2 | ROS Kinetic LTS | Free | http://wiki.ros.org/kinetic/Installation/Ubuntu |
| 3 | ROS usb_cam package | Free | http://wiki.ros.org/usb_cam |

| | | | |
|----------|--------------------------------|--------------|---|
| 4 | ROS cv_bridge package | Free | http://wiki.ros.org/cv_bridge |
| 5 | ROS Dynamixel controller | Free | https://github.com/arebgun/dynamixel_motor |
| 6 | Windows 7 or higher | Around \$120 | https://www.microsoft.com/en-in/software-download/windows7 |
| 7 | RoboPlus (Windows application) | Free | http://www.robotis.com/download/software/RoboPlusWeb%28v1.1.3.0%29.exe |

This table gives you an idea of the software we are going to be using for this project. We may need both Windows and Ubuntu for doing this project. It will be great if you have dual operating systems on your computer.

Let's see how to install all this software first.

Installing dependent ROS packages

We have already installed and configured Ubuntu 16.04 and ROS Kinetic. Let's look at the dependent packages we need to install for this project.

Installing the `usb_cam` ROS package

Let's look at the use of the `usb_cam` package in ROS first. The `usb_cam` package is the ROS driver for **Video4Linux (V4L)** USB cameras. V4L is a collection of device drivers in Linux for real-time video capture from webcams. The `usb_cam` ROS package works using V4L devices and publishes the video stream from devices as ROS image messages. We can subscribe to it and perform our own processing using it. The official ROS page of this package is given in the previous table. You can check out this page for different settings and configurations this package offers.

Creating a ROS workspace for dependencies

Before starting to install the `usb_cam` package, let's create a ROS workspace for storing the dependencies of all the projects mentioned in the section. We can create another workspace for keeping the project code.

Create a ROS workspace called `ros_project_dependencies_ws` in the home folder. Clone the `usb_cam` package into the `src` folder:

```
| $ git clone https://github.com/bosch-ros-pkg/usb_cam.git
```

Build the workspace using `catkin_make`.

After building the package, install the `v4l-utils` Ubuntu package. It is a collection of command-line V4L utilities used by the `usb_cam` package:

```
| $ sudo apt-get install v4l-utils
```

Configuring a webcam on Ubuntu 16.04

After installing these two, we can connect the webcam to the PC to check whether it is properly detected by our PC.

Open a Terminal and execute the `dmesg` command to check the kernel logs. If your camera is detected in Linux, it may give you logs like this:

```
| $ dmesg
```



The terminal window displays the kernel logs for a USB webcam connection. The logs show the device being detected as a high-speed USB device, its vendor and product IDs, and its manufacturer and serial number. It also shows the creation of a media interface, a Linux video capture interface, and a USB video class driver. The logs indicate that the device is an iBall Face2Face Webcam C12.0.

```
[ 86.483102] usb 1-1.5: new high-speed USB device number 6 using ehci-pci
[ 86.620403] usb 1-1.5: New USB device found, idVendor=0c45, idProduct=6340
[ 86.620409] usb 1-1.5: New USB device strings: Mfr=2, Product=1, SerialNumber=3
[ 86.620412] usb 1-1.5: Product: iBall Face2Face Webcam C12.0
[ 86.620414] usb 1-1.5: Manufacturer: iBall Face2Face Webcam C12.0
[ 86.620416] usb 1-1.5: SerialNumber: iBall Face2Face Webcam C12.0
[ 86.657389] media: Linux media interface: v0.10
[ 86.677503] Linux video capture interface: v2.00
[ 86.703833] usb 1-1.5: 3:1: cannot get freq at ep 0x84
[ 86.722072] usbcore: registered new interface driver snd-usb-audio
[ 86.722096] uvcvideo: Found UVC 1.00 device iBall Face2Face Webcam C12.0 (0c45:6340)
[ 86.735670] input: iBall Face2Face Webcam C12.0 as /devices/pci0000:00/0000:00:00:1a.0/
t/input16
[ 86.735747] usbcore: registered new interface driver uvcvideo
[ 86.735749] USB Video Class driver (1.1.1)
```

Figure 1: Kernels logs of the webcam device

You can use any webcam that has driver support in Linux. In this project, an iBall Face2Face (<http://www.iball.co.in/Product/Face2Face-C8-0--Rev-3-0-90>) webcam is used for tracking. You can also go for the popular Logitech C310 webcam mentioned as a hardware prerequisite. You can opt for that for better performance and tracking.

If our webcam has support in Ubuntu, we can open the video device using a tool called **Cheese**. Cheese is simply a webcam viewer.

Enter the command `cheese` in the Terminal. If it is not installed, you can install it using the following command:

```
| $ sudo apt-get install cheese
```

If the driver and device are proper, you will get a video stream from the webcam, like this:

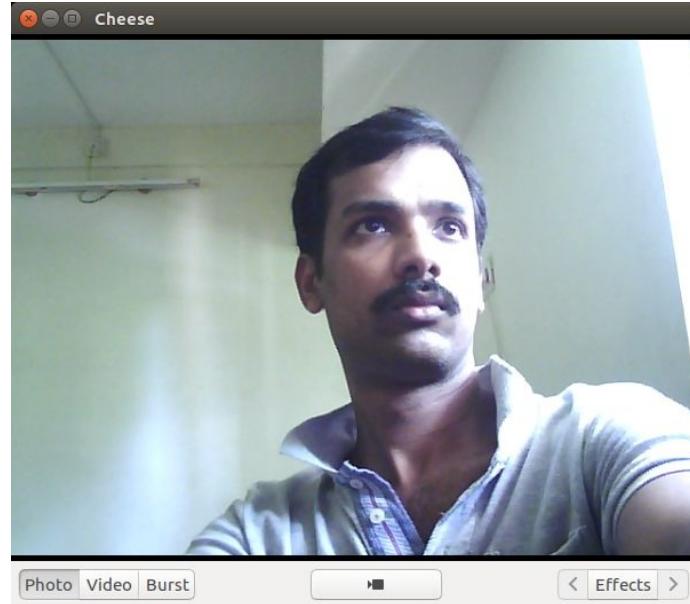


Figure 2: Webcam video streaming using Cheese

Congratulations! Your webcam is working well in Ubuntu, but are we done with everything? No. The next thing is to test the ROS `usb_cam` package. We have to make sure that it's working well in ROS!



*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/ros_robotics_projects*

Interfacing the webcam with ROS

Let's test the webcam using the `usb_cam` package. The following command is used to launch the `usb_cam` nodes to display images from a webcam and publish ROS image topics at the same time:

```
| $ roslaunch usb_cam usb_cam-test.launch
```

If everything works fine, you will get the image stream and logs in the Terminal, as shown here:

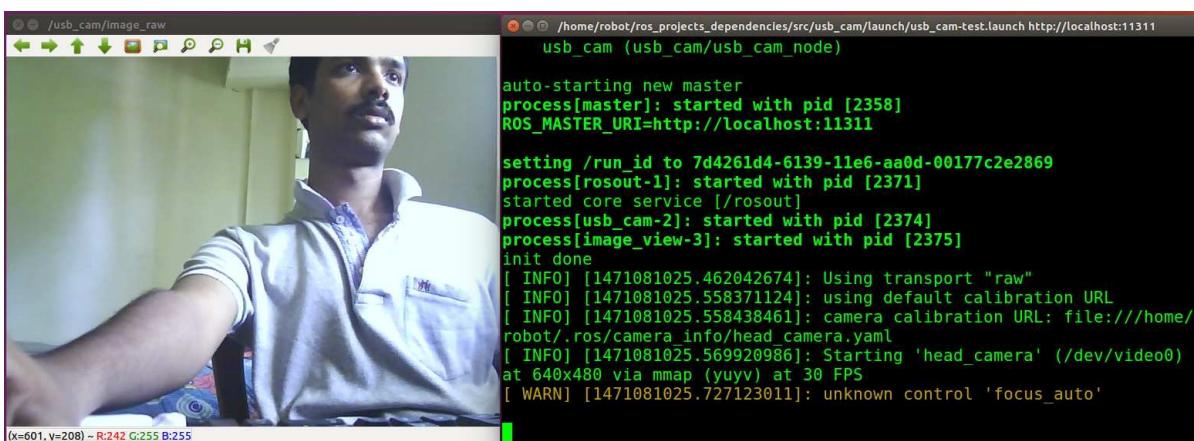


Figure 3: Working of the `usb_cam` package in ROS

The image is displayed using the `image_view` package in ROS, which is subscribed to the topic called `/usb_cam/image_raw`.

Here are the topics that `usb_cam` node is publishing:

```

robot@robot-pc:~$ rostopic list
/home/robot/ros_projects_dependencies/src/usb_c... x robot@robot-pc: ~
robot@robot-pc:~$ rostopic list
/image_view/parameter_descriptions
/image_view/parameter_updates
/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
robot@robot-pc:~$ 

```

Figure 4: The topics being published by the `usb_cam` node

We've finished interfacing a webcam with ROS. So what's next? We have to interface an AX-12 Dynamixel servo with ROS. Before proceeding to interfacing, we have to do something to configure this servo.

Next, we are going to see how to configure a Dynamixel AX-12A servo.

Configuring a Dynamixel servo using RoboPlus

The Dynamixel servo can be configured using a program called **RoboPlus**, provided by *ROBOTIS INC* (<http://en.robotis.com/index/>), the manufacturer of Dynamixel servos.

To configure Dynamixel, you have to switch your operating system to Windows. The RoboPlus tool works

on Windows. In this project, we are going to configure the servo in Windows 7.

Here is the link to download RoboPlus:

<http://www.robotis.com/download/software/RoboPlusWeb%28v1.1.3.0%29.exe>

If the link is not working, you can just search in Google for RoboPlus 1.1.3. After installing the software, you will get the following window. Navigate to the Expert tab in the software to get the application for configuring Dynamixel:

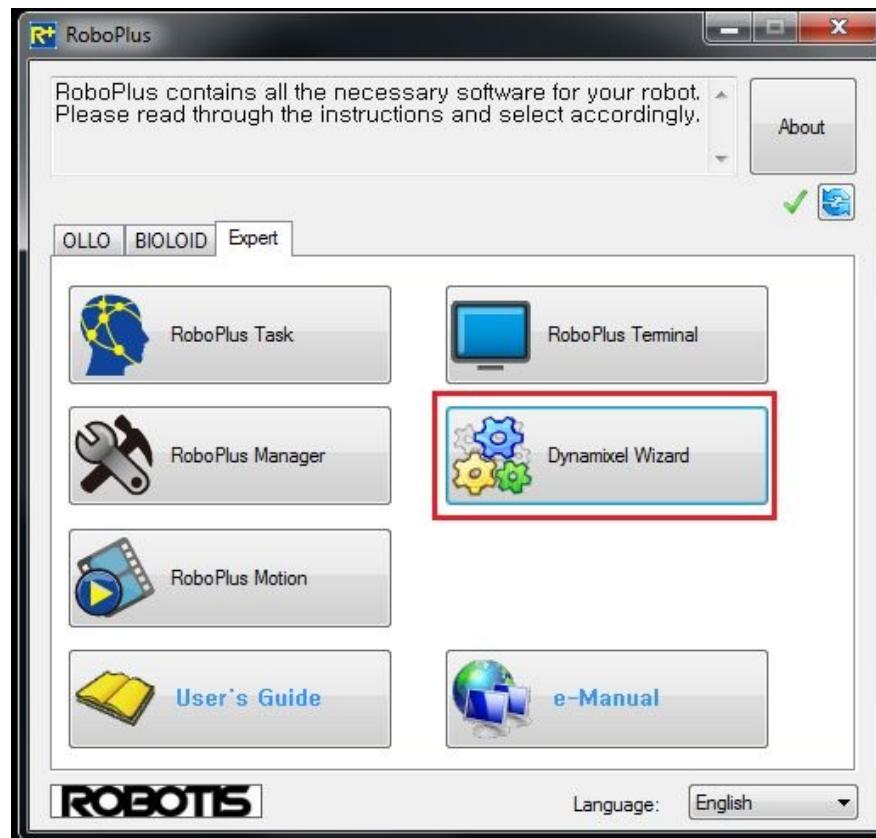


Figure 5: Dynamixel manager in RoboPlus

Before starting Dynamixel Wizard and configuring, we have to connect the Dynamixel and properly power it up. The following are images of the AX-12A servo we are using for this project and a diagram of its pin connection:

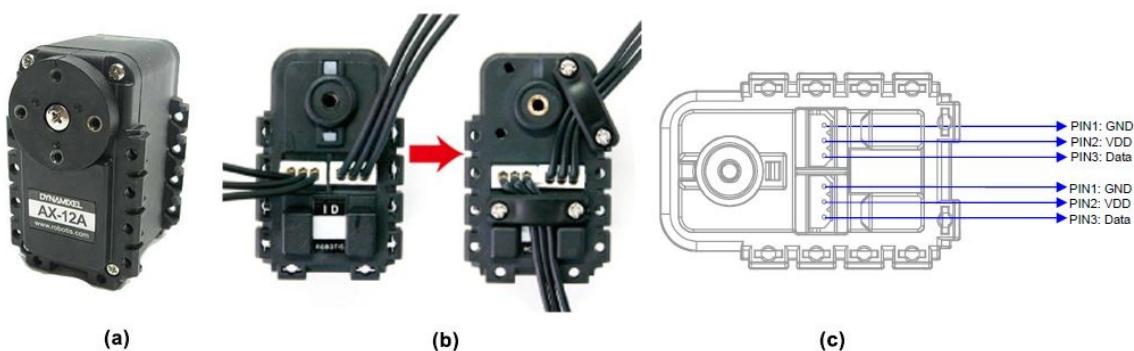


Figure 6: The AX-12A Dynamixel and its connection diagram

Unlike other RC servos, AX-12 is an intelligent actuator having a microcontroller that can monitor every

parameter of a servo and customize all of them. It has a geared drive, and the output of the servo is connected to a servo horn. We can connect any link to this servo horn. There are two connection ports behind each servo. Each port has pins such as VCC, GND, and Data. The ports of the Dynamixel are daisy-chained, so we can connect one servo to another servo. Here is the connection diagram of the Dynamixel with a computer:

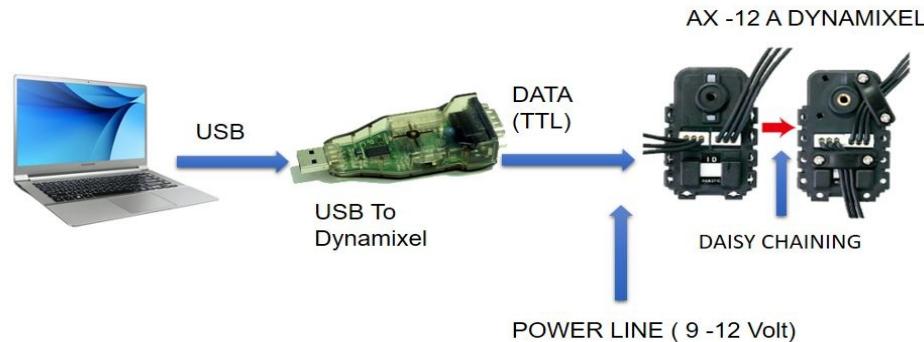


Figure 7: The AX-12A Dynamixel and its connection diagram

The main hardware component interfacing Dynamixel with the PC is called a USB-to-Dynamixel adapter. This is a USB-to-serial adapter that can convert USB to RS232, RS 484, and TTL. In AX-12 motors, data communication is done using TTL. From the previous figure, we can see that there are three pins in each port. The data pin is used to send to and receive from AX-12, and power pins are used to power the servo. The input voltage range of the AX-12A Dynamixel is from 9V to 12V. The second port in each Dynamixel can be used for daisy chaining. We can connect up to 254 servos using such chaining.



Official links of the AX-12A servo and USB-to-Dynamixel adapter: AX-12A: <http://www.trossenrobotics.com/dynamixel-ax-12-robot-actuator.aspx> USB-to-Dynamixel: <http://www.trossenrobotics.com/robots-bioloid-usb2dynamixel.aspx>

To work with Dynamixel, we should know some more things. Let's have a look at some of the important specifications of the AX-12A servo. The specifications are taken from the servo manual.

| | |
|--------------------------|--|
| ▪ Weight : | 54.6g (AX-12A) |
| ▪ Dimension : | 32mm * 50mm * 40mm |
| ▪ Resolution : | 0.29° |
| ▪ Gear Reduction Ratio : | 254 : 1 |
| ▪ Stall Torque : | 1.5N.m (at 12.0V, 1.5A) |
| ▪ No load speed : | 59rpm (at 12V) |
| ▪ Running Degree : | 0° ~ 300°, Endless Turn |
| ▪ Running Temperature : | -5°C ~ +70°C |
| ▪ Voltage : | 9 ~ 12V (Recommended Voltage 11.1V) |
| ▪ Command Signal : | Digital Packet |
| ▪ Protocol Type : | Half duplex Asynchronous Serial Communication (8bit,1stop,No Parity) |
| ▪ Link (Physical) : | TTL Level Multi Drop (daisy chain type Connector) |
| ▪ ID : | 254 ID (0~253) |
| ▪ Communication Speed : | 7343bps ~ 1 Mbps |
| ▪ Feedback : | Position, Temperature, Load, Input Voltage, etc. |
| ▪ Material : | Engineering Plastic |

Figure 8: AX-12A specifications

The Dynamixel servo can communicate with the PC at a maximum speed of 1 Mbps. It can also provide feedback about various parameters, such as its position, temperature, and current load. Unlike RC servos, this can rotate up to 300 degrees, and communication is mainly done using digital packets.

Powering and connecting the Dynamixel to a PC

Now, we are going to connect the Dynamixel to a PC. The following is a standard way of doing that:



Figure 9: Connecting the Dynamixel to a PC

The three-pin cable is first connected to any of the ports of the AX-12, and the other side has to connect to a six-port power hub. From the six-port power hub, connect another cable to the USB-to-Dynamixel. We have to set the switch of the USB-to-Dynamixel to TTL mode. The power can be either be connected through a 12V adapter or through a battery. The 12V adapter has a 2.1 x 5.5 female barrel jack, so you should check the specifications of the male adapter plug while purchasing.

Setting up the USB-to-Dynamixel driver on the PC

We have already discussed that the USB-to-Dynamixel adapter is a USB-to-serial converter with an FTDI chip (<http://www.ftdichip.com/>) on it. We have to install a proper FTDI driver on the PC in order to detect the device. The driver is required for Windows but not for Linux, because FTDI drivers are already present in the Linux kernel. If you install the RoboPlus software, the driver may already be installed along with it. If it is not, you can manually install from the RoboPlus installation folder.

Plug the USB-to-Dynamixel into the Windows PC, and check Device Manager. (Right-click on My Computer and go to Properties | Device Manager). If the device is properly detected, you'll see something like this:

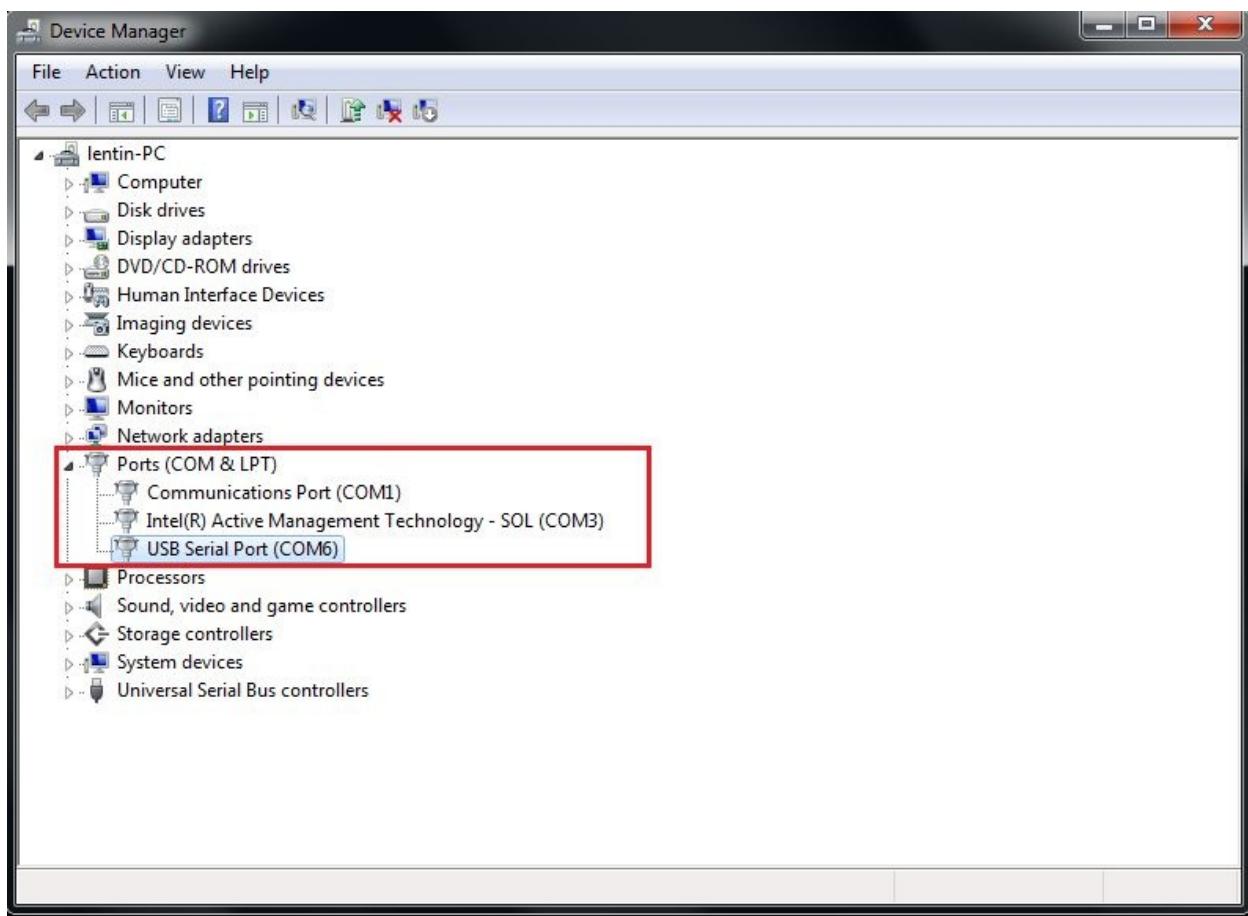


Figure 10: COM port of the USB-to-Dynamixel

If you are getting a COM port for the USB-to-Dynamixel, you can start Dynamixel manager from RoboPlus. You can connect to the serial port number from the list and click on the Search button to scan for Dynamixel, as shown in the next screenshot.

Select the COM port from the list, and connect to the port marked **1**. After connecting to the COM port, set the default baud rate to 1 Mbps, and click on the Start searching button:

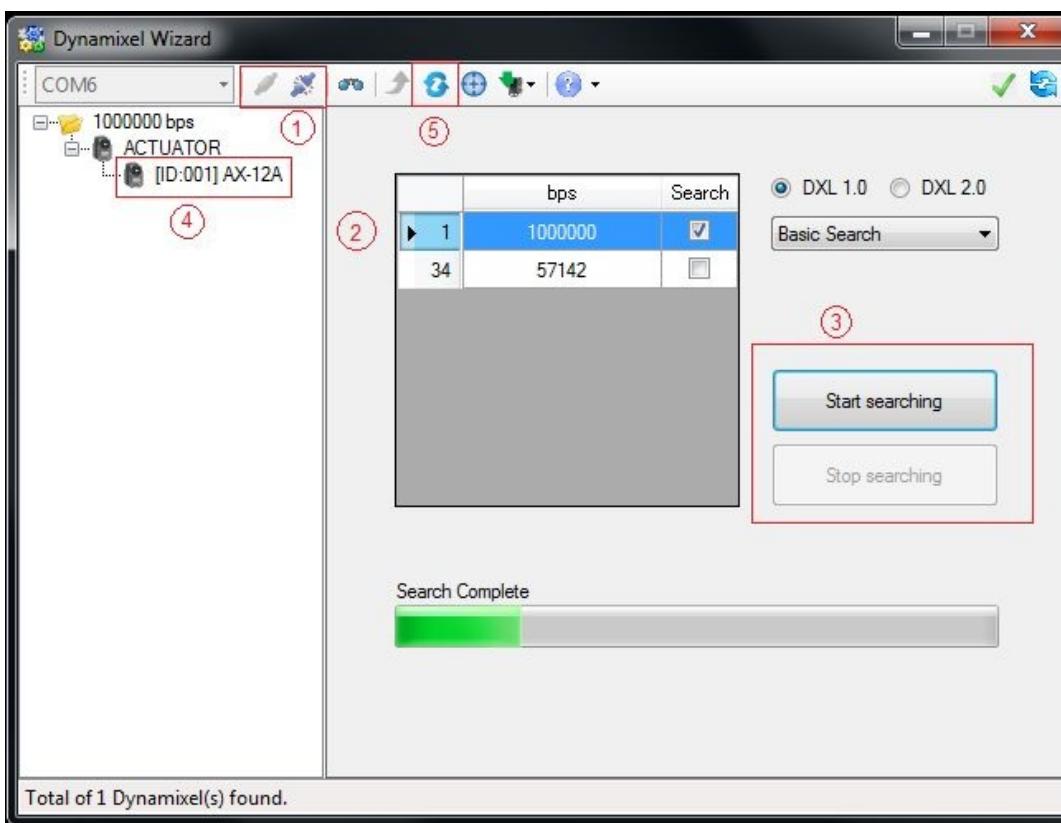


Figure 11: COM Port of the USB-to-Dynamixel

If you are getting a list of servos in the left-hand side panel, it means that your PC has detected a Dynamixel servo. If the servo is not being detected, you can perform the following steps to debug:

1. Make sure that the supply and connections are proper using a multimeter. Make sure that the servo LED on the back is blinking when power is on; if it is not coming on, it can indicate a problem with the servo or power supply.
2. Upgrade the firmware of the servo using Dynamixel manager from the option marked 5. The wizard is shown in the next set of screenshots. While using the wizard, you may need to power off the supply and turn it back on in order to detect the servo.
3. After detecting the servo, you have to select the servo model and install the new firmware. This may help you detect the servo in Dynamixel manager if the existing servo firmware is outdated.

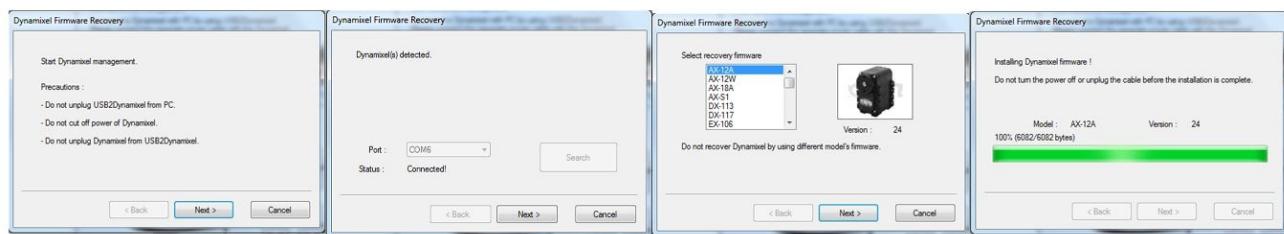


Figure 12: The Dynamixel recovery wizard

If the servos are being listed in Dynamixel Manager, click on one, and you can see its complete configuration. We have to modify some values inside the configurations for our current face-tracker project. Here are the parameters:

- ID: Set the ID to 1

- Baud rate: 1
- Moving Speed: 100
- Goal Position: 512

The modified servo settings are shown in the following figure:

| Addr | Description | Value |
|------|--------------------------------------|-------|
| 0 | Model Number | 12 |
| 2 | Version of Firmware | 24 |
| 3 | ID | 1 |
| 4 | Baud Rate | 1 |
| 5 | Return Delay Time | 250 |
| 6 | CW Angle Limit (Joint / Wheel Mode) | 0 |
| 8 | CCW Angle Limit (Joint / Wheel Mode) | 1023 |
| 11 | The Highest Limit Temperature | 70 |
| 12 | The Lowest Limit Voltage | 60 |
| 13 | The Highest Limit Voltage | 140 |
| 14 | Max Torque | 1023 |
| 16 | Status Return Level | 2 |
| 17 | Alarm LED | 0 |
| 18 | Alarm Shutdown | 37 |
| 24 | Torque Enable | 1 |
| 25 | LED | 0 |
| 26 | CW Compliance Margin | 1 |
| 27 | CCW Compliance Margin | 1 |
| 28 | CW Compliance Slope | 32 |
| 29 | CCW Compliance Slope | 32 |
| 30 | Goal Position | 512 |
| 32 | Moving Speed | 83 |
| 34 | Torque Limit | 1023 |
| 36 | Present Position | 511 |
| 38 | Present Speed | 1023 |

Figure 13: Modified Dynamixel firmware settings

After making these settings, you can check whether the servo is working well or not by changing its Goal position.

Nice! You are done configuring Dynamixel; congratulations! What next? We'll want to interface Dynamixel with ROS.



*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/ros_robotics_projects*

Interfacing Dynamixel with ROS

If you successfully configured the Dynamixel servo, then it will be very easy to interface Dynamixel with ROS running on Ubuntu. As we've already discussed, there is no need of an FTDI driver in Ubuntu because it's already built into the kernel. The only thing we have to do is install the ROS Dynamixel driver packages.

The ROS Dynamixel packages are available at the following link:

http://wiki.ros.org/dynamixel_motor

You can install the Dynamixel ROS packages using commands we'll look at now.

Installing the ROS dynamixel_motor packages

The ROS `dynamixel_motor` package stack is a dependency for the face tracker project, so we can install it to the `ros_project_dependencies_ws` ROS workspace.

Open a Terminal and switch to the `src` folder of the workspace:

```
| $ cd ~/ros_project_dependencies_ws/src
```

Clone the latest Dynamixel driver packages from GitHub:

```
| $ git clone https://github.com/arebgun/dynamixel_motor
```

Remember to do a `catkin_make` to build the entire packages of the Dynamixel driver.

If you can build the workspace without any errors, you are done with meeting the dependencies of this project.

Congratulations! You are done with the installation of the Dynamixel driver packages in ROS. We have now met all the dependencies required for the face tracker project.

So let's start working on face tracking project packages.

Creating face tracker ROS packages

Let's start creating a new workspace for keeping the entire ROS project files for this section. You can name the workspace `ros_robots_projects_ws`.

Download or clone the source code of the section from GitHub using the following link.

```
| $ git clone https://github.com/qboticslabs/ros_robots_projects
```

Now, you can copy two packages named `face_tracker_pkg` and `face_tracker_control` from the `chapter_2_codes` folder into the `src` folder of `ros_robots_projects_ws`.

Do a `catkin_make` to build the two project packages!

Yes, you have set up the face tracker packages on your system, but what if you want to create your own package for tracking? First, delete the current packages that you copied to the `src` folder, and use the following commands to create the packages.



Note that you should be in the `src` folder of `ros_robots_projects_ws` while creating the new packages, and there should not be any existing packages from the section's GitHub code.

Switch to the `src` folder:

```
| $ cd ~/ros_robots_projects_ws/src
```

The next command will create the `face_tracker_pkg` ROS package with the main dependencies, such as `cv_bridge`, `image_transport`, `sensor_msgs`, `message_generation`, and `message_runtime`.

We are including these packages because these packages are required for the proper working of the face tracker package. The face tracker package contain ROS nodes for detecting faces and determining the centroid of the face:

```
| $ catkin_create_pkg face_tracker_pkg roscpp rospy cv_bridge    image_transport sensor_msgs std_msgs  
|   message_runtime   message_generation
```

Next, we need to create the `face_tracker_control` ROS package. The important dependency of this package is `dynamixel_controllers`. This package is used to subscribe to the centroid from the face tracker node and control the Dynamixel in a way that the face centroid will always be in the center portion of the image:

```
| $ catkin_create_pkg face_tracker_pkg roscpp rospy std_msgs    dynamixel_controllers message_generation
```

Okay, you have created the ROS packages on your own. What's next? Before starting to code, you may have to understand some concepts of OpenCV and its interface with ROS. Also, you have to know how to publish ROS image messages. So let's master the concepts first.

The interface between ROS and OpenCV

Open Source Computer Vision (OpenCV) is a library that has APIs to perform computer vision applications. The project was started in Intel Russia, and later on, it was supported by Willow Garage and Itseez. In 2016, Itseez was acquired by Intel.



OpenCV website: <http://opencv.org/> Willow Garage: <http://www.willowgarage.com/> Itseez: <http://itseez.com>

OpenCV is a cross-platform library that supports most operating systems. Now, it also has an open source BSD license, so we can use it for research and commercial applications. The OpenCV version interfaced with ROS Kinetic is 3.1. The 3.x versions of OpenCV have a few changes to the APIs from the 2.x versions.

The OpenCV library is integrated into ROS through a package called `vision_opencv`.

The `vision_opencv` metapackage has two packages:

- `cv_bridge`: This package is responsible for converting the OpenCV image data type (`cv::Mat`) into ROS `Image` messages (`sensor_msgs/Image.msg`).
- `image_geometry`: This package helps us interpret images geometrically. This node will aid in processing such as camera calibration and image rectification.

Out of these two packages, we are mainly dealing with `cv_bridge`. Using `cv_bridge`, the face tracker node can convert ROS `Image` messages from `usb_cam` to the OpenCV equivalent, `cv::Mat`. After converting to `cv::Mat`, we can use OpenCV APIs to process the camera image.

Here is a block diagram that shows the role of `cv_bridge` in this project:

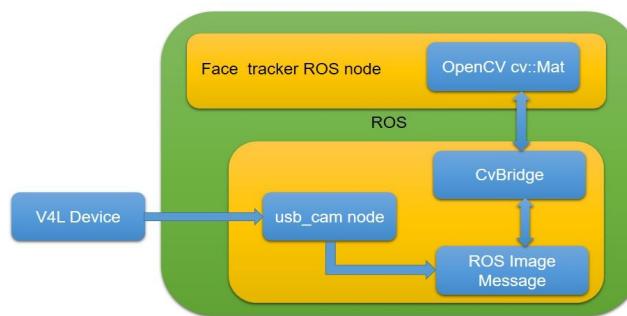


Figure 14: The role of `cv_bridge`

Here, `cv_bridge` is working between the `usb_cam` node and face-tracking node. We'll learn more about the face-tracking node in the next section. Before that, it will be good if you get an idea of its working.

Another package we are using to transport ROS `Image` messages between two ROS nodes is `image_transport` (http://wiki.ros.org/image_transport). This package is always used to subscribe to and publish image data in ROS.

The package can help us transport images in low bandwidth by applying compression techniques. This package is also installed along with the full ROS desktop installation.

That's all about OpenCV and the ROS interface. In the next section, we are going to work with the first package of this project: `face_tracker_pkg`.



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/ros_robots_projects`

Working with the face-tracking ROS package

We have already created or copied the `face_tracker_pkg` package to the workspace and have discussed some of its important dependencies. Now, we are going to discuss what this package exactly does!

This package consists of a ROS node called `face_tracker_node` that can track faces using OpenCV APIs and publish the centroid of the face to a topic. Here is the block diagram of the working of `face_tracker_node`:

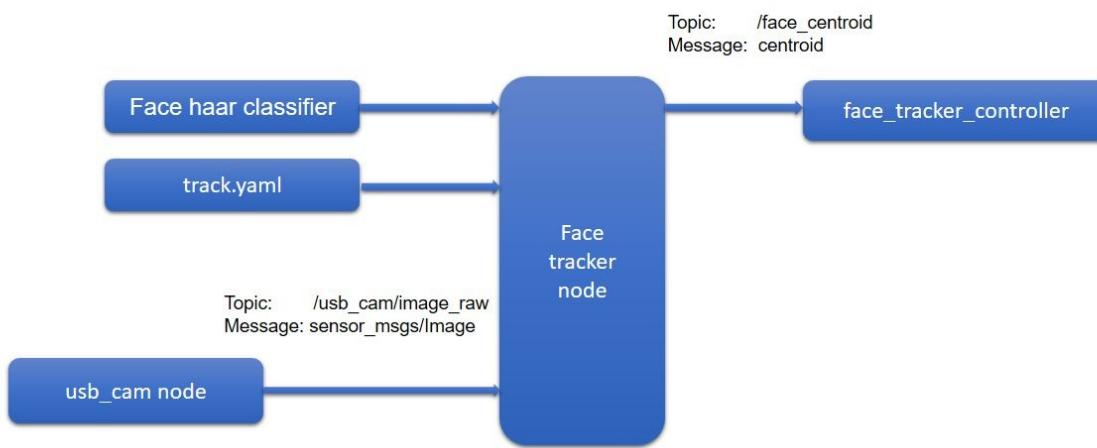


Figure 15: Block diagram of `face_tracker_node`

Let's discuss the things connected to `face_tracker_node`. One of the sections that may be unfamiliar to you is the face Haar classifier:

- **Face Haar classifier:** The Haar feature-based cascade classifier is a machine learning approach for detecting objects. This method was proposed by Paul Viola and Michael Jones in their paper *Rapid Object detection using a boosted cascade of simple features* in 2001. In this method, a cascade file is trained using a positive and negative sample image, and after training, that file is used for object detection.
 - In our case, we are using a trained Haar classifier file along with OpenCV source code. You will get these Haar classifier files from the OpenCV `data` folder (<https://github.com/opencv/opencv/tree/master/data>). You can replace the desired Haar file according to your application. Here, we are using the face classifier. The classifier will be an XML file that has tags containing features of a face. Once the features inside the XML match, we can retrieve the **region of interest (ROI)** of the face from the image using the OpenCV APIs. You can check the Haar classifier of this project from `face_tracker_pkg/data/face.xml`.
- `track.yaml`: This is a ROS parameter file having parameters such as the Haar file path, input image topic, output image topic, and flags to enable and disable face tracking. We are using ROS configuration files because we can change the node parameters without modifying the face tracker source code. You can get this file from `face_tracker_pkg/config/track.yaml`.
- `usb_cam` node: The `usb_cam` package has a node publishing the image stream from the camera to ROS `Image` messages. The `usb_cam` node publishes camera images to the `/usb_cam/raw_image` topic, and this topic is subscribed to by the face tracker node for face detection. We can change the input topic in the

`track.yaml` file if we require.

- `face_tracker_control`: This is the second package we are going to discuss. The `face_tracker_pkg` package can detect faces and find the centroid of the face in the image. The centroid message contains two values, X and Y . We are using a custom message definition to send the centroid values. These centroid values are subscribed by the controller node and move the Dynamixel to track the face. The Dynamixel is controlled by this node.

Here is the file structure of `face_tracker_pkg`:

```
.
├── CMakeLists.txt
└── config
    └── track.yaml
├── data
    └── face.xml
├── include
    └── face_tracker_pkg
├── launch
    ├── start_dynamixel_tracking.launch
    ├── start_tracking.launch
    └── start_usb_cam.launch
├── msg
    └── centroid.msg
├── package.xml
└── src
    └── face_tracker_node.cpp

7 directories, 9 files
```

Figure 16: File structure of `face_tracker_pkg`

Let's see how the face-tracking code works. You can open the CPP file at `face_tracker_pkg/src/face_tracker_node.cpp`. This code performs the face detection and sends the centroid value to a topic.

We'll look at, and understand, some code snippets.

Understanding the face tracker code

Let's start with the header file. The ROS header files we are using in the code lie here. We have to include `ros/ros.h` in every ROS C++ node; otherwise, the source code will not compile. The remaining three headers are image-transport headers, which have functions to publish and subscribe to image messages in low bandwidth. The `cv_bridge` header has functions to convert between OpenCV ROS data types. The `image_encoding.h` header has the image-encoding format used during ROS-OpenCV conversions:

```
#include <ros/ros.h>
#include <image_transport/image_transport.h>
#include <cv_bridge/cv_bridge.h>
#include <sensor_msgs/image_encodings.h>
```

The next set of headers is for OpenCV. The `imgproc` header consists of image-processing functions, `highgui` has GUI-related functions, and `objdetect.hpp` has APIs for object detection, such as the Haar classifier:

```
#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/highgui/highgui.hpp>
#include "opencv2/objdetect.hpp"
```

The last header file is for accessing a custom message called **centroid**. The `centroid` message definition has two fields, `int32 x` and `int32 y`. This can hold the centroid of the file. You can check this message definition from the `face_tracker_pkg/msg/centroid.msg` folder:

```
#include <face_tracker_pkg/centroid.h>
```

The following lines of code give a name to the raw image window and face-detection window:

```
static const std::string OPENCV_WINDOW = "raw_image_window";
static const std::string OPENCV_WINDOW_1 = "face_detector";
```

The following lines of code create a C++ class for our face detector. The code snippet creates handles of `NodeHandle`, which is a mandatory handle for a ROS node`image_transport`, which helps send ROS `Image` messages across the ROS computing graph; and a publisher for the face centroid, which can publish the centroid values using the `centroid.msg` file defined by us. The remaining definitions are for handling parameter values from the parameter file `track.yaml`:

```
class Face_Detector
{
    ros::NodeHandle nh_;
    image_transport::ImageTransport it_;
    image_transport::Subscriber image_sub_;
    image_transport::Publisher image_pub_;
    ros::Publisher face_centroid_pub;
    face_tracker_pkg::centroid face_centroid;
    string input_image_topic, output_image_topic, haar_file_face;
    int face_tracking, display_original_image, display_tracking_image,
        center_offset, screenmaxx;
```

The following is the code for retrieving ROS parameters inside the `track.yaml` file. The advantage of using ROS parameters is that we can avoid hard-coding these values inside the program and modify the values without recompiling the code:

```
try{
nh_.getParam("image_input_topic", input_image_topic);
nh_.getParam("face_detected_image_topic", output_image_topic);
nh_.getParam("haar_file_face", haar_file_face);
nh_.getParam("face_tracking", face_tracking);
nh_.getParam("display_original_image", display_original_image);
nh_.getParam("display_tracking_image", display_tracking_image);
nh_.getParam("center_offset", center_offset);
nh_.getParam("screenmaxx", screenmaxx);

ROS_INFO("Successfully Loaded tracking parameters");
}
```

The following code creates a subscriber for the input image topic and publisher for the face-detected image. Whenever an image arrives on the input image topic, it will call a function called `imageCb`. The names of the topics are retrieved from ROS parameters. We create another publisher for publishing the centroid value, which is the last line of the code snippet:

```
image_sub_ = it_.subscribe(input_image_topic, 1,
&Face_Detector::imageCb, this);
image_pub_ = it_.advertise(output_image_topic, 1);

face_centroid_pub = nh_.advertise<face_tracker_pkg::centroid>
("/face_centroid",10);
```

The next bit of code is the definition of `imageCb`, which is a callback for `input_image_topic`. What it basically does is it converts the `sensor_msgs/Image` data into the `cv::Mat` OpenCV data type. The `cv_bridge::CvImagePtr` `cv_ptr` buffer is allocated for storing the OpenCV image after performing the ROS-OpenCV conversion using the `cv_bridge::toCvCopy` function:

```
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);
    }
}
```

We have already discussed the Haar classifier; here is the code to load the Haar classifier file:

```
string cascadeName = haar_file_face;
CascadeClassifier cascade;
if( !cascade.load( cascadeName ) )
{
    cerr << "ERROR: Could not load classifier cascade" << endl;
}
```

We are now moving to the core part of the program, which is the detection of the face performed on the converted OpenCV image data type from the ROS `Image` message. The following is the function call of `detectAndDraw()`, which is performing the face detection, and in the last line, you can see the output image topic being published. Using `cv_ptr->image`, we can retrieve the `cv::Mat` data type, and in the next line, `cv_ptr->toImageMsg()` can convert this into a ROS `Image` message. The arguments of the `detectAndDraw()` function are the OpenCV image and cascade variables:

```
detectAndDraw( cv_ptr->image, cascade );
image_pub_.publish(cv_ptr->toImageMsg());
```

Let's understand the `detectAndDraw()` function, which is adapted from the OpenCV sample code for face detection. The function arguments are the input image and cascade object. The next bit of code will convert the image into grayscale first and equalize the histogram using OpenCV APIs. This is a kind of preprocessing before detecting the face from the image. The `cascade.detectMultiScale()` function is used for this purpose (http://docs.opencv.org/2.4/modules/objdetect/doc/cascade_classification.html).

```
Mat gray, smallImg;
cvtColor( img, gray, COLOR_BGR2GRAY );
double fx = 1 / scale ;
resize( gray, smallImg, Size(), fx, fx, INTER_LINEAR );
equalizeHist( smallImg, smallImg );
t = (double)cvGetTickCount();
cascade.detectMultiScale( smallImg, faces,
    1.1, 15, 0
    |CASCADE_SCALE_IMAGE,
    Size(30, 30) );
```

The following loop will iterate on each face that is detected using the `detectMultiScale()` function. For each face, it finds the centroid and publishes to the `/face_centroid` topic:

```
for ( size_t i = 0; i < faces.size(); i++ )
{
    Rect r = faces[i];
    Mat smallImgROI;
    vector<Rect> nestedObjects;
    Point center;
    Scalar color = colors[i%8];
    int radius;

    double aspect_ratio = (double)r.width/r.height;
    if( 0.75 < aspect_ratio && aspect_ratio < 1.3 )
    {
        center.x = cvRound((r.x + r.width*0.5)*scale);
        center.y = cvRound((r.y + r.height*0.5)*scale);
        radius = cvRound((r.width + r.height)*0.25*scale);
        circle( img, center, radius, color, 3, 8, 0 );

        face_centroid.x = center.x;
        face_centroid.y = center.y;

        //Publishing centroid of detected face
        face_centroid_pub.publish(face_centroid);
    }
}
```

To make the output image window more interactive, there are text and lines to alert about the user's face on the left or right or at the center. This last section of code is mainly for that purpose. It uses OpenCV APIs to do this job. Here is the code to display text such as Left, Right, and Center on the screen:

```
putText(img, "Left", cvPoint(50,240),
FONT_HERSHEY_SIMPLEX, 1,
cvScalar(255,0,0), 2, CV_AA);
putText(img, "Center", cvPoint(280,240),
FONT_HERSHEY_SIMPLEX,
1, cvScalar(0,0,255), 2, CV_AA);
putText(img, "Right", cvPoint(480,240),
FONT_HERSHEY_SIMPLEX,
1, cvScalar(255,0,0), 2, CV_AA);
```

Excellent! We're done with the tracker code; let's see how to build it and make it executable.

Understanding CMakeLists.txt

The default `CMakeLists.txt` file made during the creation of the package has to be edited in order to compile the previous source code. Here is the `CMakeLists.txt` file used to build the `face_tracker_node.cpp` class.

The first two lines state the minimum version of `cmake` required to build this package, and next line is the package name:

```
| cmake_minimum_required(VERSION 2.8.3)
| project(face_tracker_pkg)
```

The following line searches for the dependent packages of `face_tracker_pkg` and raises an error if it is not found:

```
| find_package(catkin REQUIRED COMPONENTS
|   cv_bridge
|   image_transport
|   roscpp
|   rospy
|   sensor_msgs
|   std_msgs
|   message_generation
|
| )
```

This line of code contains the system-level dependencies for building the package:

```
| find_package(Boost REQUIRED COMPONENTS system)
```

As we've already seen, we are using a custom message definition called `centroid.msg`, which contains two fields, `int32 x` and `int32 y`. To build and generate C++ equivalent headers, we should use the following lines:

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

The `catkin_package()` function is a `catkin`-provided CMake macro that is required to generate `pkg-config` and CMake files.

```
| catkin_package(
|   CATKIN_DEPENDS roscpp rospy std_msgs message_runtime
| )
| include_directories(
|   ${catkin_INCLUDE_DIRS}
| )
```

Here, we are creating the executable called `face_tracker_node` and linking it to `catkin` and OpenCV libraries:

```
| add_executable(face_tracker_node src/face_tracker_node.cpp)
```

```
target_link_libraries(face_tracker_node
    ${catkin_LIBRARIES}
    ${OpenCV_LIBRARIES}
)
```

The track.yaml file

As we discussed, the `track.yaml` file contains ROS parameters, which are required by the `face_tracker_node`. Here are the contents of `track.yaml`:

```
image_input_topic: "/usb_cam/image_raw"
face_detected_image_topic: "/face_detector/raw_image"
haar_file_face:
"/home/robot/ros_robotics_projects_ws/
src/face_tracker_pkg/data/face.xml"
face_tracking: 1
display_original_image: 1
display_tracking_image: 1
```

You can change all the parameters according to your needs. Especially, you may need to change `haar_file_face`, which is the path of the Haar face file. If we set `face_tracking: 1`, it will enable face tracking, otherwise not. Also, if you want to display the original and face-tracking image, you can set the flag here.

The launch files

The launch files in ROS can do multiple tasks in a single file. The launch files have an extension of `.launch`. The following code shows the definition of `start_usb_cam.launch`, which starts the `usb_cam` node for publishing the camera image as a ROS topic:

```
<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="auto_focus" value="false" />
    <param name="io_method" value="mmap"/>
  </node>
</launch>
```

Within the `<node>...</node>` tags, there are camera parameters that can be change by the user. For example, if you have multiple cameras, you can change the `video_device` value from `/dev/video0` to `/dev/video1` to get the second camera's frames.

The next important launch file is `start_tracking.launch`, which will launch the face-tracker node. Here is the definition of this launch file:

```
<launch>
<!-- Launching USB CAM launch files and Dynamixel controllers -->
  <include file="$(find
face_tracker_pkg)/launch/start_usb_cam.launch"/>

<!-- Starting face tracker node -->
  <rosparam file="$(find face_tracker_pkg)/config/track.yaml"
command="load"/>

  <node name="face_tracker" pkg="face_tracker_pkg"
type="face_tracker_node" output="screen" />
</launch>
```

It will first start the `start_usb_cam.launch` file in order to get ROS image topics, then load `track.yaml` to get necessary ROS parameters, and then load `face_tracker_node` to start tracking.

The final launch file is `start_dynamixel_tracking.launch`; this is the launch file we have to execute for tracking and Dynamixel control. We will discuss this launch file at the end of the chapter after discussing the `face_tracker_control` package.

Running the face tracker node

Let's launch the `start_tracking.launch` file from `face_tracker_pkg` using the following command. Note that you should connect your webcam to your PC:

```
$ roslaunch face_tracker_pkg start_tracking.launch
```

If everything works fine, you will get output like the following; the first one is the original image, and the second one is the face-detected image:

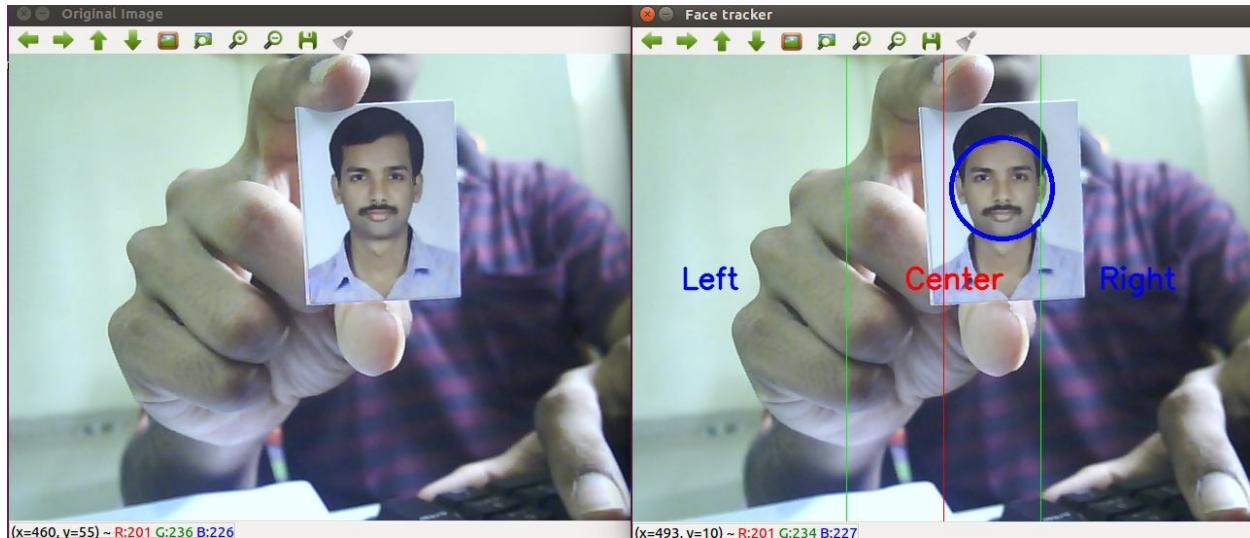


Figure 17: Face-detected image

We have not enabled Dynamixel now; this node will just find the face and publish the centroid values to a topic called `/face_centroid`.

So the first part of the project is done-what is next? It's the control part, right? Yes, so next, we are going to discuss the second package, `face_tracker_control`.

The face_tracker_control package

The `face_tracker_control` package is the control package used to track the face using the AX-12A Dynamixel servo.

Given here is the file structure of the `face_tracker_control` package:

```
.
├── CMakeLists.txt
└── config
    ├── pan.yaml
    └── servo_param.yaml
├── include
    └── face_tracker_control
├── launch
    ├── start_dynamixel.launch
    └── start_pan_controller.launch
├── msg
    └── centroid.msg
├── package.xml
└── src
    └── face_tracker_controller.cpp

6 directories, 8 files
```

Figure 18: File organization in the `face_tracker_control` package

We'll look at the use of each of these files first.

The start_dynamixel launch file

The `start_dynamixel` launch file starts Dynamixel Control Manager, which can establish a connection to a USB-to-Dynamixel adapter and Dynamixel servos. Here is the definition of this launch file:

```
<!-- This will open USB To Dynamixel controller and search for
servos -->
<launch>
    <node name="dynamixel_manager" pkg="dynamixel_controllers"
        type="controller_manager.py" required="true"
        output="screen">
        <rosparam>
            namespace: dxl_manager
            serial_ports:
                pan_port:
                    port_name: "/dev/ttyUSB0"
                    baud_rate: 1000000
                    min_motor_id: 1
                    max_motor_id: 25
                    update_rate: 20
        </rosparam>
    </node>
<!-- This will launch the Dynamixel pan controller -->
<include file="$(find
face_tracker_control)/launch/start_pan_controller.launch"/>
</launch>
```

We have to mention the `port_name` (you can get the port number from kernel logs using the `dmesg` command). The `baud_rate` we configured was 1 Mbps, and the motor ID was 1. The `controller_manager.py` file will scan from servo ID 1 to 25 and report any servos being detected.

After detecting the servo, it will start the `start_pan_controller.launch` file, which will attach a ROS joint position controller for each servo.

The pan controller launch file

As we can see from the previous subsection, the pan controller launch file is the trigger for attaching the ROS controller to the detected servos. Here is the definition for the `start_pan_controller.launch` file:

This will start the pan joint controller:

```
<launch>
    <!-- Start tilt joint controller -->
    <rosparam file="$(find face_tracker_control)/config/pan.yaml"
    command="load"/>
    <rosparam file="$(find
face_tracker_control)/config/servo_param.yaml" command="load"/>

    <node name="tilt_controller_spawner"
    pkg="dynamixel_controllers" type="controller_spawner.py"
        args="--manager=dxl_manager
              --port pan_port
              pan_controller"
        output="screen"/>
</launch>
```

The `controller_spawner.py` node can spawn a controller for each detected servo. The parameters of the controllers and servos are included in `pan.yaml` and `servo_param.yaml`.

The pan controller configuration file

The pan controller configuration file contains the configuration of the controller that the controller spawner node is going to create. Here is the `pan.yaml` file definition for our controller:

```
pan_controller:  
  controller:  
    package: dynamixel_controllers  
    module: joint_position_controller  
    type: JointPositionController  
    joint_name: pan_joint  
    joint_speed: 1.17  
    motor:  
      id: 1  
      init: 512  
      min: 316  
      max: 708
```

In this configuration file, we have to mention the servo details, such as ID, initial position, minimum and maximum servo limits, servo moving speed, and joint name. The name of the controller is `pan_controller`, and it's a joint position controller. We are writing one controller configuration for ID 1 because we are only using one servo.

The servo parameters configuration file

The `servo_param.yaml` file contains the configuration of the `pan_controller`, such as the limits of the controller and step distance of each movement; also, it has screen parameters such as the maximum resolution of the camera image and offset from the center of the image. The offset is used to define an area around the actual center of the image:

```
servomaxx: 0.5    #max degree servo horizontal (x) can turn
servomin: -0.5    # Min degree servo horizontal (x) can turn
screenmaxx: 640   #max screen horizontal (x)resolution
center_offset: 50 #offset pixels from actual center to right and
left
step_distancex: 0.01 #x servo rotation steps
```

The face tracker controller node

As we've already seen, the face tracker controller node is responsible for controlling the Dynamixel servo according to the face centroid position. Let's understand the code of this node, which is placed at `face_tracker_control/src/face_tracker_controller.cpp`.

The main ROS headers included in this code are as follows. Here, the `Float64` header is used to hold the position value message to the controller:

```
#include "ros/ros.h"
#include "std_msgs/Float64.h"
#include <iostream>
```

The following variables hold the parameter values from `servo_param.yaml`:

```
int servomaxx, servomin, screenmaxx, center_offset, center_left,
center_right;
float servo_step_distance, current_pos_x;
```

The following message headers of `std_msgs::Float64` are for holding the initial and current positions of the controller, respectively. The controller only accepts this message type:

```
std_msgs::Float64 initial_pose;
std_msgs::Float64 current_pose;
```

This is the publisher handler for publishing the position commands to the controller:

```
ros::Publisher dynamixel_control;
```

Switching to the `main()` function of the code, you can see following lines of code. The first line is the subscriber of `/face_centroid`, which has the centroid value, and when a value comes to the topic, it will call the `face_callback()` function:

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

The following line will initialize the publisher handle in which the values are going to be published through the `/pan_controller/command` topic:

```
dynamixel_control = node_obj.advertise<std_msgs::Float64>
("/pan_controller/command", 10);
```

The following code creates new limits around the actual center of image. This will be helpful for getting an approximated center point of the image:

```
center_left = (screenmaxx / 2) - center_offset;
center_right = (screenmaxx / 2) + center_offset;
```

Here is the callback function executed while receiving the centroid value coming through the `/face_centroid` topic. This callback also has the logic for moving the Dynamixel for each centroid value.

In the first section, the `x` value in the centroid is checking against `center_left`, and if it is in the left, it just

increments the servo controller position. It will publish the current value only if the current position is inside the limit. If it is in the limit, then it will publish the current position to the controller. The logic is the same for the right side: if the face is in the right side of the image, it will decrement the controller position.

When the camera reaches the center of image, it will pause there and do nothing, and that is the thing we want too. This loop is repeated, and we will get a continuous tracking:

```
void track_face(int x,int y)
{
    if (x < (center_left)){
        current_pos_x += servo_step_distance;
        current_pose.data = current_pos_x;
        if (current_pos_x < servomaxx and current_pos_x > servomin ){
            dynamixel_control.publish(current_pose);
        }
    }

    else if(x > center_right){
        current_pos_x -= servo_step_distance;
        current_pose.data = current_pos_x;
        if (current_pos_x < servomaxx and current_pos_x > servomin ){
            dynamixel_control.publish(current_pose);
        }
    }

    else if(x > center_left and x < center_right){
        ;
    }
}
```

Creating CMakeLists.txt

Like the first tracker package, there is no special difference in the control package; the difference is in the dependencies. Here, the main dependency is `dynamixel_controllers`. We are not using OpenCV in this package, so there's no need to include it:

```
cmake_minimum_required(VERSION 2.8.3)
project(face_tracker_control)
find_package(catkin REQUIRED COMPONENTS
    dynamixel_controllers
    roscpp
    rospy
    std_msgs
    message_generation
)
find_package(Boost REQUIRED COMPONENTS system)
add_message_files(
    FILES
    centroid.msg
)
## Generate added messages and services with any dependencies
## listed here
generate_messages(
    DEPENDENCIES
    std_msgs
)
catkin_package(
    CATKIN_DEPENDS dynamixel_controllers roscpp rospy std_msgs
)
include_directories(
    ${catkin_INCLUDE_DIRS}
)
add_executable(face_tracker_controller
src/face_tracker_controller.cpp)
target_link_libraries(face_tracker_controller ${catkin_LIBRARIES})
```



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/ros_robotics_projects*

Testing the face tracker control package

We have seen most of the files and their functionalities. So let's test this package first. We have to ensure that it is detecting the Dynamixel servo and creating the proper topic.

Before running the launch file, we may have to change the permission of the USB device, or it will throw an exception. The following command can be used to get permissions on the serial device:

```
|     $ sudo chmod 777 /dev/ttyUSB0
```

Note that you must replace `ttyUSB0` with your device name; you can retrieve it by looking at kernel logs. The `dmesg` command can help you find it.

Start the `start_dynamixel.launch` file using the following command:

```
$ roslaunch face_tracker_control start_dynamixel.launch
```

```
* /servomin: -0.5
* /step_distancex: 0.01

NODES
/
dynamixel_manager (dynamixel_controllers/controller_manager.py)
tilt_controller_spawner (dynamixel_controllers/controller_spawner.py)

auto-starting new master
process[master]: started with pid [6997]
ROS_MASTER_URI=http://localhost:11311

setting /run_id to 6b4d648e-62c8-11e6-ac5f-00177c2e2869
process[rosout-1]: started with pid [7010]
started core service [/rosout]
process[dynamixel_manager-2]: started with pid [7027]
process[tilt_controller_spawner-3]: started with pid [7028]
[INFO] [WallTime: 1471252362.231754] pan_port controller_spawner: waiting for controller_manager dxl_manager to startup in global namespace...
[INFO] [WallTime: 1471252362.661902] pan_port: Pinging motor IDs 1 through 25...
[INFO] [WallTime: 1471252364.696276] pan_port: Found 1 motors - 1 AX-12 [1], initialization complete.
[INFO] [WallTime: 1471252364.951534] pan_port controller_spawner: All services are up, spawning controllers...
[INFO] [WallTime: 1471252364.979589] Controller pan_controller successfully started.
[tilt_controller_spawner-3] process has finished cleanly
log file: /home/Robot/.ros/log/6b4d648e-62c8-11e6-ac5f-00177c2e2869/tilt_controller_spawner-3*.log
```

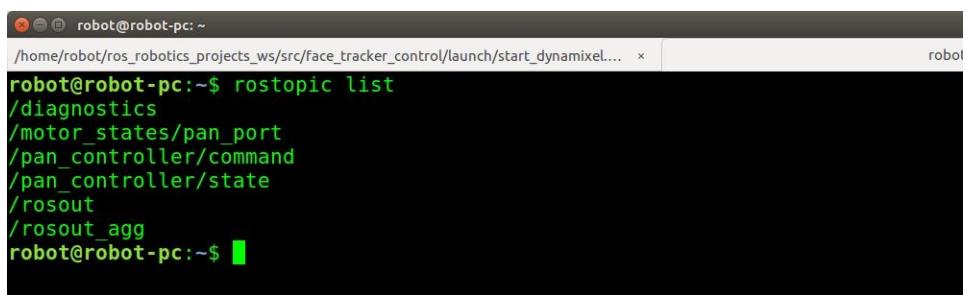
Figure 19: Finding Dynamixel servos and creating controllers

If everything is successful, you will get a message as shown in the previous figure.



If any errors occur during the launch, check the servo connection, power, and device permissions.

The following topics are generated when we run this launch file:



```
robot@robot-pc:~$ rostopic list
/diagnostics
/motor_states/pan_port
/pan_controller/command
/pan_controller/state
/rosout
/rosout_agg
robot@robot-pc:~$
```

Figure 20: Face tracker control topics

Bringing all the nodes together

Next, we'll look at the final launch file, which we skipped while covering the `face_tracker_pkg` package, and that is `start_dynamixel_tracking.launch`. This launch file starts both face detection and tracking using Dynamixel motors:

```
<launch>
<!-- Launching USB CAM launch files and Dynamixel controllers -->
<include file="$(find
  face_tracker_pkg)/launch/start_tracking.launch"/><include
  file="$(find
  face_tracker_control)/launch/start_dynamixel.launch"/>
<!-- Starting face tracker node -->

<node name="face_controller" pkg="face_tracker_control"
  type="face_tracker_controller" output="screen" />

</launch>
```

Fixing the bracket and setting up the circuit

Before doing the final run of the project, we have to do something on the hardware side. We have to fix the bracket to the servo horn and fix the camera to the bracket. The bracket should be connected in such a way that it is always perpendicular to the center of the servo. The camera is mounted on the bracket, and it should be pointed toward the center position.

The following image shows the setup I did for this project. I simply used tape to fix the camera to the bracket. You can use any additional material to fix the camera, but it should always be aligned to the center first:



Figure 21: Fixing camera and bracket to the AX-12A

If you are done with this, then you are ready to go for the final run of this project.

The final run

I hope that you have followed all instructions properly; here is the command to launch all the nodes for this project and start tracking using Dynamixel:

```
$ roslaunch face_tracker_pkg start_dynamixel_tracking.launch
```

You will get the following windows, and it would be good if you could use a photo to test the tracking, because you will get continuous tracking of the face:

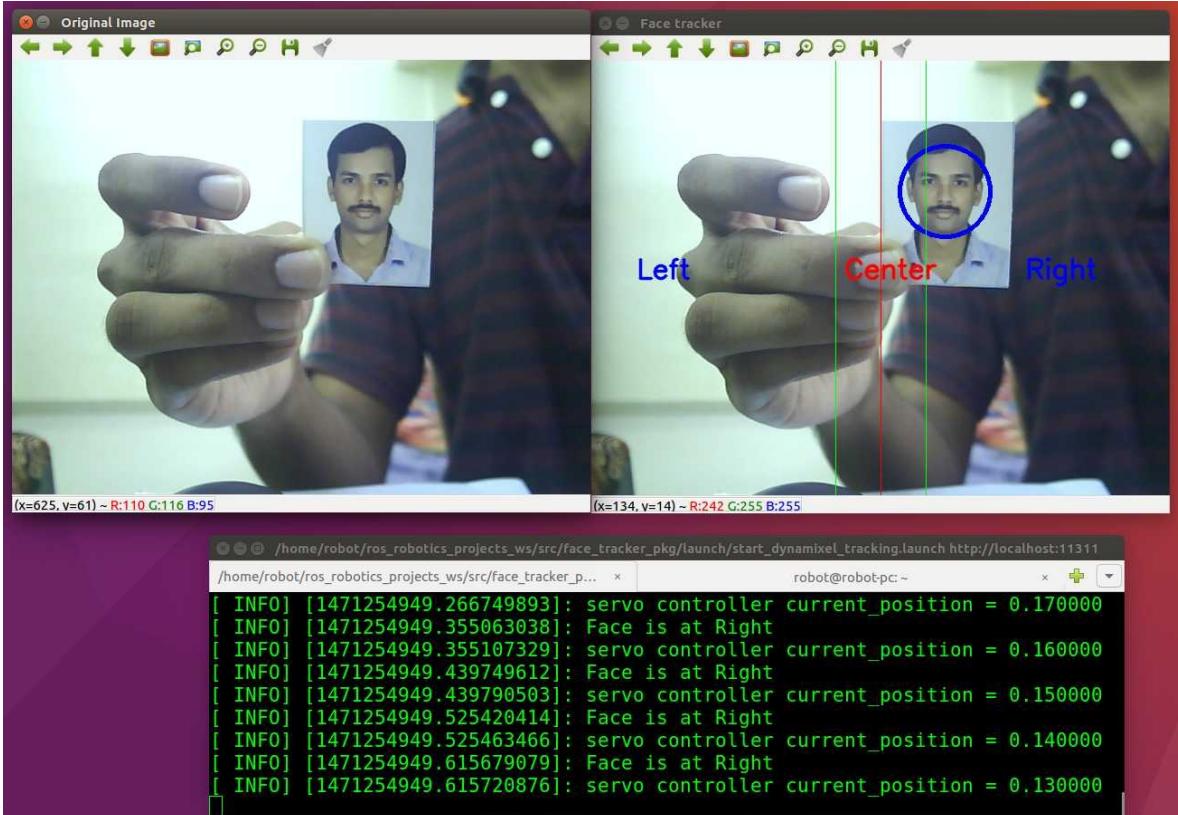


Figure 22: Final face tracking

Here, you can see the Terminal message that says the image is in the right and the controller is reducing the position value to achieve the center position.

Questions

- What is the main function of the `usb_cam` ROS package?
- What is the use of the `dynamixel_motor` package in ROS?
- What is the package for interfacing ROS and OpenCV?
- What is the difference between `face_tracker_pkg` and `face_tracker_control`?

Summary

This chapter was about building a face tracker using a webcam and Dynamixel motor. The software we used was ROS and OpenCV. Initially, we saw how to configure the webcam and Dynamixel motor, and after configuration, we were trying to build two packages for tracking. One package was for face detection, and the second package was a controller that can send a position command to Dynamixel to track the face. We have discussed the use of all the files inside the packages and did a final run to demonstrate the complete working of the system.

Building a Siri-Like Chatbot in ROS

Artificial intelligence, machine learning, and deep learning are getting very popular nowadays. All these technologies are linked, and the common goal is to mimic human intelligence. There are numerous applications for these fields; some of the relevant ones are as follows:

- **Logical reasoning:** This will generate logical conclusions from existing data. Reasoning using AI techniques is widely used in areas such as robotics, computer vision, and analytics.
- **Knowledge representation:** This is the study of how a computer could store knowledge fragments like our brains do. This is possible using AI techniques.
- **Planning:** This concept is heavily used in robotics; there are AI algorithms such as A* (star) and Dijkstra for planning a robot's path from its current position to a goal position. It is also heavily used in swarm robotics for robot planning.
- **Learning:** Humans can learn, right? What about machines? Using machine learning techniques, we can train artificial neural networks to learn data.
- **Natural language processing:** This is the ability to understand human language, mainly from text data.
- **Perception:** A robot can have various kinds of sensors, such as camera and mic. Using AI, we can analyze this sensor data and understand the meaning of it.
- **Social intelligence:** This is one of the trending fields of AI. Using AI, we can build social intelligence in a machine or robot. Robots such as Kismet and Jibo have social intelligence.

In this chapter, we will discuss knowledge representation and social intelligence. If you are going to build a robot that has skills to interact with people, you may need to store the knowledge and create some social skills. This chapter will teach you how to build a base system for such robots. Before discussing the implementation of this system, let's take a look at some social and service robots and its characteristics.



MIT Kismet: <http://www.ai.mit.edu/projects/humanoid-robotics-group/kismet/kismet.html> *Jibo:* <https://www.jibo.com/>

Social robots

In simple words, social robots are personal companions or assistive robots that can interact with human beings using speech, vision, and gestures. These robots behave like pets that can express emotions like us and can communicate their emotions using speech or gestures.

Nowadays, most social robots have an LCD display on their heads, actuators for movement, speakers and microphone for communication, and cameras for perception.

Here are some images of popular social robots:

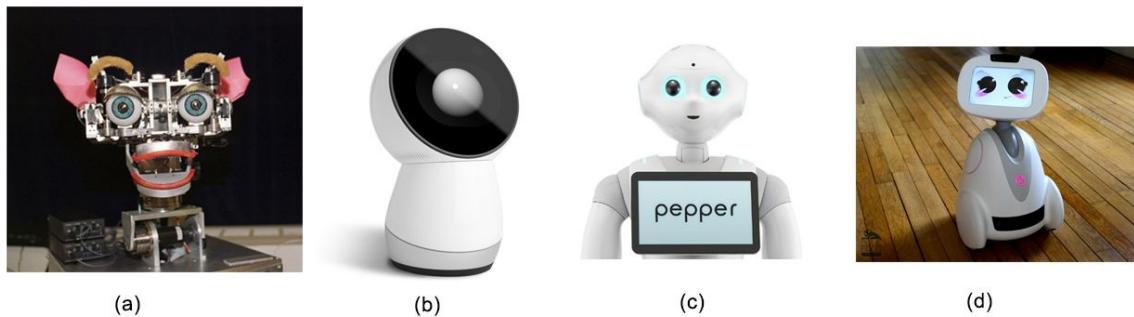


Figure 1: Famous social robots

Let's learn about them:

- **Kismet(a):** This is a social robot from MIT by Dr. Cynthia Breazeal and team, made in the 1990s. Kismet can identify people and objects and simulate different emotions. Kismet was just a research robot, not a commercial product.
- **Jibo(b):** Jibo was conceived by Dr. Cynthia and team in 2014. Jibo has a rotating head with a screen, and it can communicate with people using speech recognition and can recognize them using perception techniques.
- **Pepper(c):** Pepper is a humanoid social robot from Softbank. Unlike other social robots, this robot has two arms and a mobile base similar to a humanoid robot. Like other social robots, it can communicate with people and has tactile sensors on its body.
- **Buddy(d):** This robot buddy has similar characteristics to the previous robots. It has a mobile base for movement and a screen on the head to express emotions.



Pepper: <https://www.ald.softbankrobotics.com/en/cool-robots/pepper> *Buddy:* <http://www.bluefrogrobotics.com/en/home/>

These may have high intelligence and social skills. But most of the robots' source code is not open source, so we can't explore much about the software platforms and algorithms they use to implement them. But in this chapter, we are going to look at some of the open source solutions to build intelligence and social skills in robots.

Building social robots

A service or social robot may have capabilities to perceive the world using inbuilt cameras, interact with humans using speech and make decisions using artificial intelligence algorithms. These kinds of robots are a bit complicated in design, we can see a typical building block diagram of a social robot in the following figure.

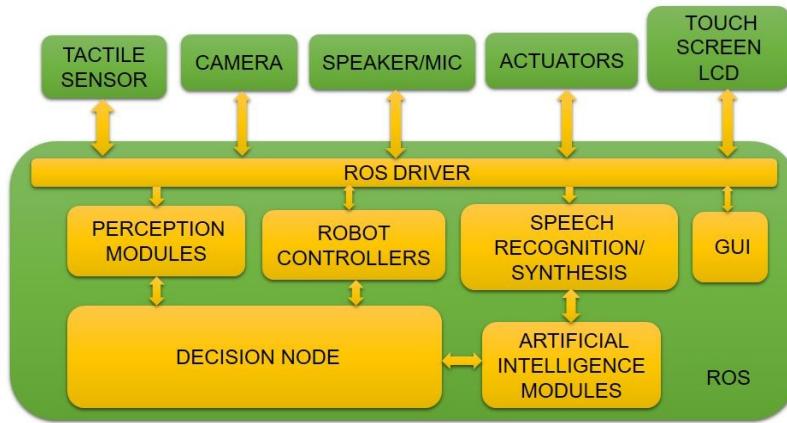


Figure 2: Block diagram of a typical social robot

The robot has sensors such as tactile sensor, camera, microphone, and touch screen and will have some actuators for its movement. The actuators will help the robot to move its head or body. There may be mobile service robots which has extra motors for navigation.

Inside the software block, you may find modules for perception which handle camera data and finding necessary objects from the scene, speech recognition/synthesis, artificial intelligence modules, robot controller modules for controlling the actuators, decision-making node which combine all data from sensors and makes the final decision on what to do next. The ROS driver layer help to interface all sensors, actuators to ROS and the GUI can be an interactive visualization in the LCD panel.

In this chapter, we are going to implement the speech recognition or synthesis block with artificial intelligence which can communicate with people using text and speech. The reply from the bot should be like a human's.

We are going to implement a simple AI Chatbot using **AIML (Artificial Intelligence Markup Language)** which can be integrated to a social robot.

Let's see how to make software for such an interactive robot, starting with the prerequisites to build the software.

Prerequisites

Here are the prerequisites for doing this project:

- Ubuntu 16.04 LTS
- Python 2.7
- PyAIML: AIML interpreter in Python
- ROS Kinetic
- The `sound_play` ROS package: text-to-speech package in ROS

Let's get start with AIML.

Getting started with AIML

AIML (Artificial Intelligence Markup Language) is an XML-based language to store segments of knowledge inside XML tags. AIML files help us store knowledge in a structured way so that we can easily access it whenever required.

AIML was developed by *Richard Wallace* and the free software community worldwide between 1995 and 2002. You may have heard about chatter bots such as **Artificial Linguistic Internet Computer Entity (A.L.I.C.E.)** and **ELIZA**. AIML is the base of these chatter bots. The dataset of the A.L.I.C.E. chatter bot is available under the GNU GPL license, and there are open source AIML interpreters available in C++, Python, Java, and Ruby. We can use these interpreters to feed our input to the knowledge base and retrieve the best possible reply from it.

AIML tags

There is a set of AIML tags to represent knowledge inside files. The following are some of the important tags and their uses:

- <aiml>: Each AIML file starts with this tag and ends with the </aiml> tag. Basically, it holds the version of AIML and character encoding of the file. The <aiml> tag is not mandatory, but it will be useful when handling a huge AIML dataset. Here is the basic usage of the <aiml> tag:

```
<aiml version="1.0.1" encoding="UTF-8">
</aiml>
```

- <category>: Each knowledge segment is kept under this tag. This tag holds the input pattern from the user and outputs a response for it. The possible input from the user is kept under the <pattern> tag, and the corresponding response is under the <template> tag. Here is an example of the category, pattern, and template tags:

```
<aiml version="1.0.1" encoding="UTF-8">
  <category>
    <pattern> WHAT IS YOUR NAME </pattern>
    <template> MY NAME IS ROBOT </template>
  </category>
</aiml>
```

When a user asks the robot, "What is your name?", the robot replies, "My name is Robot." This is how we store knowledge for the robot.

- <pattern>: This tag consists of user input. From the preceding code, we can see that WHAT IS YOUR NAME is the user input. There will only be one pattern inside a category, placed after the category tag. Inside a pattern, we can include wild cards such as * or -, which can replace a string in the corresponding position.
- <template>: The template tag consists of responses to user input. In the previous code, MY NAME IS ROBOT is the response.
- <star index = "n" />: This tag helps extract a word from a sentence. The n indicates which word of the sentence is to be extracted:

<star index= "1" />: This indicates the first fragment of the template sentence.

<star index= "2" />: This indicates the second fragment of the template sentence.

Using the star index, we can extract the word from user input and insert the word into the response if needed.

Here is an example of using wildcards and a start index:

```
<aiml version="1.0.1" encoding="UTF-8">
  <category>
    <pattern> MY NAME IS * </pattern>
    <template>
      NICE TO SEE YOU <star index="1"/>
```

```
</template>
</category>

<category>
<pattern> MEET OUR ROBOTS * AND * </pattern>
<template>
    NICE TO SEE <star index="1"/> AND <star
    index="2"/>.
</template>
</category>
</aiml>
```

Here, we can reuse the word that comes in the * position in the `<template>` tag. Consider this input:

```
You: MY NAME IS LENTIN
Robot: NICE TO SEE YOU LENTIN
```

In the second category, you will get the following reply from the robot for the given input:

```
You: MEET OUR ROBOTS ROBIN AND TURTLEBOT
Robot: NICE TO SEE ROBIN AND TURTLEBOT
```

These are the basic tags used inside AIML files. Next, we'll see how to load these files and retrieve an intelligent reply from the AIML knowledge base for a random input from the user.



The following link will give you the list of AIML tags: <http://www.alicebot.org/documentation/aiml-reference.html>

The PyAIML interpreter

There are AIML interpreters in many languages used to load the AIML knowledge base and interact with it. One of the easiest ways of loading and interacting with AIML files is using an AIML interpreter in Python called PyAIML. The PyAIML module can read all the categories, patterns, and templates and can build a tree. Using a backtracking depth-first search algorithm, it can search for the appropriate response from the user in order to give the proper reply.

PyAIML can be installed on Windows, Linux, and Mac OS X. In Ubuntu, there are prebuilt DEB binaries that we can install from Software Center. We can also install PyAIML from source code. The current PyAIML will work well in Python 2.7. Let's look at how we can install it.

Installing PyAIML on Ubuntu 16.04 LTS

Installing PyAIML on Ubuntu is pretty easy and straightforward. We can install the package using the following command:

```
| $ sudo apt-get install python-aiml
```

The version of PyAIML will be 0.86.

We can also install PyAIML from source code. Clone the source code from Git using the following command:

```
| $ git clone https://github.com/qboticslabs/pyaiml
```

After cloning the package, switch to the PyAIML folder and install using the following command:

```
| $ sudo python setup.py install
```

Great! You are done with the installation. Let's check whether your installation is correct.

Playing with PyAIML

Take a Python interpreter Terminal and just try to import the AIML module using the following command:

```
| >>> import aiml
```

If the module is loaded properly, the pointer you will come to the next line without getting an error. Congratulations! Your installation is correct.

Let's see how to load an AIML file using this module.

To play with this module, first we need an AIML file. Save the following content in an AIML file called `sample.aiml` in the `home` folder. You can save the file anywhere, but it should be in the same path where the Python Terminal was started.

```
<aiml version="1.0.1" encoding="UTF-8">
  <category>
    <pattern> MY NAME IS * </pattern>
    <template>
      NICE TO SEE YOU <star/>
    </template>
  </category>

  <category>
    <pattern> MEET OUR ROBOTS * AND * </pattern>
    <template>
      NICE TO SEE <star index="1"/> AND <star index="2"/>.
    </template>
  </category>
</aiml>
```

After saving the AIML file, let's try to load it. The first step is to build an object of the PyAIML module called `Kernel()`. The object name here is `bot`:

```
| >>> bot = aiml.Kernel()
```

`Kernel()` is the main class doing the searching from the AIML knowledge base.

We can set the robot name using the following command:

```
| >>> bot.setBotPredicate("name", ROBIN)
```

The next step is to load the AIML files; we can load one or more AIML files to memory.

To learn a single AIML file, use the following command:

```
| >>> bot.learn('sample.aiml')
```

If the AIML file is correct, then you will get a message like this:

```
| Loading sample.aiml... done (0.02 seconds)
```

This means that the sample AIML file is loaded properly in memory.

We can retrieve the response from the AIML file using the following command:

```
| >>> print bot.respond("MY NAME IS LENTIN")
| 'NICE TO SEE YOU LENTIN'
```

If the user input is not in the file, you will get the following message:

```
| 'WARNING: No match found for input:'
```

Loading multiple AIML files

We have seen how to load a single AIML file to memory and retrieve response for a user input. In this section, we are going to see how to load multiple AIML files to memory; we are going to use these files for our AIML-based bots. Various AIML datasets are available on the Web, and some are also included in the code bundle. Given here is a file called `startup.xml` that helps us load all AIML files in a single run. It's a simple AIML file with a pattern called `LOAD AIML B`. When it gets this input from the user, it will learn all AIML files in that path using `<learn>*.aiml</learn>` tags:

```
<aiml version="1.0">
  <category>
    <pattern>LOAD AIML B</pattern>
    <template>
      <!-- Load standard AIML set -->
      <learn>*.aiml</learn>
    </template>
  </category>
</aiml>
```

We can use the following code to load this XML file and "learn" all the XML files to memory. After loading the AIML files, we can save the memory contents as a brain file. The advantage is that we can avoid the reloading of AIML files. Saving into a brain file will be helpful when we have thousands of AIML files:

```
#!/usr/bin/env python
import aiml
import sys
import os
#Changing current directory to the path of aiml files
#This path will change according to your location of aiml files
os.chdir('/home/robot/Desktop/aiml/aiml_data_files') bot =
aiml.Kernel()
#If there is a brain file named standard.brn, Kernel() will
#initialize using bootstrap() method
if os.path.isfile("standard.brn"): bot.bootstrap(brainFile =
"standard.brn") else:
#If there is no brain file, load all AIML files and save a new
brain bot.bootstrap(learnFiles = "startup.xml", commands = "load
aiml b") bot.saveBrain("standard.brn")
#This loop ask for response from user and print the output from
Kernel() object
while True: print bot.respond(raw_input("Enter input >"))
```

You can see that the AIML files are stored at `/home/robot/Desktop/aiml/aiml_data_files/`. All AIML files including `startup.xml` and AIML brain files are stored in the same folder. You can choose any folder you want. In the previous code, we are using a new API called `bootstrap()` for loading, saving, and learning AIML files. The program tries to load a brain file called `standard.brn` first, and if there is no brain file, it will learn from `startup.xml` and save the brain file as `standard.brn`. After saving the brain file, it will start a `while` loop to start interacting with the AIML file.

If you run the code and there is no brain file, you may get output like this:

```
robot@robot-pc: ~/Desktop/aiml
Loading reduction.names.aiml... done (0.85 seconds)
Loading geography.aiml... done (0.16 seconds)
Loading wallace.aiml... done (0.11 seconds)
Loading emotion.aiml... done (0.02 seconds)
Loading science.aiml... done (0.01 seconds)
Loading biography.aiml... done (0.08 seconds)
Loading computers.aiml... done (0.02 seconds)
Loading pyschology.aiml... done (0.11 seconds)
Loading date.aiml... done (0.00 seconds)
Loading psychology.aiml... done (0.10 seconds)
Loading politics.aiml... done (0.01 seconds)
Loading mp1.aiml... done (0.62 seconds)
Loading mp0.aiml... done (1.08 seconds)
Loading mp6.aiml... done (0.37 seconds)
PARSE ERROR: Unexpected <category> tag (line 40, column 0)
PARSE ERROR: Unexpected </category> tag (line 43, column 0)
Loading ai.aiml... done (0.04 seconds)
PARSE ERROR: Unexpected </category> tag (line 104, column 0)
PARSE ERROR: Unexpected </category> tag (line 144, column 0)
Loading update_mccormick.aiml... done (0.01 seconds)

Kernel bootstrap completed in 12.90 seconds
Saving brain to standard.brn... done (0.65 seconds)
Enter input >
```

Figure 3: Loading multiple AIML files

Creating an AIML bot in ROS

The previous subsections were about understanding AIML tags and how to work with them using the PyAIML module. Let's see how to create an interactive AIML bot using ROS. The following figure shows the complete block diagram of the interactive bot:

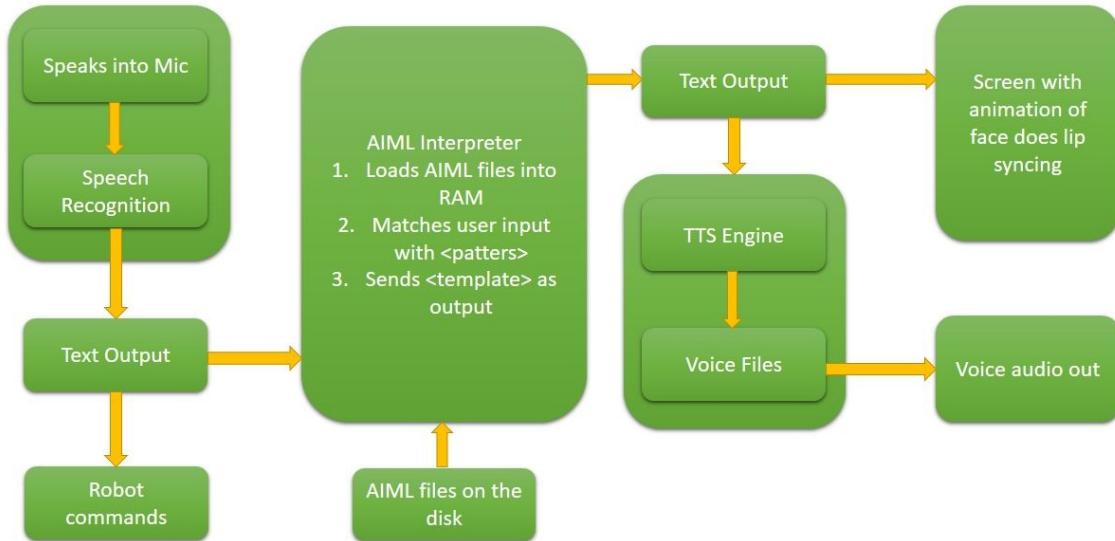


Figure 4: Interactive AIML bot

Here is how the entire system works: The speech of the user is converted into text using the speech recognition system in ROS. Then, it will input either to the AIML engine or send as a robot command. The robot commands are specific commands meant for robot control. If the text is not a robot command, it will send it to the AIML engine, which will give an intelligent reply from its database. The output of the AIML interpreter will be converted to speech using the text-to-speech module. The speech will be heard through speaker at the same time a virtual face of the robot will be animated on the screen, syncing with the speech.

In this chapter, we are mainly dealing with the AIML part and TTS using ROS; you can refer to other sources to perform speech recognition in ROS as well.

The AIML ROS package

In this section, we are going to create a simple package to load the AIML files to memory using ROS nodes. The following is the block diagram of the working AIML ROS package:

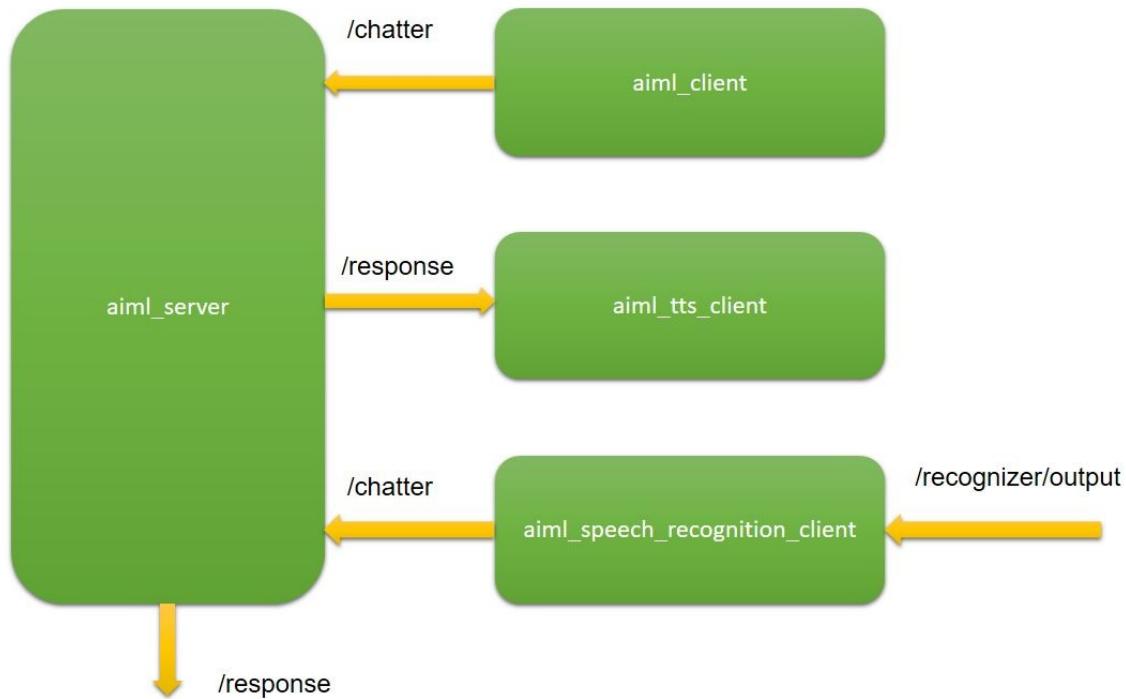


Figure 5: Working of the AIML ROS package

Here's the explanation for the nodes shown in the diagram:

- **aiml_server**: This ROS node loads AIML files from the database and saves them into brain files. It subscribes to a topic called **/chatter** (`std_msgs/String`). The string data from the **/chatter** topic is the input of the AIML interpreter. The response from the AIML interpreter is published through the **/response** (`std_msgs/String`) topic.
- **aiml_client**: This ROS node waits for user input, and once it gets the input, it will publish it to the **/chatter** topic.
- **aiml_tts_client**: The AIML server publishes the response to the **/response** topic. The `tts` client node will subscribe to this topic and convert it to speech.
- **aiml_speech_recognition_client**: This node will subscribe to the output from the speech recognition system and publish it to the **/chatter** topic.

The user can interact with AIML either by text chatting or speech. The speech recognition node will not do speech recognition; instead, it will receive the converted text from a speech recognition system and input it to the AIML server.

To create or install the `ros-aiml` package, you may need to install some dependency packages.

Installing the ROS sound_play package

The `sound_play` package is a **TTS** convertor package in ROS. You can obtain more information about the package from http://wiki.ros.org/sound_play. To install this package, you will need install some Ubuntu package dependencies. Let's go through the commands to install them.

Installing the dependencies of sound_play

Update your Ubuntu repositories using the following command:

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

These are the dependencies required for the `sound_play` package:

```
| $ sudo apt-get install libgstreamer1.0-dev libgstreamer-plugins-base1.0-dev gstreamer1.0 gstreamer1.0-plugins-base gstreamer1.0-plugins-good gstreamer1.0-plugins-ugly python-gi festival
```

After installing these Ubuntu packages, you can install the `sound_play` package using the following steps.

Installing the sound_play ROS package

Clone the `audio-common` packages into `ros_project_dependencies_ws`:

```
| ros_project_dependencies_ws/src$ git clone https://github.com/ros-drivers/audio_common
```

Install the packages using `catkin_make`.

After installing these packages, you can make sure it is properly installed using the following command:

```
| $ rosnode list
```

If it switches to the `sound_play` package, you have installed it successfully.

Congratulations! You are done with all dependencies! Next, we will create the `ros-aiml` package.



*You can clone the source code discussed in the section from the following Git repository:
https://github.com/qboticslabs/ros_robots_projects*

Creating the ros_aiml package

Using the following command, we can create the `ros_aiml` package:

```
| $ catkin_create_pkg ros_aiml rospy std_msgs sound_play
```

Inside the `ros_aiml` package, create folders called `data`, `scripts`, and `launch` to store the AIML files, Python scripts, and ROS launch files. This is the structure of the `ros_aiml` package:

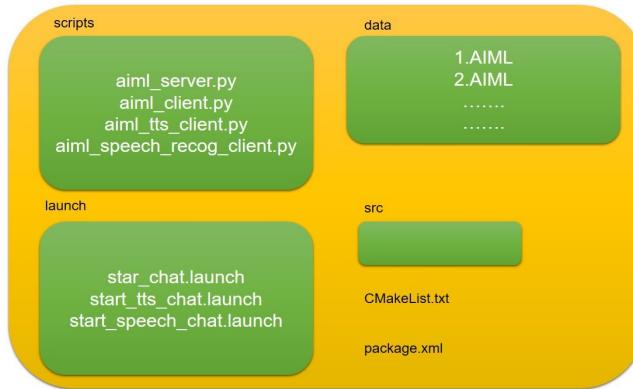


Figure 6: Structure of `ros_aiml`

You can keep the AIML files inside the `data` folder, and all launch files can be kept inside the `launch` folder. The scripts are saved inside the `scripts` folder. Let's look at each script.

The aiml_server node

As we've already discussed, `aiml_server` is responsible for loading and saving the AIML and AIM brain files. It is subscribed to the `/chatter` topic, which is the input of the AIML interpreter and publishes the `/response` topic, which is the response from the AIML interpreter. This is the main code snippet of `aiml_server.py`:

```
def load_aiml(xml_file):

    data_path = rospy.get_param("aiml_path")
    print data_path
    os.chdir(data_path)

    if os.path.isfile("standard.brn"):
        mybot.bootstrap(brainFile = "standard.brn")

    else:
        mybot.bootstrap(learnFiles = xml_file, commands = "load aiml
b")
        mybot.saveBrain("standard.brn")

def callback(data):

    input = data.data
    response = mybot.respond(input)
    rospy.loginfo("I heard:: %s",data.data)
    rospy.loginfo("I spoke:: %s",response)
    response_publisher.publish(response)

def listener():

    rospy.loginfo("Starting ROS AIML Server")
    rospy.Subscriber("chatter", String, callback)

    # spin() simply keeps python from exiting until this node is
    stopped
    rospy.spin()

if __name__ == '__main__':
    load_aiml('startup.xml')
    listener()
```

This ROS node is doing the same thing as the code that we used to load and save the AIML files. That code is converted into a ROS node that can accept input and send the response through a topic.



You can clone the source code discussed in the section from the following Git repository:

https://github.com/qboticslabs/ros_robotics_projects

The AIML client node

The client code will wait for user input and publish the user input to the `/chatter` topic:

```
#!/usr/bin/env python
import rospy
from std_msgs.msg import String
pub = rospy.Publisher('chatter', String, queue_size=10)
rospy.init_node('aiml_client')
r = rospy.Rate(1) # 10hz

while not rospy.is_shutdown():
    input = raw_input("\nEnter your text :> ")
    pub.publish(input)
    r.sleep()
```

The aiml_tts client node

The TTS client subscribes to the `/response` topic and converts the response to speech using the `sound_play` APIs:

```
#!/usr/bin/env python
import rospy, os, sys
from sound_play.msg import SoundRequest
from sound_play.libsoundplay import SoundClient
from std_msgs.msg import String
rospy.init_node('aiml_soundplay_client', anonymous = True)

soundhandle = SoundClient()
rospy.sleep(1)
soundhandle.stopAll()
print 'Starting TTS'

def get_response(data):
    response = data.data
    rospy.loginfo("Response ::%s",response)
    soundhandle.say(response)

def listener():
    rospy.loginfo("Starting listening to response")
    rospy.Subscriber("response",String, get_response,queue_size=10)
    rospy.spin()
if __name__ == '__main__':
    listener()
```

The AIML speech recognition node

The speech recognition node subscribes to `/recognizer/output` and publishes to the `/chatter` topic:

```
#!/usr/bin/env python
import rospy
from std_msgs.msg import String
rospy.init_node('aiml_speech_recog_client')
pub = rospy.Publisher('chatter', String, queue_size=10)
r = rospy.Rate(1) # 10hz

def get_speech(data):
    speech_text=data.data
    rospy.loginfo("I said:: %s",speech_text)
    pub.publish(speech_text)

def listener():
    rospy.loginfo("Starting Speech Recognition")
    rospy.Subscriber("/recognizer/output", String, get_speech)
    rospy.spin()

while not rospy.is_shutdown():
    listener()
```

The `/recognizer/output` topic is published by ROS speech recognition packages such as Pocket Sphinx (<http://wiki.ros.org/pocketsphinx>).

Next, we'll look at the launch files used for starting each node.

start_chat.launch

The `start_chat.launch` launch file launches the `aiml_server` and `aiml_client` nodes. Before running this launch file, you have to set the data folder path that is set as the ROS parameter. You can set it as your AIML data folder path:

```
<launch>
  <param name="aiml_path"
  value="/home/robot/ros_robotics_projects_ws/src/ros_aiml/data" />
  <node name="aiml_server" pkg="ros_aiml" type="aiml_server.py"
output="screen">
  </node>
  <node name="aiml_client" pkg="ros_aiml" type="aiml_client.py"
output="screen">
  </node>
</launch>
```

start_tts_chat.launch

The launch file launches the `aiml_server`, `aiml_client`, and `aiml_tts` nodes. The difference between the previous launch file and this one is that this will convert the AIML server response into speech:

```
<launch>

  <param name="aiml_path"
    value="/home/robot/ros_robotics_projects_ws/src/ros_aiml/data" />
    <node name="aiml_server" pkg="ros_aiml" type="aiml_server.py"
output="screen">
      </node>
      <include file="$(find sound_play)/soundplay_node.launch">
    </include>
    <node name="aiml_tts" pkg="ros_aiml" type="aiml_tts_client.py"
output="screen">
      </node>
      <node name="aiml_client" pkg="ros_aiml" type="aiml_client.py"
output="screen">
      </node>
    </include>
  </node>
</launch>
```

start_speech_chat.launch

The `start_speech_chat.launch` launch file will start the AIML server, AIML TTS node, and speech recognition node:

```
<launch>
  <param name="aiml_path"
  value="/home/robot/ros_robotics_projects_ws/src/ros_aiml/data" />
  <node name="aiml_server" pkg="ros_aiml" type="aiml_server.py"
output="screen">
  </node>
  <include file="$(find sound_play)/soundplay_node.launch">
  </include>
  <node name="aiml_tts" pkg="ros_aiml" type="aiml_tts_client.py"
output="screen">
  </node>
  <node name="aiml_speech_recog" pkg="ros_aiml"
type="aiml_speech_recog_client.py" output="screen">
  </node>
</launch>
```

After creating the launch file, change its permission using the following command:

```
| $ sudo chmod +x *.launch
```

Use the following command to start interacting with the AIML interpreter:

```
| $ roslaunch ros_aiml start_chat.launch
```

We can use the following command to start interacting with the AIML interpreter. The response will be converted to speech as well:

```
| $ roslaunch ros_aiml start_tts_chat.launch
```

The following command will enable speech recognition and TTS:

```
| $ roslaunch ros_aiml start_speech_chat.launch
```

If you set up the `pocketsphinx` package for speech recognition, you can run it using the following command:

```
| $ roslaunch pocketsphinx robotcup.launch
```

```

auto-starting new master
process[master]: started with pid [3049]
ROS_MASTER_URI=http://localhost:11311

setting /run_id to 640a4dd4-70f3-11e6-bf20-00177c2e2869
process[rosout-1]: started with pid [3062]
started core service [/rosout]
process[aiml_server-2]: started with pid [3069]
process[soundplay_node-3]: started with pid [3080]
process[aiml_tts-4]: started with pid [3081]
process[aiml_speech_recog-5]: started with pid [3082]
/home/robot/ros_robots_projects_ws/src/ros_aiml/data
[INFO] [WallTime: 1472810134.642040] Starting Speech Recognition
Starting TTS
[INFO] [WallTime: 1472810135.638718] Starting listening to response
Loading brain from standard.brn... done (98290 categories in 1.80 seconds)
Kernel bootstrap completed in 1.80 seconds
[INFO] [WallTime: 1472810136.418132] Starting ROS AIML Server
[INFO] [WallTime: 1472810228.682427] I said:: Hello
[INFO] [WallTime: 1472810228.687356] I heard:: Hello
[INFO] [WallTime: 1472810228.687598] I spoke:: Hi there!
[INFO] [WallTime: 1472810228.687995] Response ::Hi there!

```

Figure 7: Output of the start_speech_chat launch file

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

```

robot@robot-pc:~$ rostopic list
/chatter
/diagnostics
/recognizer/output
/response
/robotsound
/rosout
/rosout_agg
/sound_play/cancel
/sound_play/feedback
/sound_play/goal
/sound_play/result
/sound_play/status
robot@robot-pc:~$ 

```

Figure 8: List of ROS topics

We can test the entire system without the speech recognition system too. You can manually publish the string to the /recognizer/output topic, as shown here:

```

robot@robot-pc:~$ rostopic pub /recognizer/output std_msgs/String "Hello"
publishing and latching message. Press ctrl-C to terminate

```

Figure 9: Manually publishing input to speech topic

Questions

- What are the various applications of AI in robotics?
- What is AIML and why is it used?
- What is a pattern and template in AIML?
- What is PyAIML and what are its functions?

Summary

In this chapter, we discussed building a ROS package to make a robot interactive using artificial intelligence. Using this package, we can talk to a robot and the robot can answer your queries, like human-to-human interaction. The entire chapter was about building this communication system using AIML, which is the main component of this project. We discussed AIML tags, how to work with AIML files using Python, and ultimately, how to build a ROS package based on AIML for an interactive robot. In the next chapter, we will discuss interfacing boards with the ROS.

Controlling Embedded Boards Using ROS

Do you know how a robot makes decisions according to its sensor data? It has a processing unit, right? The processing unit can be either a computer or a microcontroller. We are using high-end computers to process data if the robot has sensors such as camera, laser scanners, and LIDARs. On the other hand, microcontrollers are commonly used in all kind of robots for interfacing low-bandwidth sensors and for performing real-time tasks. Both these units are commonly found in a standard robotic system.

In small robots such as line follower, we may do everything using a single controller. The sensors such as ultrasonic distance sensors, Imus can easily interface with a microcontroller. So in a robotic system, these two units can work independently, and there will be some kind of communication happening between them.

In this chapter, we will discuss how we can communicate with an embedded controller board from a computer running on ROS. It is very useful to acquire sensor data from a controller using a computer, and you can do the remaining processing on the computer. In most of the high-end robots, both controller and computer are used for low-level and high-level control and processing.

We'll also look at some popular embedded boards and their interfacing techniques with ROS.

Here are the topics we will look in this chapter:

- Getting started with popular embedded boards
- Interfacing Arduino with ROS
- Interfacing STM32 with ROS
- Working with Raspberry Pi 2 and ROS
- Odroid and ROS

Getting started with popular embedded boards

In this section, we will look at some of the popular microcontroller boards and microcomputers that can be used in robots.

An introduction to Arduino boards

Arduino is one of the most popular embedded controller boards that can be used in robots. It is mainly used for prototyping electronics projects and robots. The boards mainly contain an AVR series controller, in which its pins are mapped as Arduino board pins. The main reason for the Arduino board's popularity is in its programming and easiness in prototyping. The Arduino APIs and packages are very easy to use. So we can prototype our application without much difficulty. The Arduino programming IDE is based on a software framework called **Wiring** (<http://wiring.org.co/>); we are coding using C/ C++ in a simplified way. The code is compiled using C/C++ compilers. Here is an image of a popular Arduino board, the Arduino Uno:

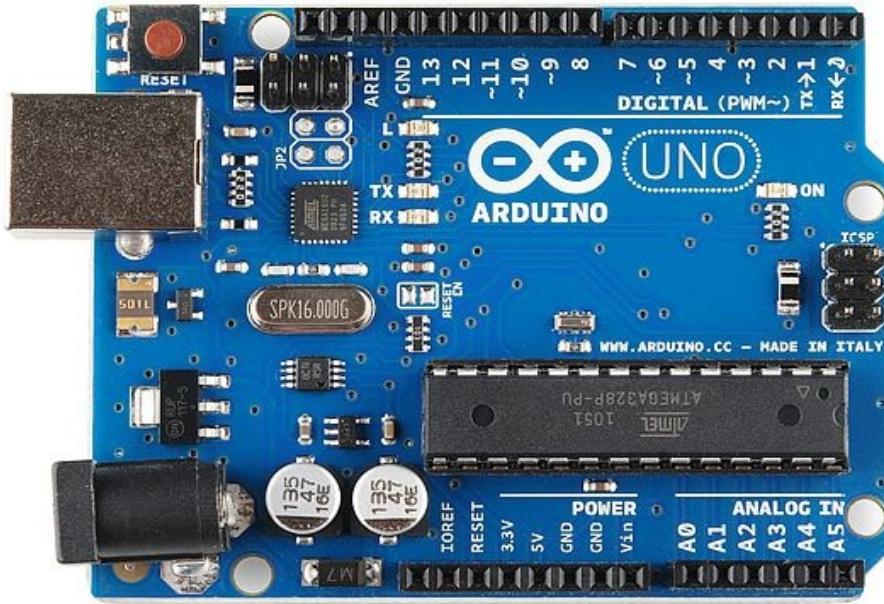


Figure 1: Arduino Uno board

How to choose an Arduino board for your robot

The following are some of the important specifications of this board that may be useful while selecting an Arduino board for your robot:

- **Speed:** Almost all Arduino boards work under 100 MHz. Most of the controllers on boards are 8 MHz and 16 MHz. If you want to do some serious processing such as implementing a PID on a single chip, then the Arduino may not be the best choice, especially if we want to run it at a higher rate. The Arduino is best suited for simple robot control. It is best for tasks such as controlling a motor driver and servo, reading from analog sensors, and interfacing serial devices using protocols such as **Universal Asynchronous Receiver/Transmitter (UART)**, **Inter-Integrated Circuit (I2C)**, and **Serial Peripheral Interface (SPI)**.
- **GPIO pins:** Arduino boards provide different kinds of I/O pins to developers, such as **general purpose input/output (GPIO)**, **analog-to-digital converter (ADC)**, and **pulse width modulation (PWM)**, I2C, UART, and SPI pins. We can choose Arduino boards according to our pin requirements. There are boards having a pin count from 9 to 54. The more pins the board has, the larger will be the size of the board.
- **Working voltage levels:** There are Arduino boards working on TTL (5V) and CMOS (3.3V) voltage levels. For example, if the robot sensors are working only in 3.3V mode and our board is 5V, then we have to either convert 3.3V to the 5V equivalent using a level shifter or use an Arduino working at 3.3V. Most Arduino boards can be powered from USB itself.
- **Flash memory:** Flash memory is an important aspect when selecting an Arduino board. The output hex file generated by the Arduino IDE may not be optimized when compared with the hex of embedded C and assembly code. If your code is too big, it is better to go for higher flash memory, such as 256 KB. Most basic Arduino boards have only 32 KB of flash memory, so you should be aware of this issue before selecting the board.
- **Cost:** One of the final criteria is of course the cost of the board. If your requirement is just for a prototype, you can be flexible; you can take any board. But if you are making a product using this, cost will be a constraint.

Getting started with STM32 and TI Launchpads

What do we do if the Arduino is not enough for our robotic applications? No worries; there are advanced ARM-based controller boards available, such as STM32 microcontroller based development boards like NUCLEO and **Texas Instrument (TI)** microcontrollers based boards like Launchpads. The STM32 is a family of 32-bit microcontrollers from a company called **STMicroelectronics** (http://www.st.com/content/st_com/en.html). They manufacture microcontrollers based on different ARM architectures, such as the Cortex-M series. The STM32 controllers offer a lot more clock speed than Arduino boards. The range of STM32 controllers are from 24 MHz to 216 MHz, and the flash memory sizes are from 16 KB to 2 MB. In short, STM32 controllers offer a stunning configuration with a wider range of features than the Arduino. Most boards work at 3.3V and have a wide range of functionalities on the GPIO pins. You may be thinking about the cost now, right? But the cost is also not high: the price range is from 2 to 20 USD.

There are evaluation boards available in the market to test these controllers. Some famous families of evaluation boards are as follows:

- **STM32 Nucleo boards:** The Nucleo boards are ideal for prototyping. They are compatible with Arduino connectors and can be programmed using an Arduino-like environment called **mbed** (<https://www.mbed.com/en/>).
- **STM32 Discovery kits:** These boards are very cheap and come built in with components such as an accelerometer, mic, and LCD. The mbed environment is not supported on these boards, but we can program the board using IAR, Keil, and **Code Composer Studio (CCS)**.
- **Full evaluation boards:** These kinds of boards are comparatively expensive and are used to evaluate all features of the controller.
- **Arduino-compatible boards:** These are Arduino header-compatible boards having STM32 controllers. Examples of these boards are Maple, OLIMEXINO-STM32, and Netduino. Some of these boards can be programmed using the Wiring language, which is used to program Arduino.

The STM32 boards are not more popular in the hobby/DIY community than the Arduino, but they are mainly used in high-end robot controllers. Here is an STM 32 Nucleo board:

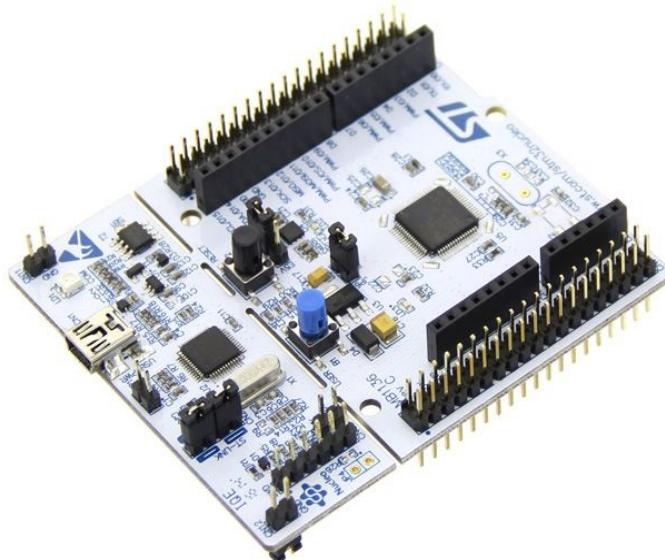


Figure 2: STM 32 NUCLEO board

The Tiva C Launchpad

One of the other alternatives to the Arduino is Launchpad from Texas Instruments. The TI controllers have specifications similar to STM32 controllers, and both are based on ARM's Cortex-M architecture. The clock speed of the controllers ranges from 48 MHz-330 MHz. The flash memory capacity is also high: up to 1 MB. The GPIO pins and cost are almost similar to STM32 boards. Some of the commonly used Launchpad boards are TM4C123G Launchpad and EK-TM4C1294XL, which is based on an ARM Cortex-M4F-based MCU. The 123G works at 80MHz and 1294XL at 120 MHz.

The good thing about these boards is that we can program them using a modified Arduino IDE called **Energia** (<http://energia.nu/>).

This is how the EK-TM4C1294XL looks:

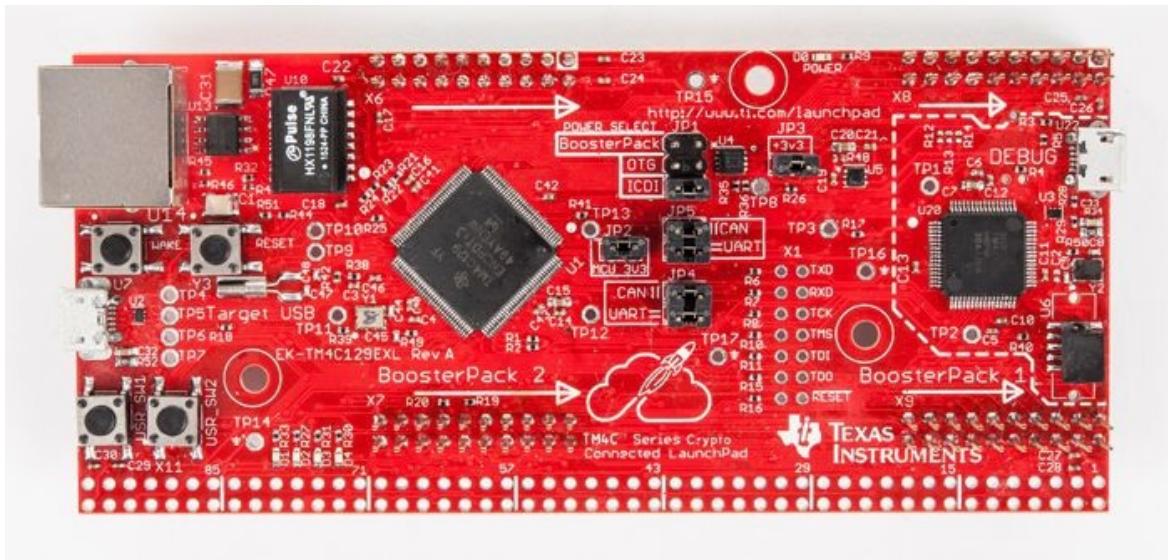


Figure 3: The EK-TM4C1294XL board



List of Arduino boards: <https://www.arduino.cc/en/Main/Products> *STM32 boards:* <https://goo.gl/w7qFuE>
Tiva C Series Launchpad: http://processors.wiki.ti.com/index.php/Tiva_C_Series_LaunchPads
Launchpad boards: [http://www.ti.com/lsds\(ti/tools-software/launchpads/launchpads.page](http://www.ti.com/lsds(ti/tools-software/launchpads/launchpads.page)

We have seen some popular controllers; now let's look at some of the high-level embedded processing units that can be used in robots.

Introducing the Raspberry Pi

Raspberry Pi is another popular embedded board, and it's a single-board computer on which we can load an operating system and use it like a full-fledged PC. It has a **system on chip (SoC)** comprising of components such as an ARM processor, RAM, and GPU. There is an Ethernet port, USB ports, HDMI, GPIO pins, sound jack, camera connector, and LCD connector:

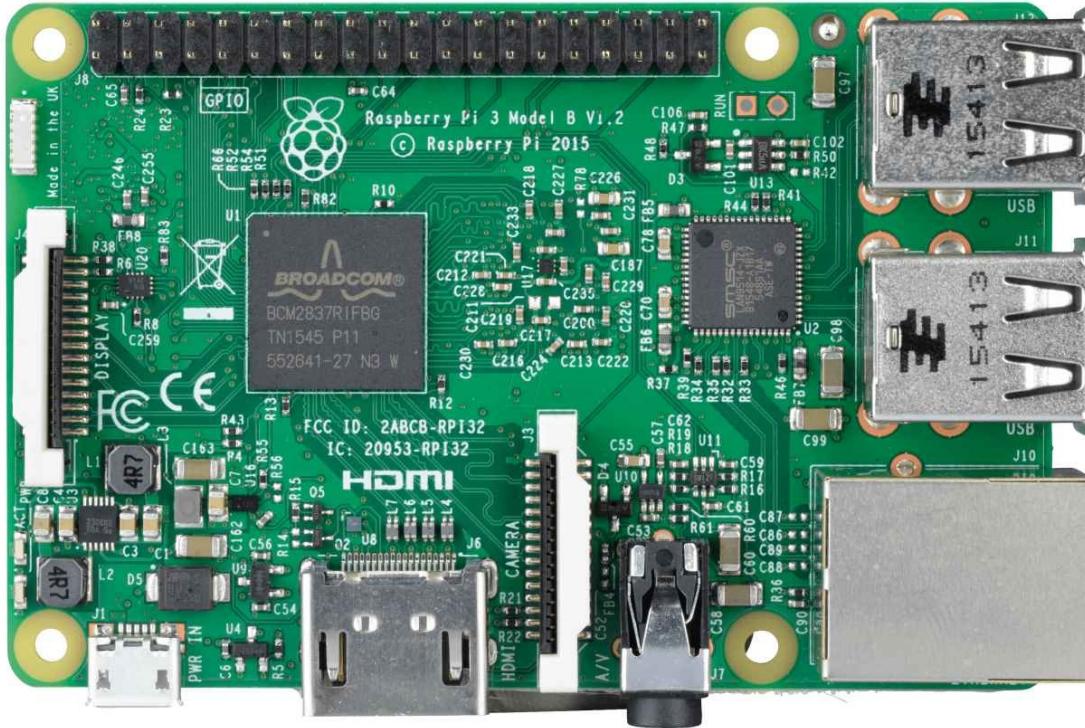


Figure 4: Raspberry Pi 3 board

 *List of Raspberry Pi boards: <https://www.raspberrypi.org/products/>*

How to choose a Raspberry Pi board for your robot

The following are some important specifications that may be useful while selecting this board for your robots:

- **Speed of the board:** The speed of the ARM processor in Raspberry Pi boards ranges from 700 MHz to 1.2 GHz. These boards are suitable for running an OS and building robotics applications on top of it. We can perform processor-intensive tasks such as image processing on the board. Don't pick this board if you have multiple image-processing applications and other tasks for the robot. They won't properly run on the board. It can freeze the entire system too. This board is perfectly suited for a single robotics application. The latest board, the Raspberry Pi 3, can offer you better performance for robotics applications.
- **Memory:** The RAM of the board ranges from 256 MB to 1 GB. If the robot application involves a lot of data processing, it may need a good amount of RAM. So for an image processing application, we should select a board with a large RAM size.
- **GPIO:** The main feature of Raspberry Pi boards is they have dedicated GPIO pins. The GPIO pins have multiple functions, such as I2C, UART, SPI, and PWM. We can't interface an analog sensor with the Pi because there are no inbuilt ADC pins. The GPIO pins are 3.3V compatible, so to interface with TTL logic, we may need a level shifter or voltage divider circuit. For interfacing analog sensors, we may need to interface an external ADC to the Raspberry Pi. The board has a maximum of 40 GPIO pins.
- **Power rating:** The Raspberry Pi works on 5V, which can take up to 2A current during operation. It will be good if you can provide this rating for the RPi. The RPi can even work from a computer USB port, but the power rating can vary according to the processing. So to be safe, it will be good if we can provide a 5V/2A rating.
- **Cost:** This is one of the most important criteria while choosing an RPi board. The price range of RPi boards is from 19 USD to 40 USD. You can choose the latest and most expensive board, the Raspberry Pi 3, to get maximum performance. The selection of board will depend on your robotics application.

The Odroid board

If you want more processing power than a Raspberry Pi board and with the same form factor, then Odroid is for you. The Odroid-C2 and Odroid-XU4 are the latest Odroid models, with 1.5 GHz and 2 GHz quad-core processors and 2 GB RAM, and almost the same power consumption as the RPi.

Odroid can be loaded with the latest version of Ubuntu, Android, and many flavors of Linux. It is a good choice if you are planning for an embedded powerhouse in a very small form factor. Let's discuss some of the models of Odroid.

The Odroid-XU4 is the most powerful and expensive board in the series. This board is ideal for running ROS and image-processing application. It has eight cores running at 2 GHz and with 2 GB of RAM.

The ODROID-C2 runs at 1.5 GHz, on a quad-core processor with 2 GB of RAM. The Odroid-C1+ and C1 have almost the same configuration as the C2, with the main difference being that the C1/C1+ only have 1 GB RAM, as opposed to the C2's 2 GB. These two boards are priced almost the same as Raspberry Pi's high-end boards. They are clear competitors to the Raspberry Pi.



Figure 5: The Odroid board series

This subsection should be enough for you to get an idea of popular embedded boards that can be used for robots. Next, we can start discussing interfacing ROS with some of these boards. We are not going to discuss too deeply about the interfacing concept; rather than that, we will mainly focus on the procedures to get the board ready to work with ROS. We will also learn about some of the sensor interfacing, using which we can read sensor values using a controller board and read into ROS.

 *List of Odroid boards:* http://www.hardkernel.com/main/products/prdt_info.php

Interfacing Arduino with ROS

Interfacing an Arduino board with ROS simply means running a ROS node on Arduino that can publish/subscribe like a normal ROS node. An Arduino ROS node can be used to acquire and publish sensor values to a ROS environment, and other nodes can process it. Also, we can control devices, for example, actuators such as DC motors, by publishing values to an Arduino node. The main communication between PC and Arduino happens over UART. There is a dedicated protocol called ROS Serial (<http://wiki.ros.org/rosserial/Overview>), implemented as a ROS metapackage called `rosserial`, which can encode and decode ROS Serial messages. Using the ROS Serial protocol, we can publish and subscribe to Arduino like a ROS node over UART.

To start with ROS interfacing of Arduino, follow these steps:

1. First, we have to install some ROS packages on Ubuntu. The following commands can be used to install them.
2. Installing the `rosserial` metapackage:

```
| $ sudo apt-get install ros-kinetic-rosserial
```

3. The following command will install the `rosserial-arduino` client package on ROS. This client package helps create a client library of the Arduino IDE for ROS. Using this library, we can create Arduino ROS nodes that work like a normal ROS node.

```
| $ sudo apt-get install ros-kinetic-rosserial-arduino
```

4. After installing these packages, you need to download and set up the Arduino IDE. We need to download this IDE to program Arduino boards. You can download the latest Arduino IDE from (<https://www.arduino.cc/en/Main/Software>).
5. You can download the Arduino IDE for Linux 64/32-bit according to your OS configuration and run the `arduino` executable after extracting the package.
6. To add the ROS library for the Arduino IDE, first you have to go to File | Preference and set the Sketchbook location, as shown in this screenshot:

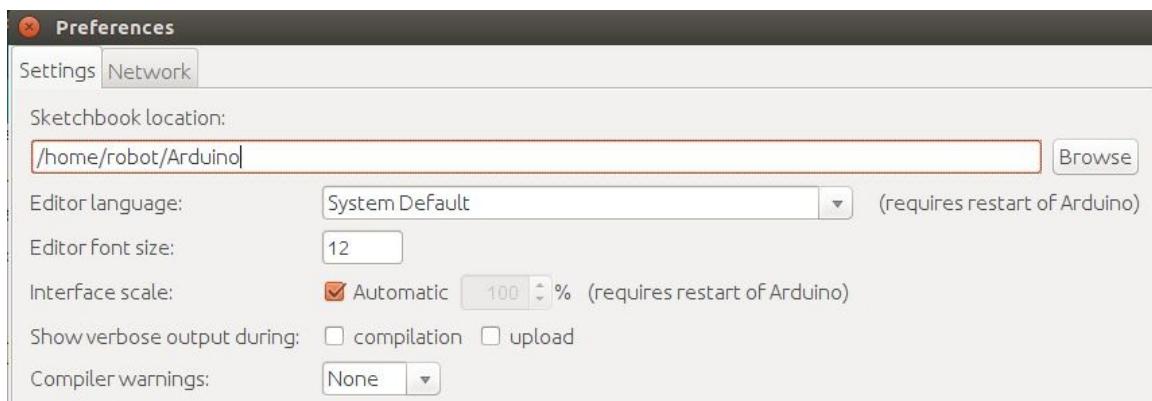


Figure 6: Arduino board preference

7. Go to the sketchbook location and create a folder called `libraries` if it is not present, and open a

Terminal inside the `libraries` folder. We are keeping all Arduino libraries on this folder. Enter the following command to generate the `ros_lib` library for Arduino:

```
$ rosrun rosserial_arduino make_libraries.py .
```

8. You will see the following messages printing during the execution of the command. You may get an error after some time, but that's perfectly fine.

```
Exporting rosserial_msgs
Messages:
Log, TopicInfo,
Services:
RequestServiceInfo, RequestMessageInfo, RequestParam,
Exporting std_srvs
Services:
Trigger, Empty, SetBool,
Exporting std_msgs
Messages:
Time, Int32MultiArray, Byte, ColorRGBA, Int8MultiArray, Int32, Float32, String, Char,
UInt8MultiArray, UInt64MultiArray, Bool, Header, Float64MultiArray, MultiArrayDimension,
Float32MultiArray, UInt32, UInt64, Int16, Int64MultiArray, UInt8, UInt16MultiArray,
Int16MultiArray, Empty, MultiArrayLayout, Int8, UInt32MultiArray, Int64, Float64, Duration,
UInt16, ByteMultiArray,
Exporting geometry_msgs
```

Figure 7: Building the Arduino ROS library

9. After the execution of this command, a folder called `ros_lib` will be generated, which is the Arduino ROS serial client library.
10. Now, you can open the Arduino IDE and check that the option highlighted in the following figure is available. You can take any of the ROS examples and compile and check whether it is building without any errors:

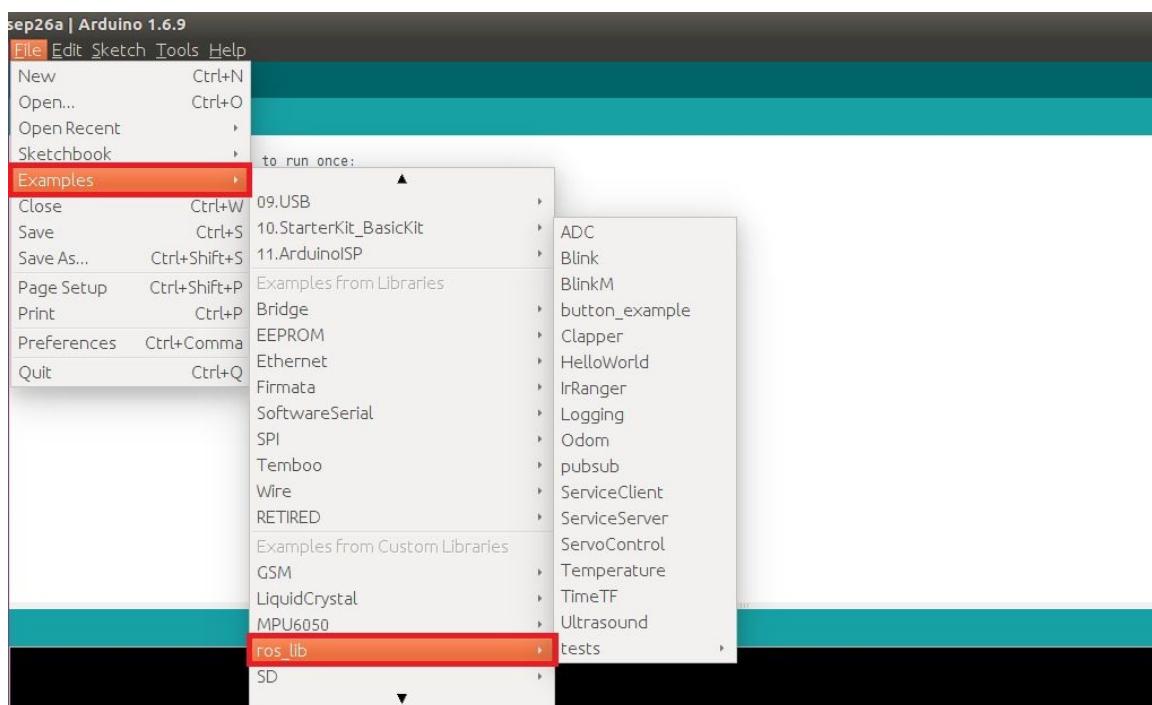


Figure 8: ros_lib on Arduino IDE

Congratulations! You have successfully set up `ros_lib` on Arduino. Now we can perform a few experiments using the ROS-Arduino interface.

Monitoring light using Arduino and ROS

We can start coding a basic Arduino-ROS node that can sense the amount of light using a **light-dependent resistor (LDR)**. You can use any Arduino for this demo; here, we are going to use the Arduino Mega 2560. Given in the following figure is the circuit of an LDR with the Arduino. The characteristic of an LDR is that it is basically a resistor in which the resistance across it changes when light falls on it. The maximum resistance is when there is no light and minimum when light falls on it.

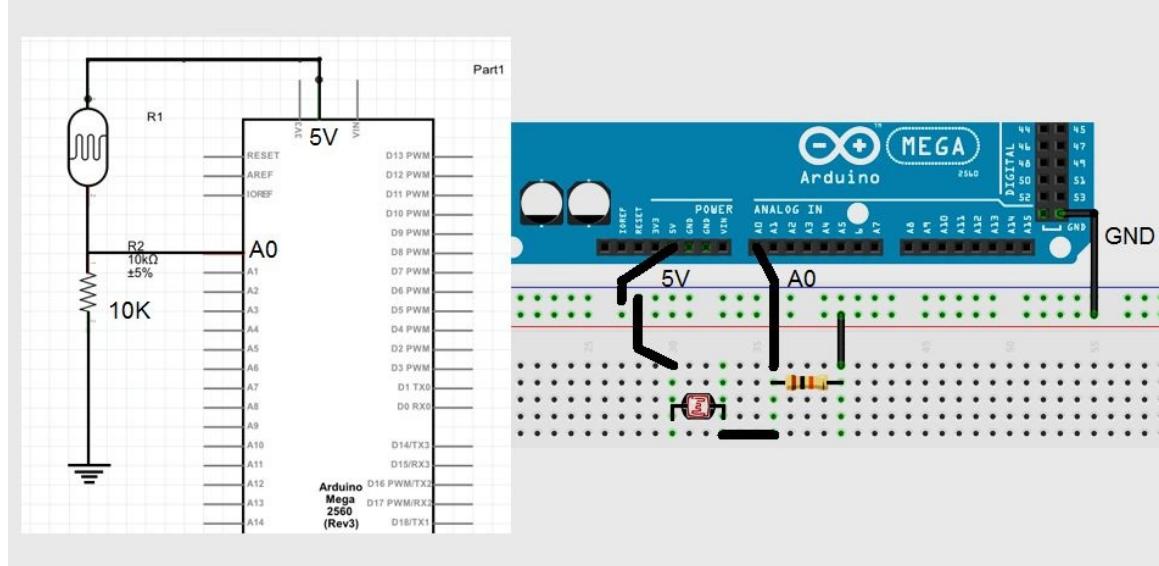


Figure 9: Arduino-LDR interfacing circuit

We have to connect one pin to 5V from the Arduino board and the next terminal to the Arduino's `A0` pin. That terminal is connected to the `GND` pin through a $10\text{ K}\Omega$ resistor. It is basically a voltage divider circuit. The equation for finding the voltage at `A0` is as follows:

$$V_{A0} = 5 * (R2 / (R1 + R2))$$

From the equation, it is clear that when there is no light, we will get the minimum voltage, and when there is light, we'll get the maximum. This value can be read out using an Arduino program.

Here is the ROS code to read from an LDR:

```
#include <Arduino.h>
#include <ros.h>
#include <rosserial_arduino/Adc.h>

rosserial_arduino::Adc adc_msg;
rosserial_arduino::Publisher p("adc", &adc_msg);

void setup()
{
    nh.initNode();
    nh.advertise(p);
}

//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;
}

long adc_timer;

void loop()
{
    adc_msg.adc0 = averageAnalog(0);
    p.publish(&adc_msg);
    nh.spinOnce();
    delay(50);
}

```

Here is the explanation of the code:

```

#include <Arduino.h>
#include <ros.h>
#include <rosserial_arduino/Adc.h>

```

The `<Arduino.h>` library contains definitions of Arduino-specific functions. The `<ros.h>` library contains Arduino-to-ROS client functionalities. The `<rosserial_arduino/Adc.h>` header contains message definitions for carrying several ADC values in a single message.

```
| ros::NodeHandle nh;
```

This creates a ROS node handle. Like other ROS nodes, we are using this handle to publish and subscribe to Arduino.

```

| rosserial_arduino::Adc adc_msg;
| ros::Publisher p("adc", &adc_msg);

```

This code will create an `adc_msg` instance and create a publisher object.

```

void setup()
{
    nh.initNode();
    nh.advertise(p);
}

```

This will initialize the node and bind the publisher object to start publishing the topic called `/adc`.

```

void loop()
{
    adc_msg.adc0 = averageAnalog(0);
    p.publish(&adc_msg);
    nh.spinOnce();
    delay(50);
}

```

In the loop, the Analog value from pin `A0` is read and the average is computed. The average value will be published to the `/adc` topic.

After compiling the code, you can select the board from Tools | Board and Serial Port from the list. You can now burn the code into the Arduino board.

Running ROS serial server on PC

After burning the code, to start subscribing or publishing to the Arduino board, we should start the ROS serial server on the PC side. Let's see how to do so:

1. Initialize `roscore`:

```
| $ roscore
```

2. Run the ROS serial server on the PC. The argument of the server is the serial device name of the Arduino device:

```
| $ rosrun rosserial_python serial_node.py /dev/ttyACM0
```

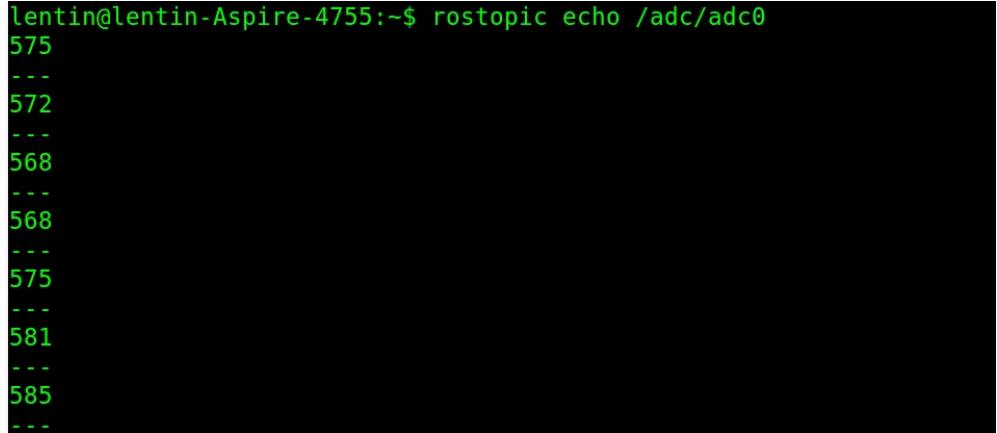
3. Now, you can see the `/adc` topic using the following command:

```
| $ rostopic list
```

4. You can echo the `/adc` topic using the following command:

```
| $ rostopic echo /adc/adc0
```

5. You may get following values:



A terminal window showing the output of the command `rostopic echo /adc/adc0`. The output displays a series of integer values representing LDR sensor readings. The values fluctuate between 568 and 585, with some intermediate values like 572 and 575.

```
lentin@lentin-Aspire-4755:~$ rostopic echo /adc/adc0
575
---
572
---
568
---
568
---
575
---
581
---
585
---
```

Figure 10: Displaying LDR values from the ROS topic

We can also visualize the sensor value using `rqt_plot` using the following command. Now you can vary the light around the sensor and can check the variation of the values. The readings of the LDR are mapped from 1 to 1023. If there is no light, that means there's a high resistance in the LDR, so there'll be a low voltage across it and low reading on the Arduino, and vice versa.

```
| $ rqt_plot adc/adc0
```

You can see this in the following graph:

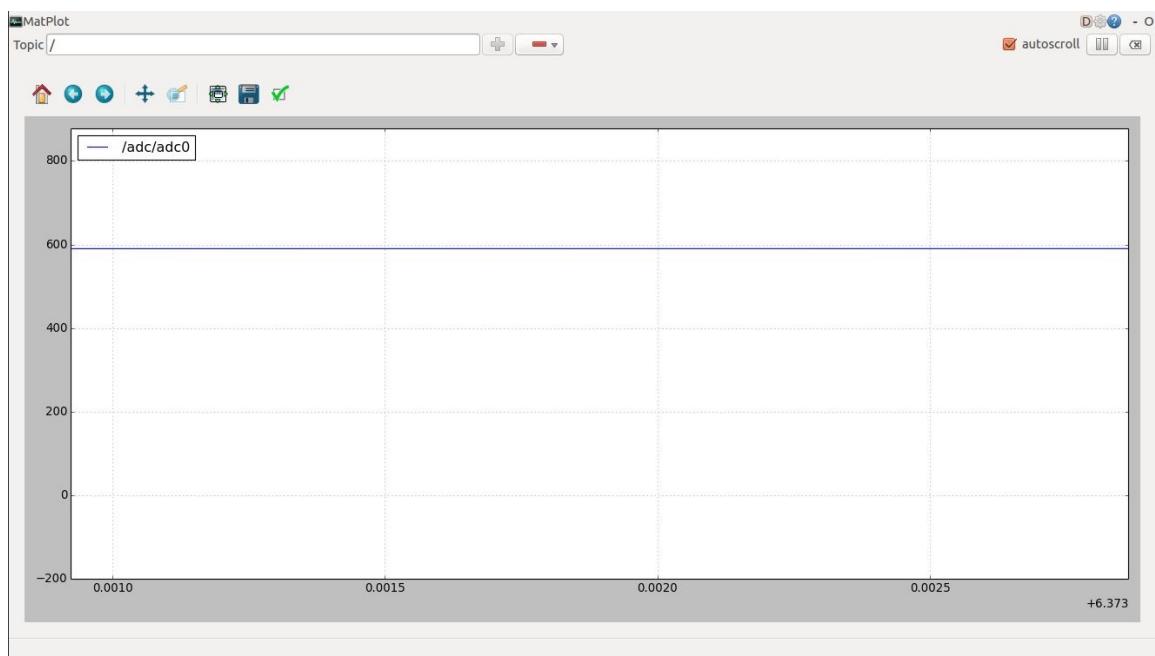


Figure 11: Visualizing LDR values in rqt_plot

Interfacing STM32 boards to ROS using mbed

If Arduino is not enough for your application, the STM 32 boards are ready to serve you. To demonstrate ROS interfacing, we are going to use an STM 32 NUCLEO L476RG (<https://developer.mbed.org/platforms/ST-Nucleo-L476RG/>). Before we begin programming, let's understand the mbed platform. The mbed platform is a software platform for programming 32-bit ARM Cortex-M microcontrollers. The mbed platform developed as a collaborative project by ARM its technology partners. We can use the online mbed IDE or offline compilers for programming the boards. The advantage of using the online IDE is it will be updated and will have more hardware support.

Let's start programming the STM 32 board:

1. The first step is to create an account on the mbed website, which is <https://developer.mbed.org>.
2. After creating an account, go to the following link to check our board has support in the mbed platform: <https://developer.mbed.org/platforms/>.
3. You can select your board from this website; for this demo, you should choose the NUCLEO L476RG board, which is available at <https://developer.mbed.org/platforms/ST-Nucleo-L476RG/>.
4. You can see an option called Add to your mbed compiler on the right-hand side of this page. You have to click on this button to add this board to the mbed compiler. We can add any number of boards to the mbed compiler; also, we can choose the board before compiling.
5. After adding the board to the compiler, we can compile a ROS node for this board. As we've already discussed, we can program the board using the online IDE or an offline compiler such as gcc4embed (<https://github.com/adamgreen/gcc4mbed>). Using offline compilers, we can only program a limited number of boards, but the online IDE can handle the latest boards.
6. The programming APIs of the ROS node in STM 32 are the same as those for Arduino, only the environment and tools are different.
7. The online `ros_lib` files for mbed are available at <https://developer.mbed.org/users/garyservin/code/>. You can find `ros_lib` for the Kinetic, Jade, and Indigo versions. You can try with the ROS version you are working on.
8. You can look at Hello World code for each ROS distribution from the preceding link.



You can check out examples for ROS Kinetic at https://developer.mbed.org/users/garyservin/code/ros_lib_kinetic. For ROS Jade, the link is https://developer.mbed.org/users/garyservin/code/ros_lib_jade/. For ROS Indigo: https://developer.mbed.org/users/garyservin/code/ros_lib_indigo

9. You can import the code into the compiler using the following option:



Figure 12: Importing code to mbed in the online compiler

- This will open the source code in the mbed online IDE, as shown in the next screenshot. Here, we are testing with Hello World code for ROS Indigo.
- The area marked **1** is the board we have added to the compiler. Area **2** is imported source code and `ros_lib` for mbed, and area **3** is the button to compile the source code. You can see the debugging details at the bottom of the compiler:

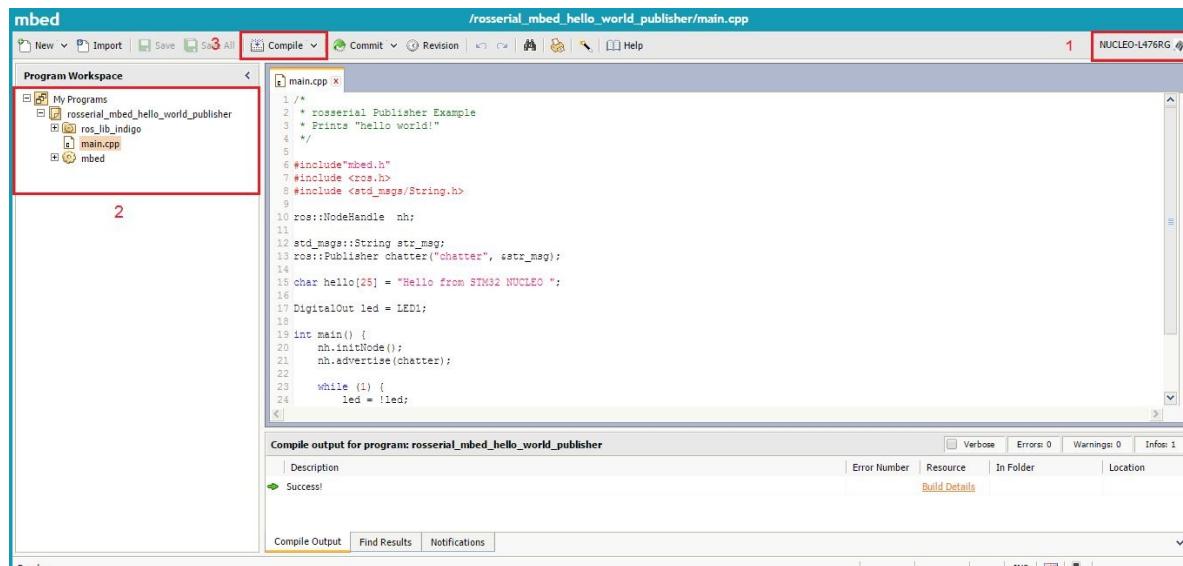


Figure 13: The mbed online compiler

- The APIs are the same as those of Arduino we saw in the previous section. In this code, we are publishing a string message, `Hello from STM32 NUCLEO`, to a topic called `/chatter`. You can display this string on a PC by running the ROS serial server.
- Click on the Compile button to download the binary file, which can be copied to the board. Plug the board to your PC, and you will see a flash drive of the board. You can copy the downloaded binary file to the flash storage, as shown here:

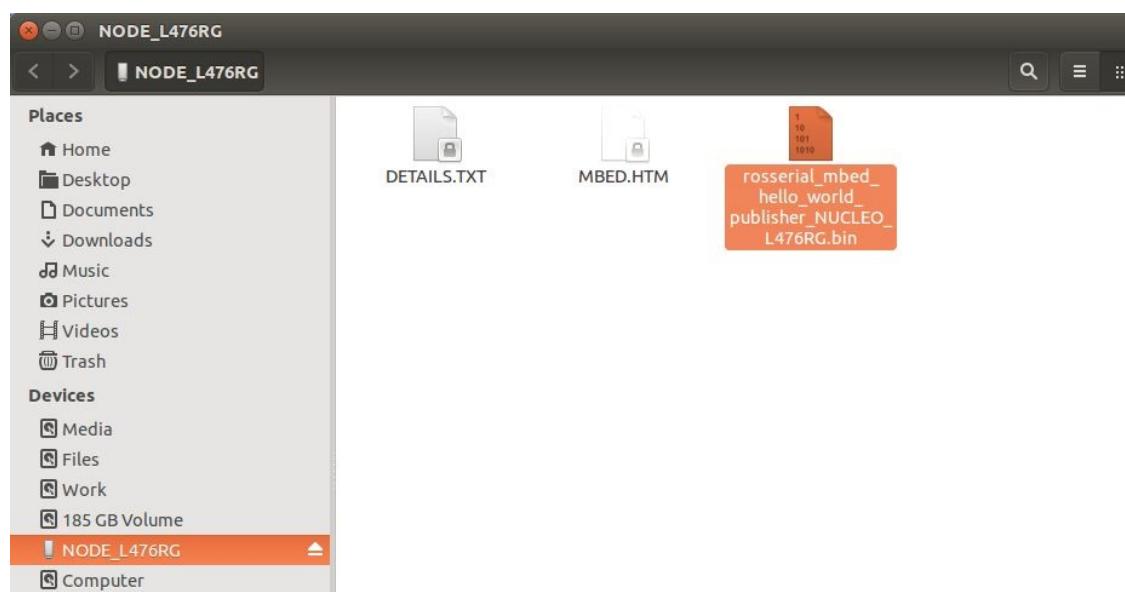


Figure 14: Binary file on flash drive

- When we copy the binary file, the board will automatically start running it. Now, the procedures have been completed. Just start the ROS server on the PC side to display topics from the board.
- Start `roscore`:

```
| $ roscore
```

16. Start the ROS server:

```
| $ rosrun rosserial_python serial_node.py /dev/ttyACM0
```

17. Now you can echo the topic using the following command:

```
| $ rostopic echo /chatter
```

18. You will get following messages on the Terminal:

```
lentin@lentin-Aspire-4755:~$ rostopic echo /chatter
data: Hello from STM32 NUCLEO
---
```

Figure 15: String message from an STM 32 board

Interfacing Tiva C Launchpad boards with ROS using Energia

Interfacing Tiva C Launchpads in ROS is very much similar to Arduino. The IDE we are using to program Tiva C boards such as the EK-TM4C123GXL and EK-TM4C1294XL is called Energia (<http://energia.nu/>). The Energia IDE is a modified version of the Arduino IDE. The procedure to generate the ROS serial client library is the same as Arduino. We have to install a few packages on Ubuntu before we start working with the Energia ROS serial client for Energia.

The following command will install the ROS serial client library for the Energia IDE:

```
| $ sudo apt-get install ros-kinetic-rosserial-tivac
```

The following command will install the C libraries for the i386 platform. This library is required if you run Energia on 64-bit Ubuntu.

```
| $ sudo dpkg --add-architecture i386  
| $ sudo apt-get update  
| $ sudo apt-get install libc6:i386
```

After installing these packages, you can download and extract the Energia IDE. You can download the latest Energia version from <http://energia.nu/download/>. We are using Energia-018 here, and you can launch Energia by running `energia` from the extracted folder. You will get an IDE like this, which is very much like the Arduino IDE except the color:

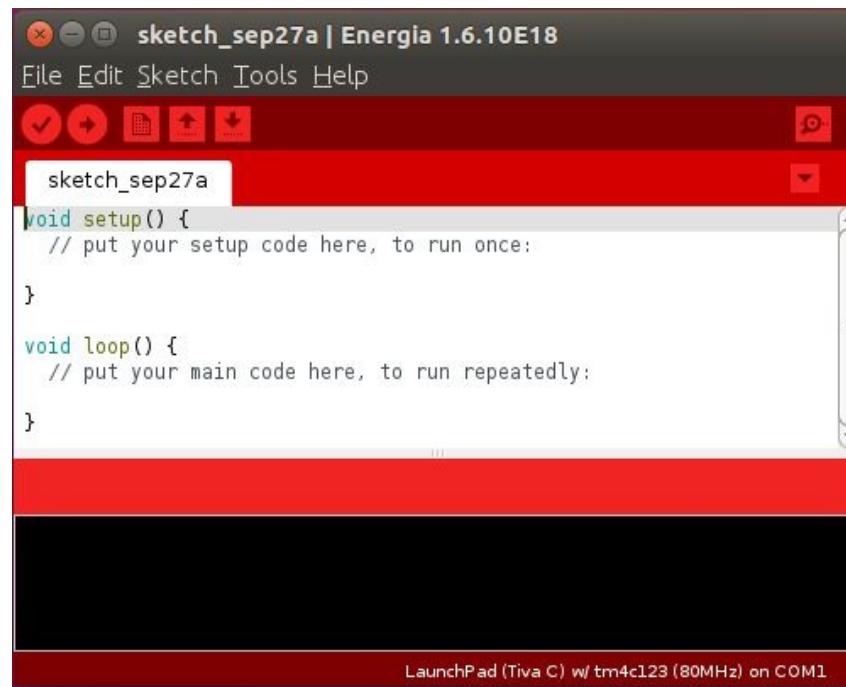


Figure 16: Energia IDE

Creating the ROS library for Energia is the same as for Arduino:

1. Go to File | Preference and set the sketchbook location.
2. Create a folder called `libraries` if one doesn't exist inside this location, and run the following command to create `ros_lib`:

```
| $ rosrun rosserial_tivac make_libraries_energia
```

3. If everything works fine, you can access the ROS examples like this:

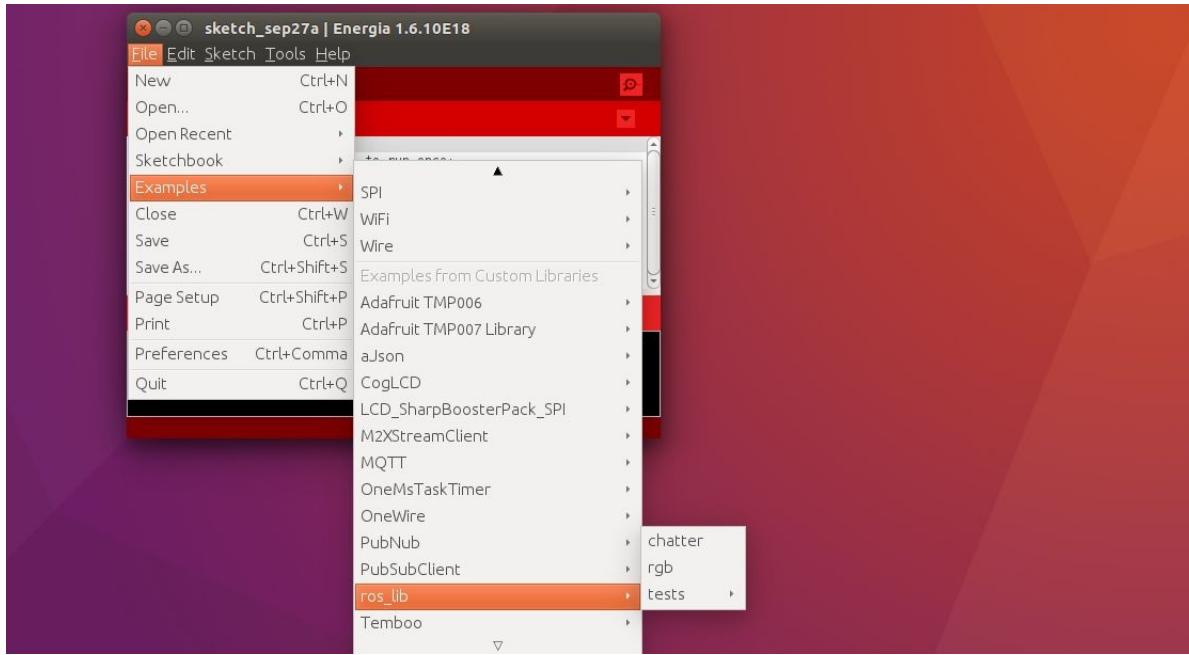


Figure 17: `ros_lib` in the Energia IDE

We can try with the `rgb` example first. The Tiva C board has a tricolor LED integrated with some port pins. Using this code, we can publish RGB values to a topic, and the board will turn on and off the LED according to the topic values. We can input values `0` or `1` for each LED. If the value is `0`, that LED will be off, and if it is `1`, it will have maximum brightness.

We can compile the code and upload it to the desired board and start the ROS serial server using the following set of commands.

Starting `roscore`:

```
| $ roscore
```

Starting the ROS serial server:

```
| $ rosrun rosserial_python serial_node.py /dev/ttyACM0
```

We will get a topic called `/led` when we start the ROS serial server, and we can publish the values to the topic using the following command:

```
| $ rostopic pub led std_msgs/ColorRGBA "r: 0.0 g: 0.0 b: 1.0 a: 1.0"
```

Here, the type of `/led` topic is `std_msgs/ColorRGBA`, and `r`, `g`, and `b` correspond to red, green, and blue, and `A` is for alpha or transparency. We are not using the alpha value.

We have seen how to make a controller board as an ROS node, and now we will see how to run ROS on a single-board computer.

Running ROS on Raspberry Pi and Odroid boards

As we discussed earlier, Raspberry Pi and Odroid boards work like a PC. We can install customized Linux on each board and install ROS on it. There are two methods to run ROS on these boards. We can either install a fresh version of a Linux OS on it and install ROS from scratch or download a prebuilt image of the OS with ROS. The first option is a long procedure, and it will take a while to build ROS on Linux. You can follow the procedure at <https://goo.gl/LvW2ZN> to install ROS from scratch. In this section, we are dealing with ROS installation from a prebuilt binary.

Here is the link to download Raspberry Pi 2 images with ROS preinstalled:

<http://www.mauriliodicicco.com/raspberry-pi2-ros-images/>

Also, you can download Odroid-ROS images from the following links:

<http://forum.odroid.com/viewtopic.php?f=112&t=11994>

You can burn the OS to an SD card using the following tools:

On Windows, you can use Win32DiskImager, which can be downloaded from the following link:

<https://sourceforge.net/projects/win32diskimager/>

For Odroid, we need a customized version of Win32DiskImager, and it can be downloaded from the following link:

http://dn.odroid.com/DiskImager_ODROID/Win32DiskImager-odroid-v1.3.zip

This is what Win32DiskImage looks like in Odroid:

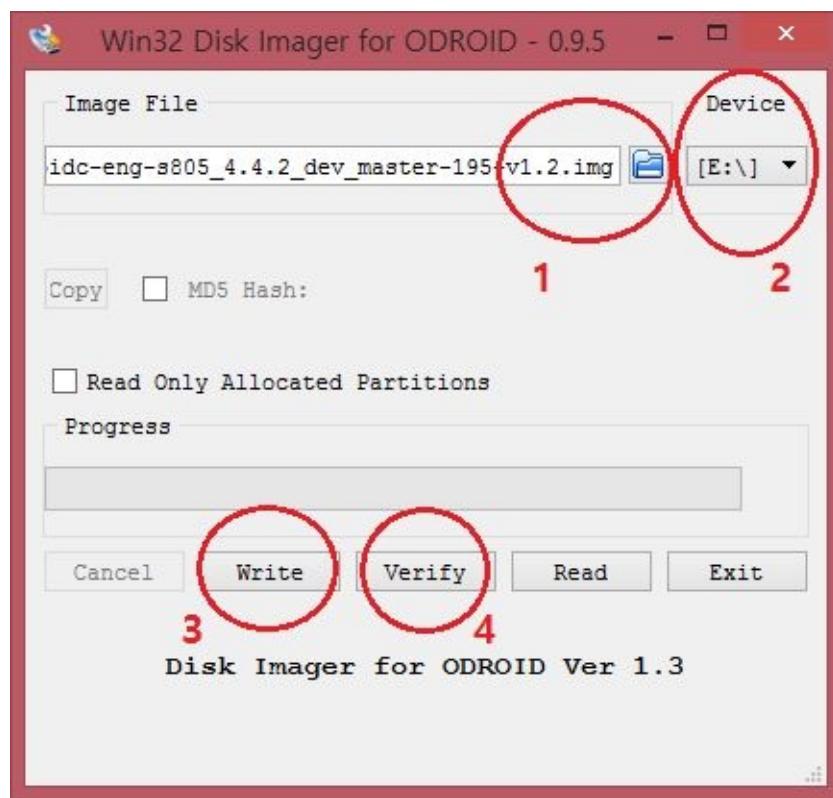


Figure 18: Win32DiskImager for Odroid

In Linux, you can use a tool called **dd** (Disk Dump); the following command helps you install OS images to an SD card:

```
| $ sudo apt-get install pv
```

The **pv** tool can help you monitor the progress of this operation:

```
| $ dd bs=4M if=image_name.img | pv | sudo dd of=/dev/mmcblk0
```

Here, `image_name.img` is the OS image name, and `/dev/mmcblk0` is the SD card reader device.

Boot the board from the SD card and check whether the board is booting properly. If it is, we can communicate with the board using Wi-Fi or wired LAN with the PC.

Now let's look at the methods to connect a single board computer to your PC.

Connecting Raspberry Pi and Odroid to PC

We can connect the RPi and Odroid boards in two ways to a PC. One is through a router in which both devices are on same network, or we can directly connect to a PC without a router. The connection through a router is simple and straightforward. Each device will get an IP address, and we can communicate with each device using it. But using direct communication, there is no IP assigned; we can do it using a Wi-Fi hotspot or wired LAN hotspot from a PC.

The following is the procedure to create a wired hotspot on Ubuntu for interfacing these boards:

1. Click on Edit Connection... from the network option in Ubuntu, as shown in the following figure. Click Add button to create a new connection.

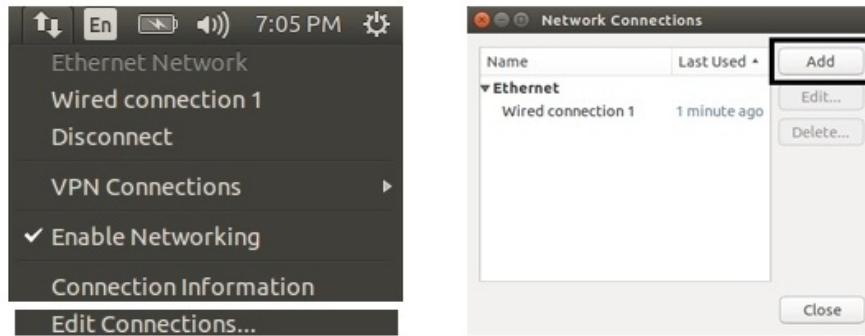


Figure 19: Creating a new network connection in Ubuntu

2. Create a new Ethernet connection, name the connection Share, and in the connection settings, change the IPV4 setting to Shared to other Computers, as shown here:

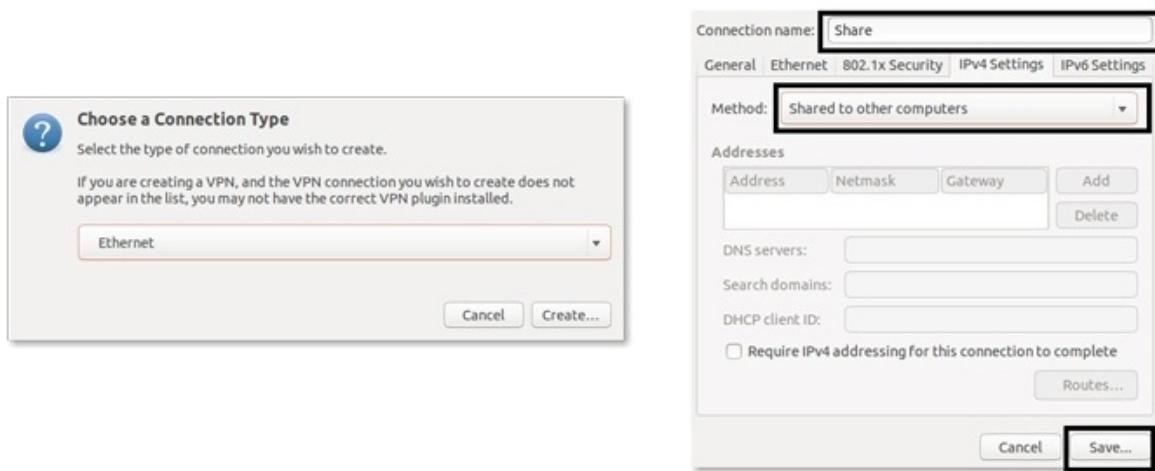


Figure 20: Creating a new Ethernet connection in Ubuntu

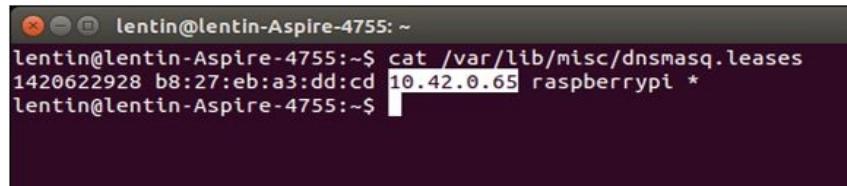
3. After creating the connection, you can plug the micro SD card to the board and boot the device; also, connect the wired LAN cable from the board to the PC.
4. When the board boots up, it will automatically connect to the Share network. If it is not connecting, you can manually click on the Share network name to connect to it. When it is connected, it means the

board has an IP address and the PC can communicate with the board using this IP; also, the important thing is that the PC is sharing its Internet connection if it has one.

5. But how do we find the IP of a board that is connected to a PC? There is a way to find out. The following command will unveil the IP:

```
| $ cat /var/lib/misc/dnsmasq.leases
```

6. The `dnsmasq` utility is a lightweight DNS and DHCP server. We can get an active client connected to the server by looking at the file called `dnsmasq.leases`. The output of this command is as follows:



```
lentin@lentin-Aspire-4755: ~
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 21: The IP of active clients connected to dnsmasq

7. Great going! If you get the IP, you can communicate with the board using **Secure Shell (SSH)**. Here are the commands to start an SSH shell from a PC to each board:

- From PC to Raspberry Pi:

```
| $ ssh pi@ip_address_of_board
```

- The password is `raspberry`

- From PC to Odroid:

```
| $ ssh odroid@ip_adress_of_board
```

- The password is `odroid`

If everything works fine, you will get the board's shell, and you can access the ROS commands from the shell.

Controlling GPIO pins from ROS

In Arduino and other controller boards, what we have created a did was make a hardware ROS node. But RPi and Odroid are single-board computers, so we can run ROS on the board itself. We can run ROS on these two boards in three ways. We can run ROS on the same board, or we can run the ROS master on the board and connect other ROS nodes from the PC or make the PC the ROS master and make the board the client.

In this section, we are going to create a simple demo to blink an LED using ROS topics from the same board. To work with Raspberry Pi and Odroid, we have to use a library called `wiringpi`.

Here are the commands to install `wiringpi` on Raspberry Pi:

```
$ git clone git clone git://git.drogon.net/wiringPi  
$ cd wiringPi  
$ sudo ./build
```

And these are the commands to install `wiringpi` on Odroid:

```
$ git clone https://github.com/hardkernel/wiringPi.git  
$ cd wiringPi  
$ sudo ./build
```

After installing `wiringpi`, we should know the GPIO pin layout of each board in order to program it. The GPIO pin layout of the boards is as follows:

| 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 22: GPIO pin layout of Raspberry Pi

The pin layout of Odroid is similar to RPi. Here is the GPIO pin layout of Odroid C1/C2:

| ODROID-C1 40pin Layout | | | | | | | | | | |
|------------------------|--------------|--------------|-------------|--------|----|-------|--------------|--------------|----------------|-----------------------|
| | | | | | | | | | | Power Pin |
| | | | | | | | | | | Special Function |
| | | | | | | | | | | GPIO/Special Function |
| WiringPi GPIO# | Export GPIO# | ODROID-C PIN | Label | HEADER | | Label | ODROID-C PIN | Export GPIO# | WiringPi GPIO# | |
| | | | 3V3 | 1 | 2 | 5V0 | | | | |
| | | 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 | 1 | |
| 2 | 116 | GPIOX.BIT19 | #116 | 13 | 14 | GND | | | | |
| 3 | 115 | GPIOX.BIT18 | #115 | 15 | 16 | #104 | GPIOX.BIT7 | 104 | 4 | |
| | | | 3V3 | 17 | 18 | #102 | GPIOX.BIT5 | 102 | 5 | |
| 12 | 107 | MOSI | GPIOX.BIT10 | MOSI | 19 | 20 | GND | | | |
| 13 | 106 | MISO | GPIOX.BIT9 | MISO | 21 | 22 | #103 | GPIOX.BIT6 | 103 | 6 |
| 14 | 105 | SCLK | GPIOX.BIT8 | SCLK | 23 | 24 | CE0 | GPIOX.BIT20 | CE0 | 117 |
| | | | GND | 25 | 26 | #118 | GPIOX.BIT21 | 118 | 11 | |
| | | | 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 | 26 |
| 23 | 108 | | GPIOX.BIT11 | #108 | 33 | 34 | GND | | | |
| 24 | 97 | | GPIOX.BIT0 | #97 | 35 | 36 | #98 | GPIOX.BIT1 | 98 | 27 |
| | | | ADC.AIN1 | AIN1 | 37 | 38 | 1V8 | 1V8 | | |
| | | | GND | 39 | 40 | AIn0 | ADC.AIN0 | | | |

Figure 23: GPIO pin layout of Odroid

Creating a ROS package for the blink demo

We are done with installing `wiringpi`; let's create a ROS package for the LED blink demo. I hope you have already created a ROS workspace on the board. For this demo, we are connecting the LED anode to the twelfth pin of the board (first pin in `wiringpi`). The LED cathode is connected to GND.

The following figure shows the circuit of the demo. It is applicable to RPi and Odroid.

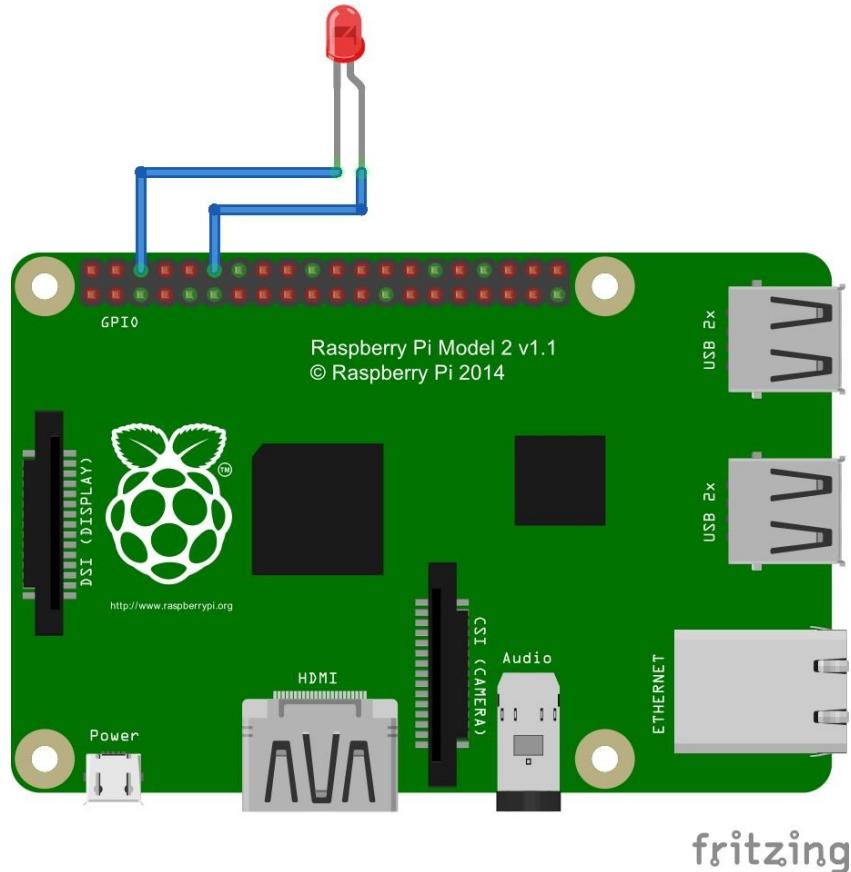


Figure 24: Board connected to an LED

Okay! Let's make a ROS package for creating a blinking ROS node. Here is the command to create a ROS package for this demo:

```
| $ catkin_create_pkg ros_wiring_example roscpp std_msgs
```

You will also get the complete package from `chapter_4_codes/ros_wiring_example`.

Create an `src` folder inside the new package, and copy the `blink.cpp` file from the existing code. The blink code is as follows:

```
#include "ros/ros.h"
#include "std_msgs/Bool.h"

#include <iostream>
#include "wiringPi.h"

//Wiring PI 1
#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,"blink_led");
    ROS_INFO("Started Odroid-C1 Blink Node");
    wiringPiSetup ();
    pinMode(LED, OUTPUT);

    ros::NodeHandle n;
    ros::Subscriber sub =
        n.subscribe("led_blink",10,blink_callback);
    ros::spin();
}

```

The preceding code will subscribe to a topic called `/led_blink`, which is a Boolean type. If the value is true, the LED will turn on, otherwise it'll be off.

The following is the `CMakeLists.txt` file for compiling the code:

```

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)

set(wiringPi_include "/usr/local/include")

include_directories(
  ${catkin_INCLUDE_DIRS}
  ${wiringPi_include}
)
LINK_DIRECTORIES("/usr/local/lib")

add_executable(blink_led src/blink.cpp)

target_link_libraries(blink_led
  ${catkin_LIBRARIES} wiringPi
)

```

After changing `CMakeLists.txt`, we can perform a `catkin_make` to build the ROS node.

If everything builds successfully, we can run the demo using the following procedure.

Running the LED blink demo on Raspberry Pi and Odroid

To run the demo, launch multiple SSH Terminals and execute each command in each Terminal.

Start `roscore`:

```
| $ roscore
```

Run the executable as root in another Terminal. We are running the node with root privilege, because GPIO handling needs root. If you are working with RPi, the username will be `pi` instead of `odroid`:

```
| $ sudo -s  
# cd /home/odroid/catkin_ws/build/ros_wiring_examples  
#./blink_led
```

You can publish `1` and `0` to `/led_blink` to test the node working from another Terminal:

```
| $ rostopic pub /led_blink std_msgs/Bool 1  
$ rostopic pub /led_blink std_msgs/Bool 0
```

To run on the Raspberry Pi, we have to perform few more steps. We have to add the following lines to the `.bashrc` folder of the root user. You can do so using the following command:

```
| $ sudo -i  
$ nano .bashrc
```

Add the following lines to the `.bashrc` file:

```
| source /opt/ros/<ros_version>/setup.sh  
source /home/pi/catkin_ws/devel/setup.bash  
export ROS_MASTER_URI=http://localhost:11311
```

Questions

- What are ROS serial client libraries?
- What are the functions of a ROS serial server?
- What are mbed and Energia?
- What are the functions of `wiringpi`?

Summary

In this chapter, we dealt with ROS interfacing of embedded controller boards and single-board PCs. We started by discussing popular controller boards, such as Arduino, STM 32-based boards and Tiva C boards. In the single-board computer category, we went through Raspberry Pi and Odroid. After discussing each board, we learned about interfacing ROS with controllers and single computers. We covered LDR interfacing with the Arduino, Hello World example on the STM 32, and RGB demo on the Tiva C Launchpad. For single-board computers, we created a basic LED blink demo using ROS.

In the next chapter, we will discuss teleoperating a robot using hand gestures.

Teleoperate a Robot Using Hand Gestures

As you all know, robots can be controlled mainly in the following modes:

- **Manual:** In manual control, the robot is controlling manually by a human. The controlling is done using a remote controller or teach pendant.
- **Semiautonomous:** The semiautonomous robot will have both manual and autonomous control. For simple task, it can work autonomously but in complex task it may change its mode to manual.
- **Fully autonomous:** An autonomous robot has complete control over its action and can think for itself. It can learn and adapt, and very much everything is controlled by the robot itself.

We can choose the model of robot control based on our application. In this chapter, we are mainly discussing implementing a manual robot control; we can call it distance control or teleoperation. In teleoperation, the robot and human can be far apart, and the operator may not able to see the real robot moving but may get some visual feedback. Rather than manual control, some teleoperated robots have different levels of autonomy integrated. The robots can take action entirely by following the commands sent by the operator or only receiving high-level commands and taking care of other stuff autonomously.

We are going to be discussing a project to teleoperate a robot using hand gestures. The major component that we are using to detect the gestures is an **inertial measurement unit (IMU)**. The IMU is fitted into a hand glove, and with specific hand gestures, we can move or rotate the robot. The project uses the Arduino-ROS combination to compute IMU orientation and send it to PC. A ROS node runs on the PC, which maps the orientation data into twist messages (`geometry_msgs/Twist`), which is the command velocity of the robot. We will look at more analysis of the project design in the upcoming sections.

We are going to discuss following topics in this chapter:

- Teleoperating a TurtleBot using a keyboard
- Gesture teleop: teleoperating using hand gestures
- Setting up the project
- Interfacing the IMU MPU-9250 with the Arduino and ROS
- Visualizing the IMU TF on Rviz
- Converting IMU data into twist messages
- Integration and final run
- Teleoperating using an Android phone

Teleoperating ROS Turtle using a keyboard

This section is for beginners who haven't worked with teleoperation in ROS yet. In this section, we will see how to teleoperate a robot manually using a keyboard. Using a keyboard, we can translate and rotate the robot. One of the basic example to demonstrate keyboard teleoperation is ROS `turtlesim`.

The following commands launch `turtlesim` with keyboard teleoperation. You can run each command on separate Terminals.

Run `roscore`:

```
| $ roscore
```

Run a `turtlesim` node using the following command. This command will launch the `turtlesim` window:

```
| $ rosrun turtlesim turtlesim_node
```

Run the keyboard teleoperation node. We can change the turtle's position by pressing arrow keys on the keyboard:

```
| $ rosrun turtlesim turtle_teleop_key
```

The screenshot of the moving turtle using arrow keys is shown here:

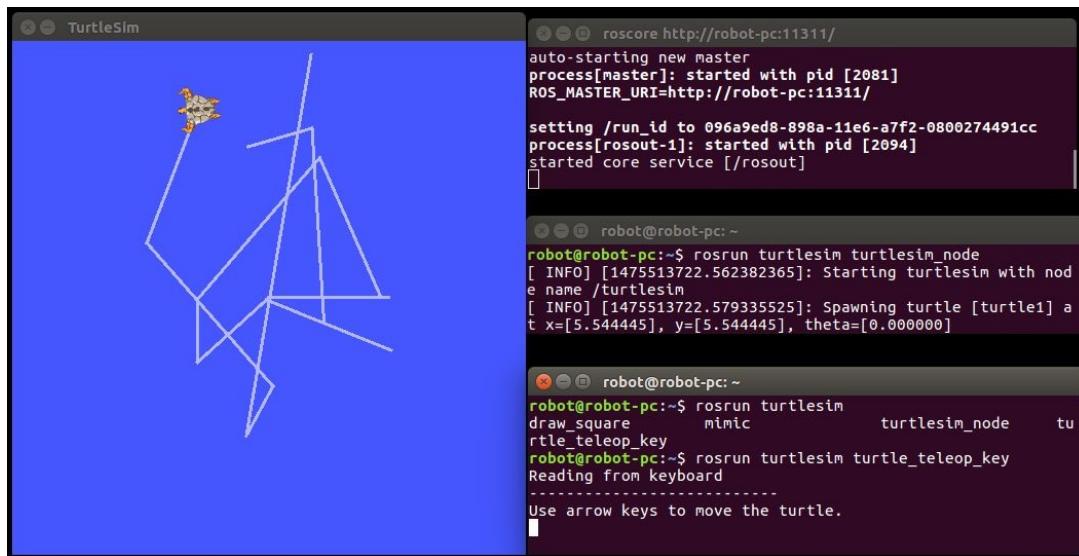


Figure 1: Turtlesim keyboard teleoperation

In ROS, most of the robot packages are bundled with a teleop node for manual control of the robot. This control can either be through keyboard, joystick, or some other input device.

Teleoperating using hand gestures

The idea of this project is converting IMU orientation into the linear and angular velocity of the robot. Here is the overall structure of this project.

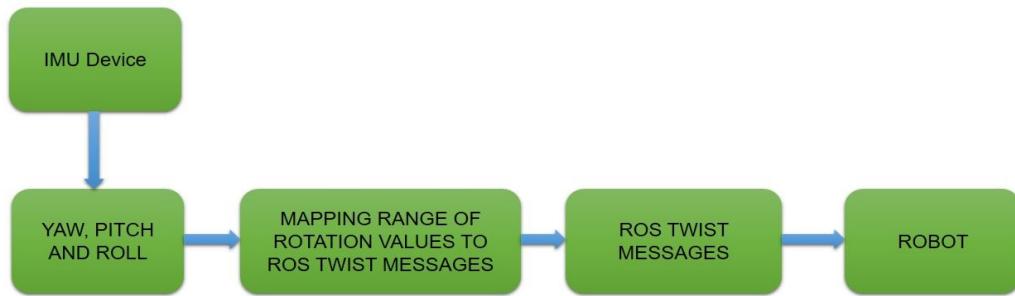


Figure 2: Basic structure of the gesture teleop project

For the IMU device, we are using an IMU called MPU-9250 (<https://www.invensense.com/products/motion-tracking/9-axiis/mpu-9250/>). The IMU will interface with an Arduino board using the I2C protocol. The orientation values from the IMU are computed by the Arduino and send to PC through the `rosserial` protocol. The orientation values are received on the PC side as ROS topics and converted into twist messages using a ROS node.

Here is the project block diagram with the **MPU 9250** and Arduino board:

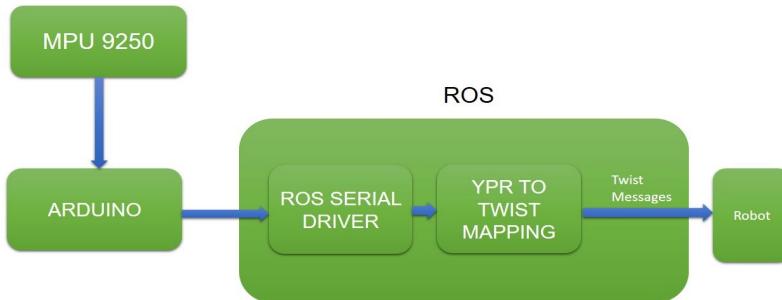


Figure 3: Functional block diagram of the robot teleop project

We are using a hand glove in which an Arduino board is fixed in the palm area and an MPU-9250 is fixed on the finger area, as shown in the following image:



Figure 4: Hand glove with Arduino and MPU-9250

There are four kinds of arm gestures used in this project:

- Vertical elbow rotation:
 - Clockwise
 - Anticlockwise
- Up pitch movement of hand
- Down pitch movement of hand

Vertical elbow rotation is mapped to the rotation of the robot in the z direction, and up and down pitch movement of the hand is mapped into forward and reverse movement of the robot. Here's a depiction of how the movements of our arm and motion of the robot are mapped:

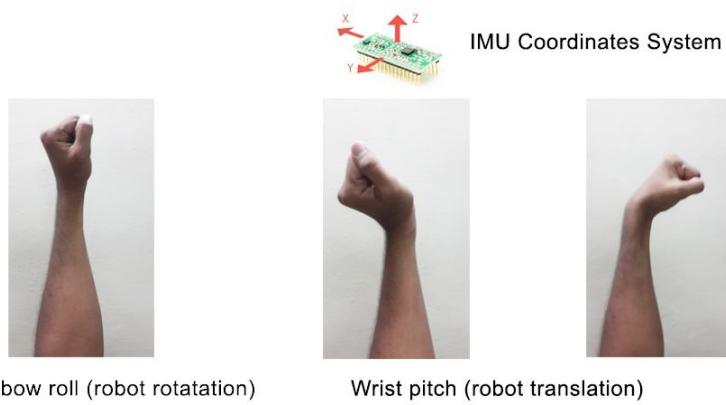


Figure 5: Hand gestures and corresponding motion mapping

The mapping goes like this: the robot will stop the movement when the IMU in the hand is horizontal to the ground. We can call this the home position. In the home position, the robot will not move. When the elbow starts rotating about the vertical axis, the robot's velocity will be such that it will rotate along the z axis. The robot's rotation will depend on how many degrees the elbow is rotated. The robot will keep on rotating until the IMU reaches the home position. For moving the robot forward and backward, we can pitch the hand as shown in the preceding figure. If the hand pitching is upward, the resultant robot velocity can move the robot backward, and vice versa.

Here is the coordinate-system representation of the IMU and movement happening along each axis and the motion assigned to each hand movement:

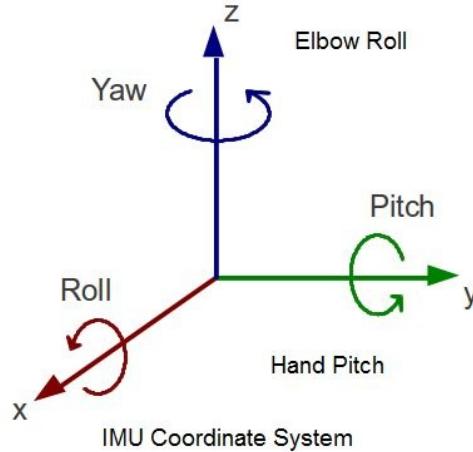


Figure 6: Hand gestures and its motion mapping

The following table will give you a quick idea about the mapping of hand gestures and robot motion:

| Hand movement | Robot motion |
|--|-----------------------------|
| Elbow rotation clockwise (IMU yaw) | Robot rotates clockwise |
| Elbow rotation anticlockwise (IMU yaw) | Robot rotates anticlockwise |
| Hand pitch upward (IMU pitch) | Robot moves backward |
| Hand pitch downward (IMU pitch) | Robot moves forward |

Note that we are using two components of rotation from the IMU, which are yaw and pitch. The yaw rotation of the IMU is facing upward, and pitch rotation is facing toward you. When we rotate the elbow, the yaw value of the IMU changes, and when we pitch the hand, the pitch value of the IMU changes. These changes will be converted to the linear and angular velocity of the robot.

Setting up the project

Let's set up the project. To finish this project, you may need the following electronic components. You can see the component name and the link to buy it from the following table:

| No | Name | Link |
|----|------------------------------------|---|
| 1 | Arduino Mega - 2560 with USB cable | https://www.sparkfun.com/products/11061 |
| 2 | MPU - 9250 breakout | https://amzn.com/B00OPNUO9U |
| 3 | Male to Female jumper wires | https://amzn.com/B00PBZMN7C |
| 4 | Hand glove | https://amzn.com/B00WH4NXLA |

You can use any Arduino having I2C communication. You can also use MPU - 6050 /9150, both of which are compatible with this project. A few words about the MPU - 9250 IMU: it is a 9-axis motion tracking device consisting of a gyro, accelerometer, and compass. MPU - 6050/9150/9250 models have an inbuilt **Digital Motion Processor (DMP)**, which can fuse the accelerometer, gyro, and magnetometer values to get accurate 6DOF/9DOF motion components. In this project, we are only taking the yaw and pitch rotation components.



If you want to learn more about I2C, check out the following link: <https://learn.sparkfun.com/tutorials/i2c> Read more about the MPU series: <https://www.invensense.com/technology/motion/>

Interfacing the MPU-9250 with the Arduino and ROS

So the first step in this project is to interface the IMU to the Arduino to get the rotation values and send those values to ROS. We're essentially making an Arduino-ROS node that is receiving IMU values and publishing the yaw, pitch, and roll as well as the **transformation (TF)** corresponding to the IMU movement as ROS topics.

The following figure shows the interfacing of IMU with the Arduino. The IMU is interfaced using the I2C protocol:

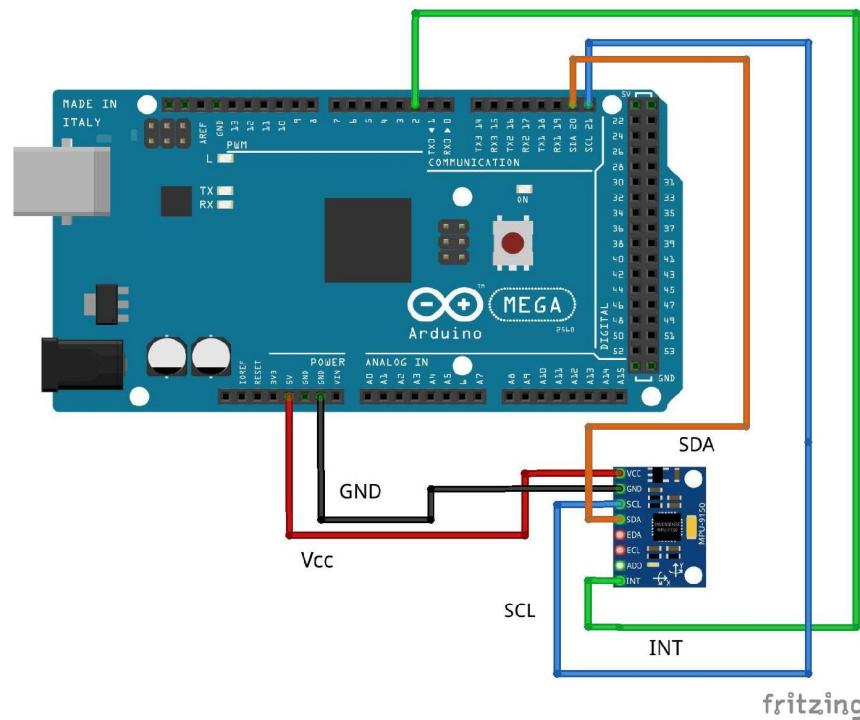


Figure 7: Interfacing MPU 9250/9150/6050 with Arduino

The connection from Arduino to MPU-9250 is shown in this table:

| Arduino pins | MPU - 9250 pins |
|--------------|-----------------|
| 5V | VCC |
| GND | GND |
| SCL(21) | SCL |
| SDA(20) | SDA |
| Digital PIN2 | INT |

To start working on IMU values in ROS, we have to create a ROS-Arduino node that is receiving IMU values and send it as ROS topics. I hope you have set up the Arduino IDE in your system. For running this code, you will need the Arduino library for the MPU - 9250. Note that you can use the MPU - 9150 library for working with this IMU, and you can clone that library's files using the following command:

```
| $ git clone https://github.com/sparkfun/MPU-9150_Breakout/tree/master/firmware
```

Copy `firmware/I2Cdev` and `MPU6050` into the `arduino_sketch_location/libraries` folder. The sketchbook location can be obtained from the File | Preferences IDE option.

Once you've copied both of these folders, you can compile the ROS-Arduino node. You can open the code from `chapter_5_codes/MPU9250_ROS_DMP`. Just try to compile the code and check whether it's working or not. I hope that you have already set the Arduino ROS serial client library , which is `ros_lib`. The entire procedure was mentioned in *Chapter 22, Controlling Embedded Boards Using ROS*.

The following figure shows the flowchart of the complete code. We'll go through a detailed explanation of code after this.

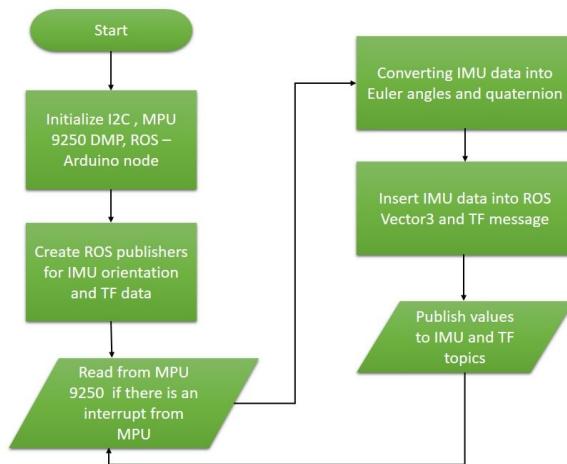


Figure 8: Flowchart of Arduino-ROS node

The Arduino-IMU interfacing code

Let's discuss the code from the beginning. The following Arduino headers help us read IMU values using the I2C protocol. The `MPU6050_6Axis_MotionApps20.h` header has functions to enable DMP and retrieve values from it.

```
| #include "Wire.h"  
| #include "I2Cdev.h"  
| #include "MPU6050_6Axis_MotionApps20.h"
```

The following line of code will create an `MPU6050` handle, which can be used for the MPU-9250. We can use this object to initialize and retrieve values from the IMU.

```
| MPU6050 mpu;
```

As you know, we have to include `ros.h` to access ROS serial client APIs. We are also including `vector3.h`, which has the definition of the `Vector3` ROS message. This message can carry three values of orientation. The `tf/transform_broadcaster.h` header has TF broadcaster classes, which basically send transforms of IMU values with respect to a fixed frame:

```
| #include <ros.h>  
| #include <geometry_msgs/Vector3.h>  
| #include <tf/transform_broadcaster.h>
```

After defining headers, we have to define handles of the TF message and broadcaster, as given here:

```
| geometry_msgs::TransformStamped t;  
| tf::TransformBroadcaster broadcaster;
```

In the next line of code, we are creating a `NodeHandle`, which essentially helps us subscribe to and publish ROS topics like a normal ROS node:

```
| ros::NodeHandle nh;
```

To hold the orientation values, which are yaw, pitch, and roll, we are creating a `Vector3` ROS message. This message is published by the Arduino node on a topic named `/imu_data`.

```
| geometry_msgs::Vector3 orient;
```

The following line of code creates a publisher object for the `/imu_data` topic. We are publishing the orientation using this object.

```
| ros::Publisher imu_pub("imu_data", &orient);
```

The `frameid` value is `/base_link`, which is static, and the `child` frame is `/imu_frame` which moves according to the IMU data.

```
| char frameid[] = "/base_link";  
| char child[] = "/imu_frame";
```

These are variables to hold orientation values, such as quaternion, gravity vector, and yaw, pitch, and roll:

```
Quaternion q;
VectorFloat gravity;
float ypr[3];
```

Here is the interrupt-detection routine, for whenever data is ready to be read from the IMU. The routine basically sets the `mpuInterrupt` variable as `true`.

```
volatile bool mpuInterrupt = false;
void dmpDataReady() {
    mpuInterrupt = true;
}
```

Next is the `setup()` function of Arduino, which does several I2C initializations for the Arduino, ROS node handler, TF broadcaster, ROS publisher, MPU object, and DMP inside MPU:

```
void setup() {
    Wire.begin();
    nh.initNode();
    broadcaster.init(nh);
    nh.advertise(imu_pub);
    mpu.initialize();
    devStatus = mpu.dmpInitialize();
```

If DMP is initialized, we can enable it and attach an interrupt on the Arduino's second digital pin, which is the first interrupt pin of the Arduino. Whenever data is ready to be read from buffer, the IMU will generate an interrupt. The code also checks the DMP status and sets a variable to check whether DMP is ready or not. This will be useful while executing the `loop()` function. We are also using a variable called `packetSize` to store the MPU buffer size.

```
if (devStatus == 0) {
    mpu.setDMPEnabled(true);
    attachInterrupt(0, dmpDataReady, RISING);
    mpuIntStatus = mpu.getIntStatus();
    dmpReady = true;
    packetSize = mpu.dmpGetFIFOPacketSize();
}
```

Inside the `loop()` function, the code checks whether `dmpReady` is true or not. If it is not true, that means DMP is not initialized, so it will not execute any code. If it is ready, it will wait for interrupts from the MPU.

```
if (!dmpReady) return;
while (!mpuInterrupt && fifoCount < packetSize) {
    ;
}
```

If there is an interrupt, it will go to the `dmpDataReady()` interrupt-detection routine and set the `mpuInterrupt` flag as `true`. If it is true, then the previous `while` loop will exit and start running the following code. We are resetting the `mpuInterrupt` flag to false, reading the current status of the MPU, and retrieving the **first-in first-out (FIFO)** count. FIFO is basically a buffer, and the first entry to the buffer will be processed first.

```
mpuInterrupt = false;
mpuIntStatus = mpu.getIntStatus();
fifoCount = mpu.getFIFOCount();
```

After reading the status and FIFO count, we can reset the FIFO if an overflow is detected. Overflows can happen if your code is too inefficient.

```
if ((mpuIntStatus & 0x10) || fifoCount == 1024) {
    mpu.resetFIFO();
```

If the data is ready, we will again compare the FIFO buffer size and the DMP packet size; if equal, FIFO data will be dumped into the `fifoBuffer` variable.

```
else if (mpuIntStatus & 0x01) {
    while (fifoCount < packetsize) fifoCount =
        mpu.getFIFOCount();
    mpu.getFIFOBytes(fifoBuffer, packetsize);
    fifoCount -= packetsize;
```

After storing the DMP data in the buffer, we can extract the rotation components, such as quaternion, gravity vector, and Euler angle.

```
mpu.dmpGetQuaternion(&q, fifoBuffer);
mpu.dmpGetGravity(&gravity, &q);
mpu.dmpGetYawPitchRoll(ypr, &q, &gravity);
```

We need to get the Euler angle in degrees, and it is going to be published in the `/imu_data` topic. Here is the code for doing it. The `ypr` value we're getting from the MPU object will be in radians, which should be converted to degree using the following equations:

```
orient.x = ypr[0] * 180/M_PI;
orient.y = ypr[1] * 180/M_PI;
orient.z = ypr[2] * 180/M_PI;
imu_pub.publish(&orient);
```

Here is how we'll publish the TF data. We have to insert the frame, quaternion values, and time stamping to the TF message headers. Using the TF broadcaster, we can publish it.

```
t.header.frame_id = frameid;
t.child_frame_id = child;
t.transform.translation.x = 1.0;
t.transform.rotation.x = q.x;
t.transform.rotation.y = q.y;
t.transform.rotation.z = q.z;
t.transform.rotation.w = q.w;
t.header.stamp = nh.now();
broadcaster.sendTransform(t);
```

We have to call `nh.spinOnce()` to process each operation we have performed using ROS APIs, so the publishing and subscribing operations are performed only while calling the `spinOnce()` function. We are also blinking the onboard LED to indicate the program activity.

```
nh.spinOnce();
delay(200);
blinkState = !blinkState;
digitalWrite(LED_PIN, blinkState);
delay(200);
```

That is all about the ROS-Arduino node. Now what you can do is compile and upload this code to Arduino. Make sure that all other settings on the Arduino IDE are correct.

After uploading the code to the Arduino, the remaining work is on the PC side. We have to run the ROS serial server node to obtain the Arduino node topics. The first step to verify the IMU data from Arduino is by visualizing it. We can visualize the IMU data by observing the TF values in Rviz.

Visualizing IMU TF in Rviz

In this section, we are going to visualize the TF data from Arduino on Rviz. Here's the procedure to do that.

Plug the Arduino to the PC and find the Arduino's serial port. To get topics from the Arduino-ROS node, we should start a ROS serial server on the PC, listening on the Arduino serial port. We did this in *Chapter 22, Controlling Embedded Boards Using ROS*. Still, let's look at the commands again in this section too.

Starting `roscore` first:

```
| $ roscore
```

Starting the ROS serial server:

```
| $ rosrun rosserial_python serial_node.py /dev/ttyACM0
```

You can get the following topics when you run the previous node:

```
lentin@lentin-Aspire-4755:~$ rostopic list
 imu_data
 /rosout
 /rosout_agg
 /tf
lentin@lentin-Aspire-4755:~$
```

Figure 9: Listing ROS topics from Arduino

You can simply echo these topics, or visualize the TF data on Rviz. You can run Rviz using the following command. The `base_link` option is the fixed frame, and we can mention that on the command line itself.

```
| $ rosrun rviz rviz -f base_link
```

The Rviz window will pop up, and if there is no TF option on the left-hand side of Rviz, add it from the Add | TF.

You may get a visualization like shown here, where `imu_frame` will move according to the rotation of the IMU:

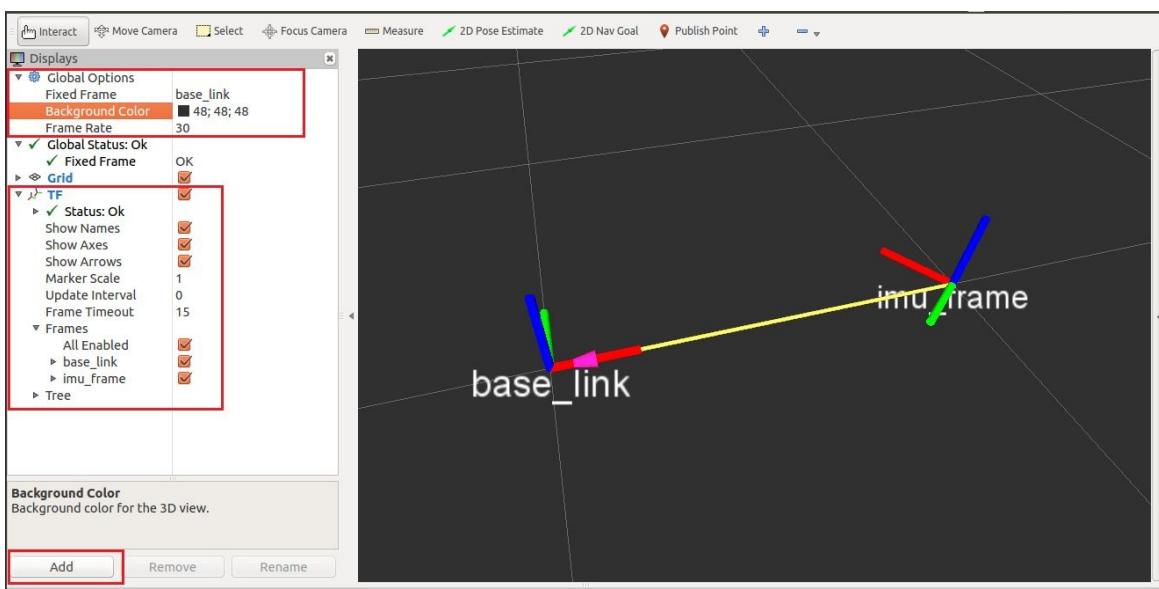


Figure 10: Visualizing IMU data in Rviz

Converting IMU data into twist messages

If you are able to the visualization in Rviz, you are done with the interfacing. The next step is to convert IMU orientation into command velocity as ROS twist messages. For this, we have to create a ROS package and a Python script. You can get this package from `chapter_5_codes/gesture_teleop`; look for a script called `gesture_teleop.py` from the `gesture_teleop/scripts` folder.

If you want to create the package from scratch, here is the command:

```
| $ catkin_create_pkg gesture_teleop rospy roscpp std_msgs sensor_msgs geometry_msgs
```

Now let's look at the explanation of `gesture_teleop.py`, which is performing the conversion from IMU orientation values to `twist` commands.

In this code, what we basically do is subscribe to the `/imu_data` topic and extract only the yaw and pitch values. When these values change in the positive or negative direction, a step value is added or subtracted from the linear and angular velocity variable. The resultant velocity is sent using ROS twist messages with a topic name defined by the user.

We need the following modules to perform this conversion. As you know, `rospy` is a mandatory header for a ROS Python node.

```
| import rospy
| from geometry_msgs.msg import Twist
| from geometry_msgs.msg import Vector3
```

After importing the modules, you will see the initialization of some parameters; these parameters are keeping in a file called `gesture_teleop/config/teleop_config.yaml`. If the node can't retrieve parameters from the file, it will load the default values mentioned in the code.

Here is the subscriber for the `/imu_data` topic, in which the topic name is defined as a variable. The callback function is called `Get_RPY` and the message type is `vector3`.

```
| rospy.Subscriber(imu_topic,Vector3,Get_RPY)
```

The `Get_RPY` simply computes the delta value of the yaw and pitch values of the IMU data and sends those values along with the yaw value to another function called `Send_Twist()`:

```
| def Get_RPY(rpy_data):
|     global prev_yaw
|     global prev_pitch
|     global dy,dp
|     dy = rpy_data.x - prev_yaw
|     dp = rpy_data.y - prev_pitch
|     Send_Twist(dy,dp,rpy_data.y)
|     prev_yaw = rpy_data.x
|     prev_pitch = rpy_data.y
```

The following code is the definition of `Send_Twist()`. This is the function generating the twist message from the orientation values. Here, the linear velocity variable is `control_speed` and angular velocity variable is

`control_turn`. When the pitch value is very less and change in yaw value is zero, the position is called home position. In home position, the IMU will be horizontal to the ground. In this position, the robot should stop its movement. We are assigning both the speeds as zero at this condition. In other cases, the control speed and control turn is computed up to the maximum or minimum speed limits. If the speed is beyond the limit, it will switch to the limiting speed itself. The computed velocities are assigned to a twist message header and published to the ROS environment.

```
def Send_Twist(dy,dp,pitch):
    global pub
    global control_speed
    global control_turn
    dy = int(dy)
    dp = int(dp)
    check_pitch = int(pitch)
    if (check_pitch < 2 and check_pitch > -2 and dy == 0):
        control_speed = 0
        control_turn = 0
    else:
        control_speed = round(control_speed + (step_size * dp),2)
        control_turn = round(control_turn + ( step_size * dy),2)
        if (control_speed > high_speed):
            control_speed = high_speed
        elif (control_turn > high_turn):
            control_turn = high_turn
        if (control_speed < low_speed):
            control_speed = low_speed
        elif (control_turn < low_turn):
            control_turn = low_turn
        twist = Twist()
        twist.linear.x = control_speed; twist.linear.y = 0;
        twist.linear.z = 0
        twist.angular.x = 0; twist.angular.y = 0; twist.angular.z
    = control_turn
    pub.publish(twist)
```

That is all about the converter node, which converts orientation data to twist commands. Next is the configuration file of the `gesture_teleop.py` node. This node stores the essential parameters of the converter node. The file is named `teleop_config.yaml` and placed in the `gesture_teleop/config/` folder. The file consists of the IMU data topic, limits of linear and angular velocity, and step size.

```
imu_topic: "/imu_data"
low_speed: -4
high_speed: 4
low_turn: -2
high_turn: 2
step_size: 0.02
```

Integration and final run

We are almost done! But how to test this teleop tool? We can create some launch file that can start all these nodes and work with some robot simulation. The `gesture_teleop/launch` folder has three launch files. Let's take a look at them.

The `gesture_teleop.launch` file is a generic launch file that can be used for any robot. The only thing we need to edit is the command velocity topic. Here is the definition of this launch file:

```
<launch>
  <param name="teleop_topic" value="/cmd_vel"/>
  <rosparam command="load" file="$(find
    gesture_teleop)/config/teleop_config.yaml"/>
  <node name="rosserial_server_node" pkg="rosserial_python"
    type="serial_node.py" args="$(arg port)" output="screen"/>
  <node name="gesture_teleop_node" pkg="gesture_teleop"
    type="gesture_teleop.py" output="screen"/>
</launch>
```

This launch file defines `teleop_topic`. You can change the command velocity topic name according to each robot's configuration. It also loads the config file called `teleop_config.yaml`. Then, start the ROS serial server node and then the gesture teleop node.

The other two launch files are `gesture_teleop_turtlebot.launch` and `gesture_teleop_turtlebot_2D.launch`. The first launch file starts the gesture teleop of TurtleBot, which also launches the TurtleBot simulation in Gazebo, and the second launch file launches the ROS turtlesim and its gesture teleop node.

Let's start turtlesim with its gesture teleop node:

```
| $ roslaunch gesture_teleop gesture_teleop_turtlebot_2D.launch
```

You may get the turtlesim window and control the turtle using the gesture teleop:

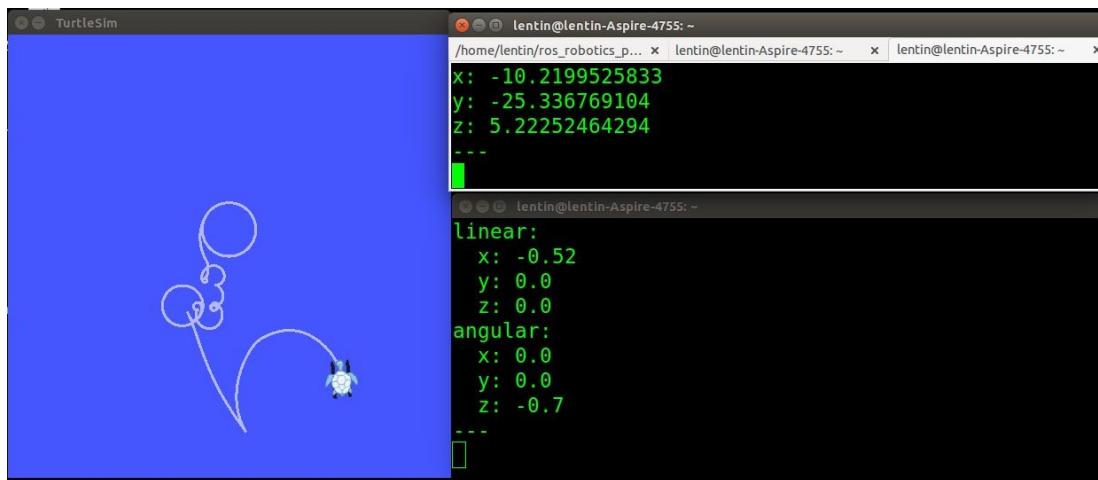


Figure 11: Gesture teleop on turtlesim

You can rotate the turtle by moving the IMU in the Z axis and can move forward and backward by pitching in the Y-axis. You can stop the robot's movement by bringing the IMU to the home position.

We can also teleop TurtleBot using the following launch file:

```
| $ roslaunch gesture_teleop gesture_teleop_turtlebot.launch
```

You may get the following simulation in Gazebo if the TurtleBot packages are already installed on your system:

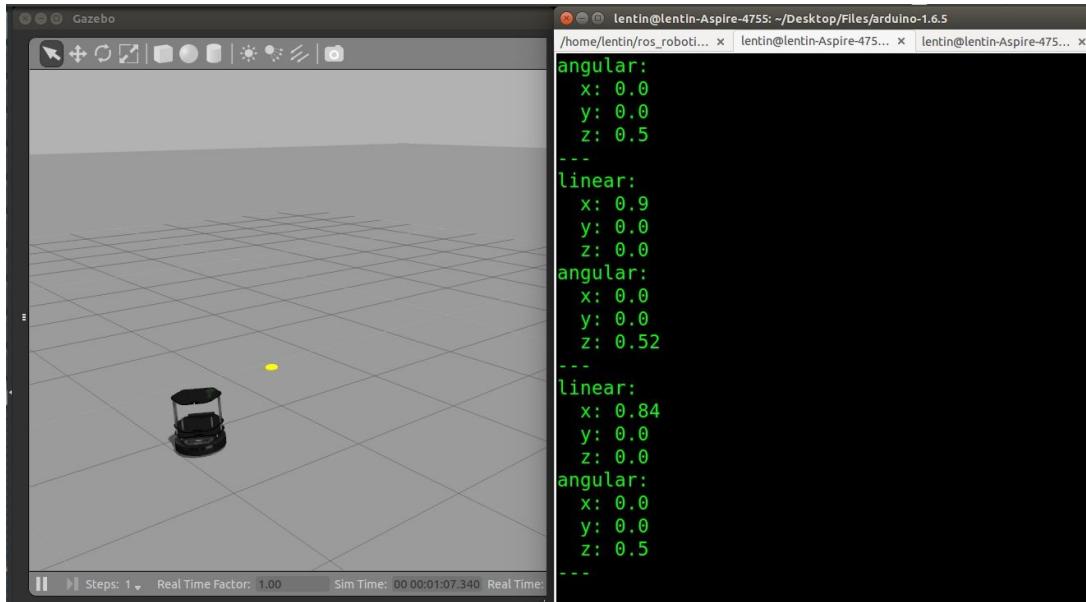


Figure 12: Gesture teleop on TurtleBot simulation.

Similar to turtlesim, we can rotate and translate TurtleBot using yaw and pitch movements.

Teleoperating using an Android phone

If it is difficult to build the previous circuit and set everything up, there is an easy way to do so with your Android phone. You can manually control either using a virtual joystick or the tilt of the phone.

Here is the Android application you can use for this:

<https://play.google.com/store/apps/details?id=com.robotca.ControlApp>.

The application's name is ROS Control. You can also search on Google Play Store for it.

Here is the procedure to connect your Android phone to a ROS environment:

Initially, you have to connect both your PC and Android device to a local Wi-Fi network in which each device can communicate with each other using IP addresses.

After connecting to the same network, you have to start `roscore` on the PC side. You can also note the IP address of the PC by entering the command `ifconfig`.

```
wlan0      Link encap:Ethernet HWaddr 94:39:e5:4d:7d:da
           inet addr:192.168.1.102 Bcast:192.168.1.255 Mask:255.255.255.0
             inet6 addr: fe80::9639:e5ff:fe4d:7dda/64 Scope:Link
                UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
              RX packets:1963 errors:0 dropped:0 overruns:0 frame:0
              TX packets:2252 errors:0 dropped:0 overruns:0 carrier:0
                collisions:0 txqueuelen:1000
               RX bytes:345822 (345.8 KB)  TX bytes:248404 (248.4 KB)
```

Figure 13: Retrieving the IP address of a PC with ifconfig

1. After obtaining the IP address of the PC, you can start the app and create a robot configuration, as shown in this figure:

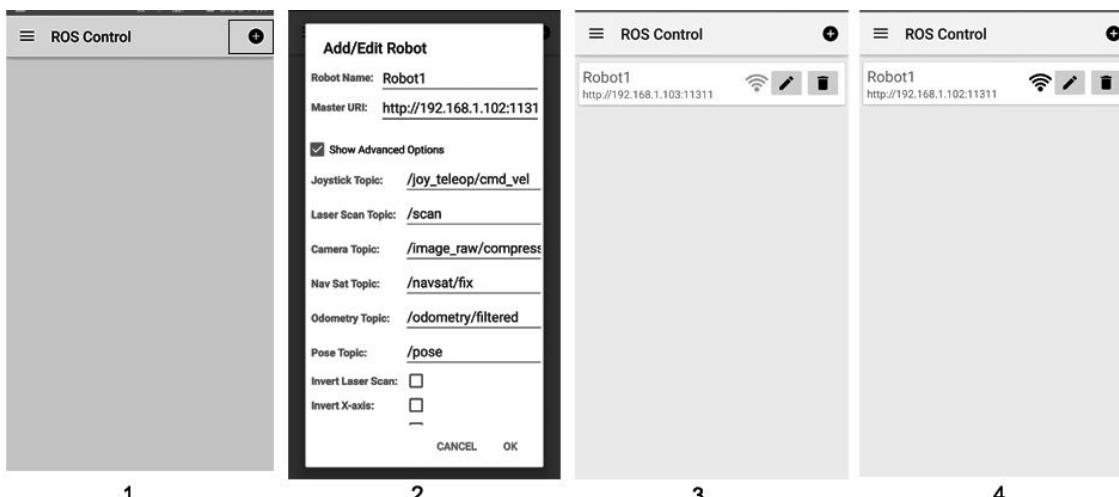


Figure 14: Configuring ROS Control app in Android

2. The + symbol in the top-right corner of the app is used to add a robot configuration in the app. Press

it and you'll see a window to enter the various topic names, Robot Name, and Master URI.

3. You have to change the Master URI from `localhost:11311` to `IP_of_PC:11311`; for example, it is `192.168.1.102.11311` in this case, which is shown in the preceding figure marked as **2**.
4. We can enter the topic name of the teleop here, so twist messages will be published to that topic. For TurtleBot, the topic name is `/cmd_vel_mux/input/teleop`. Press OK if you are done with the configuration, and you will see the third screen. In case your phone is not connected to the PC, press that configuration option and it will connect to the PC, which is shown as **4** in the figure.
5. When it connects to the PC, you will get another window, in which you can interact with the robot. Those windows are shown here:

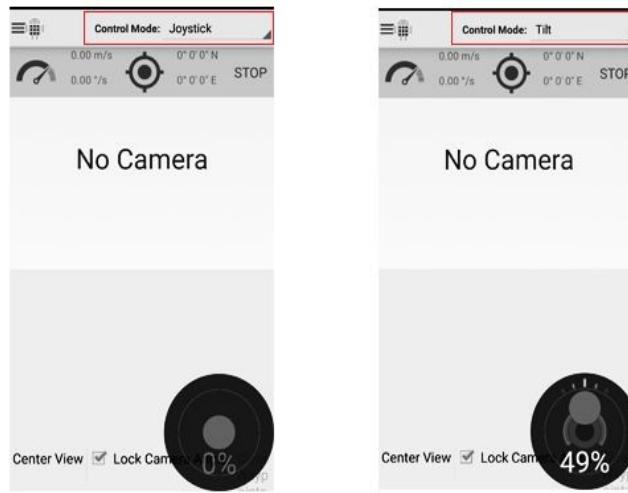


Figure 15: Controlling robot using virtual joystick and tilt of the phone.

6. The control mode can change from `Joystick` to `Tilt`, with which will work according to the tilting of phone. You can tilt the phone and change the robot's rotation and translation. Make sure you're running robot hardware or a simulation that accepts twist messages to move. You also need confirm that the topic name that you give the app is the same as the robot teleop topic.
7. After connecting your phone to the PC through the app, you will get the following topics on the PC side. You can confirm whether you're getting this before starting the robot simulation on the PC:

```
/clock  
/cmd_vel_mux/input/teleop  
/image_raw/compressed  
/navsat/fix  
/odometry/filtered  
/pose  
/rosout  
/rosout_agg  
/scan
```

Figure 16: Listing the ROS topics from the app

8. If you are getting these topics, you can start a robot simulation, such as TurtleBot, using the following command:

```
| $ roslaunch turtlebot_gazebo turtlebot_empty_world.launch
```

9. Now you can see that the robot is moving with your commands from your phone. Here is a screenshot of this operation:

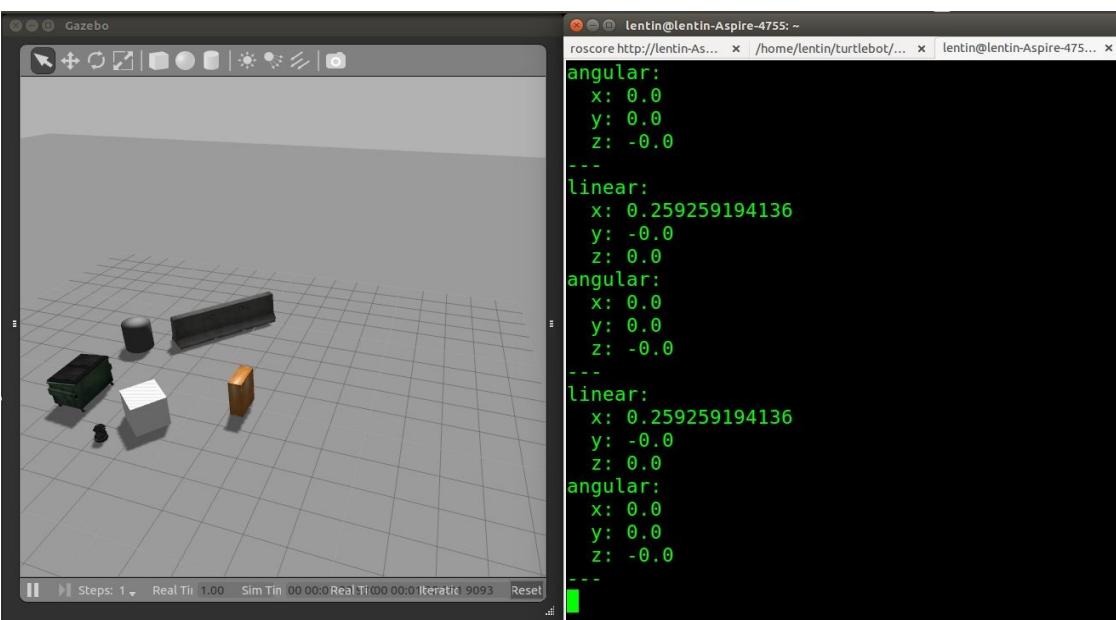


Figure 17: Controlling TurtleBot from Android app

Questions

- What are the main modes of controlling a differential drive robot?
- What is the twist message in ROS for?
- What is DMP and what is the use of DMP in this project?
- How can we teleoperate a robot from an Android phone?

Summary

This chapter was about making a gesture-based teleoperation project for a ROS-based robot. We used an IMU to detect gestures and interfaced with the Arduino to get the values from the IMU. The Arduino is interfaced with ROS using the ROS serial protocol. The PC is running a ROS node that can convert IMU orientation into linear and angular velocity and send it as a twist message. This twist message can be used in any robot just by changing the teleop topic name. We can also visualize the IMU orientation data in Rviz using TF data from Arduino. If it is too difficult to build this circuit, we can use an Android app called ROS Control that can move the robot using the inbuilt IMU on the phone.

In the next chapter, we'll be dealing with 3D object recognition using ROS.

Object Detection and Recognition

Object recognition has an important role in robotics. It is the process of identifying an object from camera images and finding its location. Using this, a robot can pick an object from the workspace and place it at another location.

This chapter will be useful for those who want to prototype a solution for a vision-related task. We are going to look at some popular ROS packages to perform object detection and recognition in 2D and 3D. We are not digging more into the theoretical aspects, but you may see short notes about the algorithm while we discuss their applications.

You will learn about the following topics:

- Getting started with object detection and recognition
- The `find_object_2d` package in ROS
- Installing `find_object_2d`
- Detecting and tracking an object using a webcam
- Detecting and tracking using 3D depth sensors
- Getting started with 3D object recognition
- Introducing the object-recognition package in ROS
- Installing object-recognition packages
- Detecting and recognizing objects using 3D meshes
- Training and detecting using real-time capture
- Final run

So let's begin with the importance of object detection and recognition in robotics.

Getting started with object detection and recognition

So what's the main difference between detection and recognition? Consider face detection and face recognition. In face detection, the algorithm tries to detect a face from an image, but in recognition, the algorithm can also state information about whose face is detected. It may be the person's name, gender, or something else.

Similarly, object detection involves the detection of a class of object and recognition performs the next level of classification, which tells us the name of the object.

There is a vast number of applications that use object detection and recognition techniques. Here is a popular application that is going to be used in Amazon warehouses:



Figure 1: A photo from an Amazon Picking Challenge

Amazon is planning to automate the picking and placing of objects from the shelves inside their warehouses. To retrieve objects from the shelves, they are planning to deploy robotic arms such as the one shown in the previous image. Whenever the robot gets an order to retrieve a specific object and place it in a basket, it should identify the position of object first, right? So how does the robot understand the object position? It should need some kind of 3D sensor, right? And also, on the software side, it should have some object recognition algorithm for recognizing each object. The robot will get the object coordinates only after the recognition. The detected coordinates will be relative to the vision sensors, which have to transform into robot end-effector coordinates of the tip of the robot to reach the object position. After reaching the object position, what should be the robot do? It should grasp the object and place it in the basket, right? The task looks simple, doesn't it? But it's not as simple as we think. Here is the coordinate system of a robotic arm, end effector, Kinect, and the object:

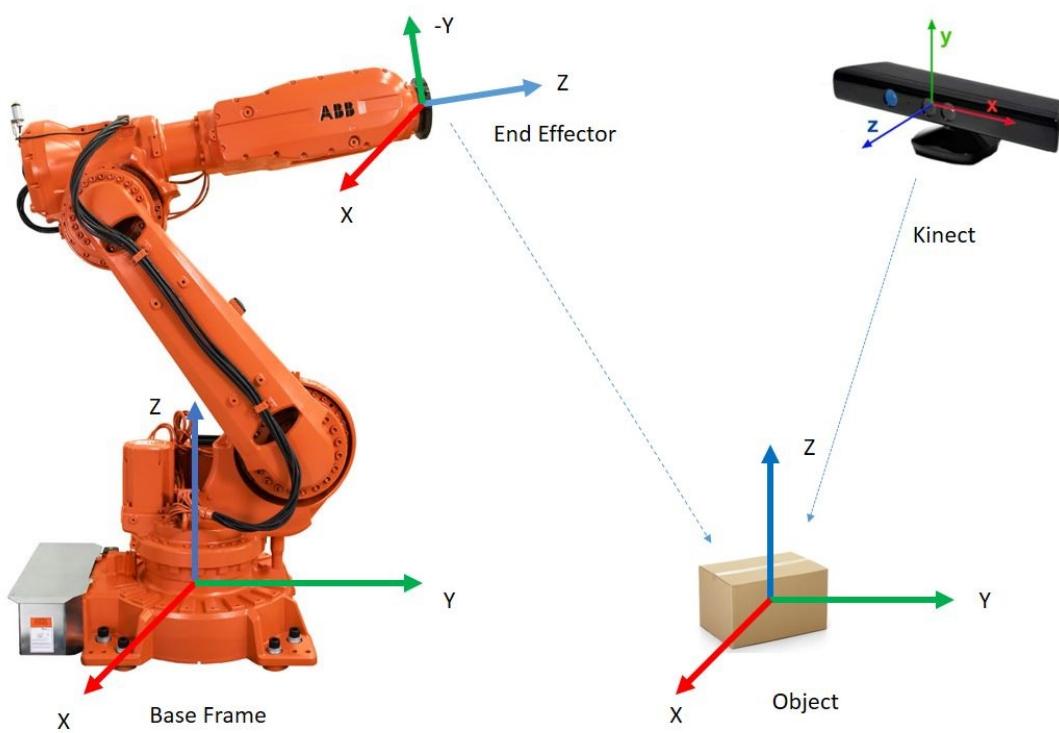


Figure 2: Coordinate system of each component

Amazon organizes a challenge called the Amazon Picking Challenge, which was first conducted as a part of ICRA 2015 (<http://www.ieee-ras.org/conference/robot-challenges>), and in 2016, it was conducted along with Robocup (<http://www.robocup2016.org/en/events/amazon-picking-challenge/>). The challenge was all about solving the pick-and-place problem we just discussed. In effect, the object recognition and detection tasks have immense scope in the industry, not only in Amazon but also in areas such as agriculture, defense, and space.

In the following section, we will see how to implement object detection and recognition in our applications. There are some good ROS packages to do this stuff. Let's discuss each package one by one.

The `find_object_2d` package in ROS

One of the advantages of ROS is that it has tons of packages that can be reused in our applications. In our case, what we want is to implement an object recognition and detection system. The `find_object_2d` package (http://wiki.ros.org/find_object_2d) implements SURF, SIFT, FAST, and BRIEF feature detectors (<https://goo.gl/B8H9Zm>) and descriptors for object detection. Using the GUI provided by this package, we can mark the objects we want to detect and save them for future detection. The detector node will detect the objects in camera images and publish the details of the object through a topic. Using a 3D sensor, it can estimate the depth and orientation of the object.

Installing find_object_2d

Installing this package is pretty easy. Here is the command to install it on Ubuntu 16.04 and ROS Kinetic:

```
| $ sudo apt-get install ros-kinetic-find-object-2d
```

Installing from source code

Switch into the ROS workspace:

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

Clone the source code into the `src` folder:

```
| $ git clone https://github.com/introlab/find-object.git src/find_object_2d
```

Build the workspace:

```
| $ catkin_make
```

Running find_object_2d nodes using webcams

Here is the procedure to run the detector nodes for a webcam. If we want to detect an object using a webcam, we first need to install the `usb_cam` package, which was discussed in *Chapter 20, Face Detection and Tracking Using ROS, OpenCV, and Dynamixel Servos*.

1. Start `roscore`:

```
| $ roscore
```

2. Plug your USB camera into your PC, and launch the ROS `usb_cam` driver:

```
| $ roslaunch usb_cam usb_cam-test.launch
```

This will launch the ROS driver for USB web cameras, and you can list the topics in this driver using the `rostopic list` command. The list of topics in the driver is shown here:

```
robot@robot-pc:~$ rostopic list
/image_view/parameter_descriptions
/image_view/parameter_updates
/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
```

Figure 3: Topics being published from the camera driver

3. From the topic list, we are going to use the raw image topic from the cam, which is being published to the `/usb_cam/image_raw` topic. If you are getting this topic, then the next step is to run the object detector node. The following command will start the object detector node:

```
| $ rosrun find_object_2d find_object_2d image:=/usb_cam/image_raw
```

This command will open the object detector window, shown in the previous screenshot, in which we can see the camera feed and the feature points on the objects.

4. So how can we use it for detecting an object? Here are the procedures to perform a basic detection using this tool:

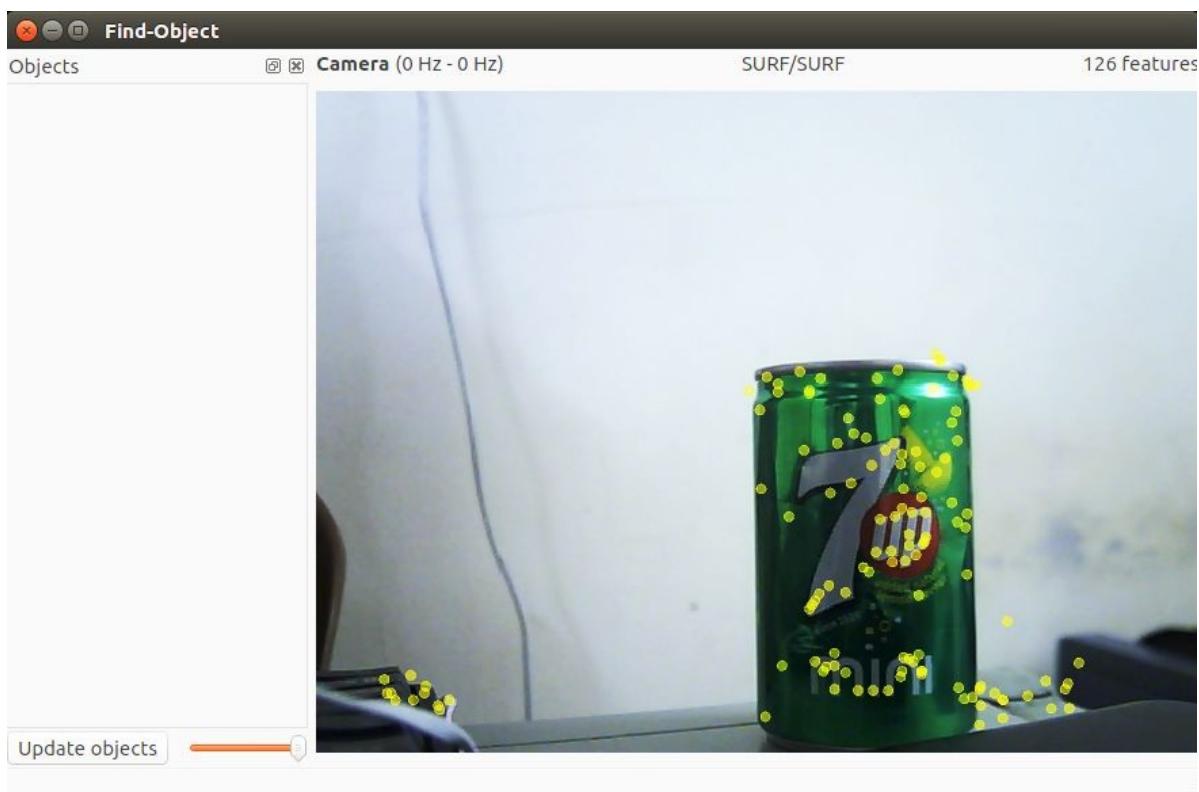


Figure 4: The Find-Object detector window

5. You can right-click on the left-hand side panel (Objects) of this window, and you will get an option to Add objects from scene. If you choose this option, you will be directed to mark the object from the current scene, and after completing the marking, the marked object will start to track from the scene. The previous screenshot shows the first step, which is taking a snap of the scene having the object.
6. After aligning the object toward the camera, press the Take Picture button to take a snap of the object:

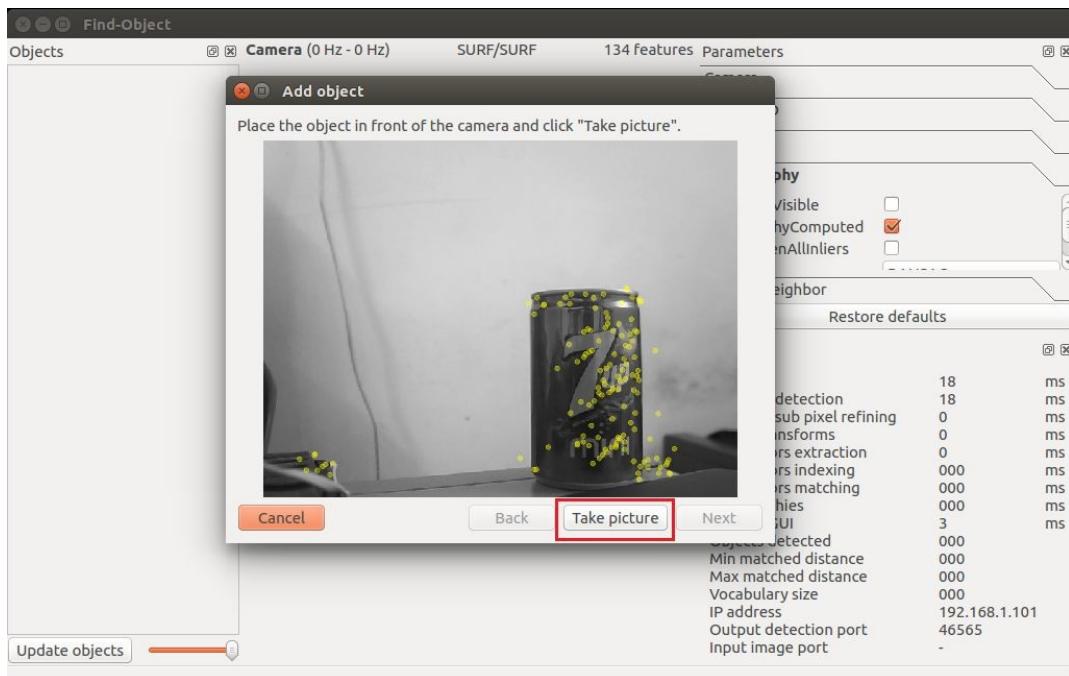


Figure 5: The Add object wizard for taking a snap of the object

7. The next window is for marking the object from the current snap. The following figure shows this. We can use the mouse pointer to mark the object. Click on the Next button to crop the object, and you

can proceed to the next step:

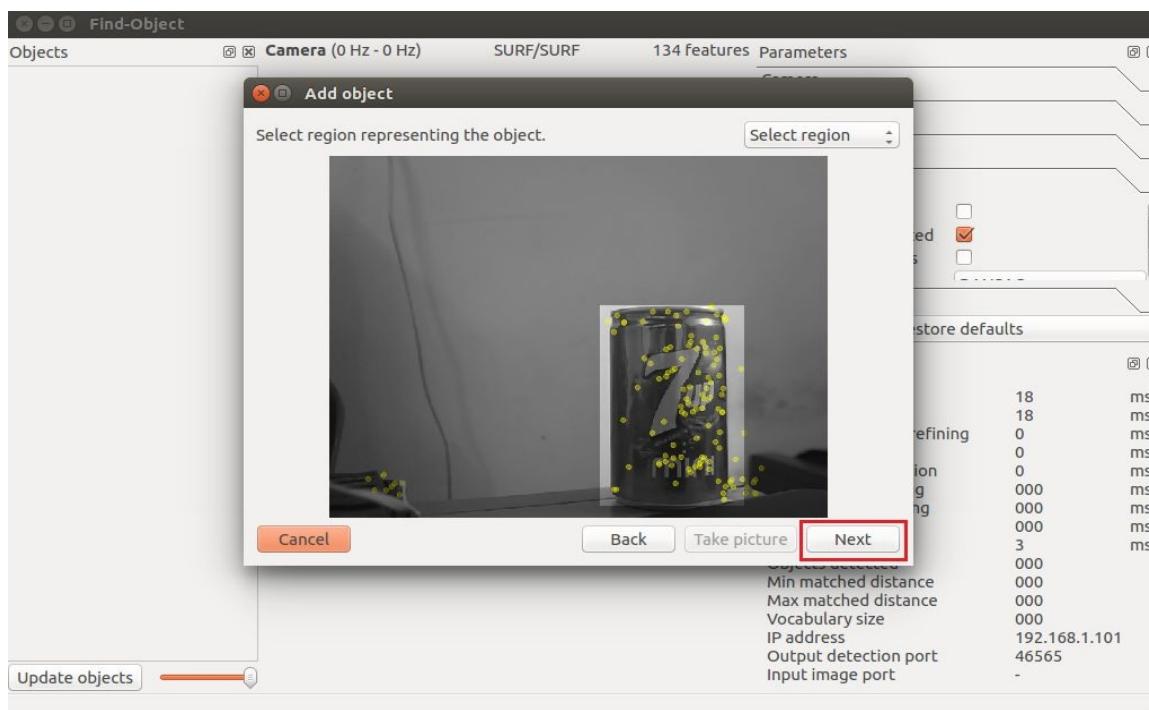


Figure 6: The Add object wizard for marking the object

8. After cropping the object, it will show you the total number of feature descriptors on the object, and you can press the End button to add the object template for detection. The following figure shows the last stage of adding an object template to this detector application:

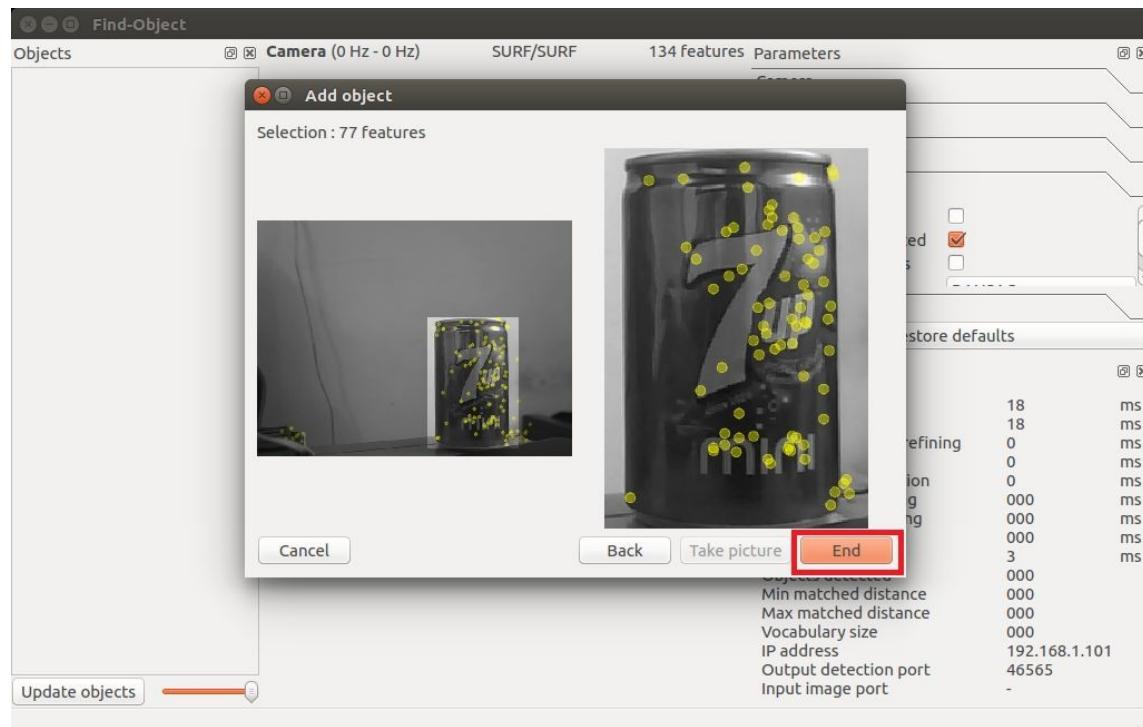


Figure 7: The last step of the Add object wizard

9. Congratulations! You have added an object for detection. Immediately after adding the object, you will be able to see the detection shown in the following figure. You can see a bounding box around the detected object:

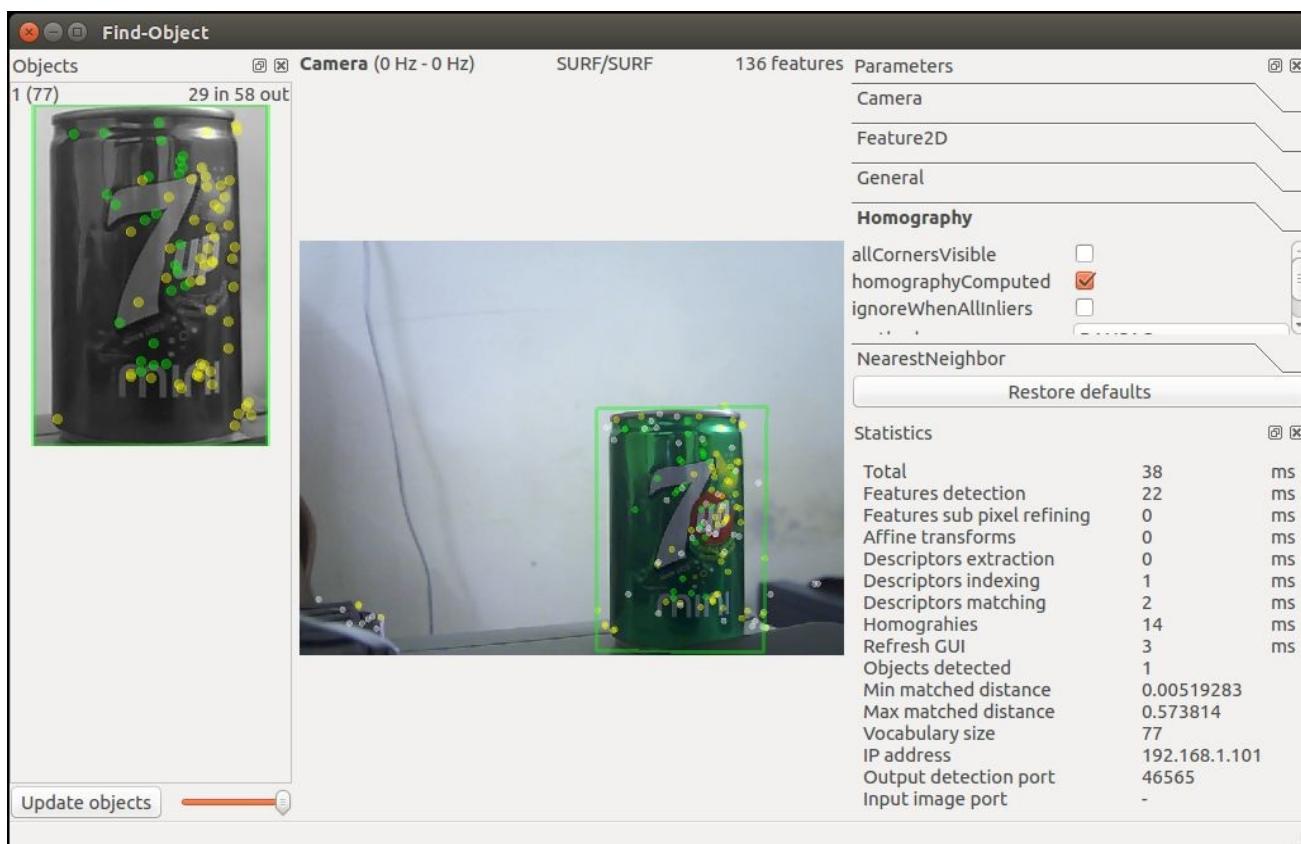


Figure 8: The Find-Object wizard starting the detection

- Is that enough? What about the position of the object? We can retrieve the position of the object using the following command:

```
$ rosrun find_object_2d print_objects_detected
```

```
Object 1 detected, Qt corners at (349.005890,193.805511) (548.637444,194.173731) (347.745106,476.882712) (546.294145,476.546963)
...
Object 1 detected, Qt corners at (348.654999,193.916397) (548.521667,194.306640) (347.606602,477.062803) (546.462457,476.123968)
...
Object 1 detected, Qt corners at (349.047729,194.432526) (549.321203,192.786282) (348.294037,476.387848) (546.751276,476.828125)
...
Object 1 detected, Qt corners at (348.831024,193.950211) (548.481769,194.160784) (347.401401,477.281268) (546.429143,476.099446)
...
Object 1 detected, Qt corners at (348.817383,194.296692) (549.181819,193.494925) (347.031157,477.201164) (546.595589,476.701254)
...
Object 1 detected, Qt corners at (348.775452,193.905640) (548.325150,194.470809) (347.195352,477.604420) (546.620953,476.316763)
...
Object 1 detected, Qt corners at (349.173157,194.101913) (548.643213,193.599104) (348.129605,476.705982) (546.500466,476.719158)
...
Object 1 detected, Qt corners at (349.087555,194.182556) (549.201040,193.357581) (348.503663,475.883037) (546.713381,477.380333)
```

Figure 9: The object details

- You can also get the complete information about the detected object from the `/object` topic. The topic publishes a multi-array that consist of the width and height of the object and the homography matrix to compute the position and orientation of the object and its scale and shear values. You can echo the `/objects` topic to get output like this:

```

objects:
  layout:
    dim: []
    data_offset: 0
    data: [3.0, 202.0, 393.0, 1.0872979164123535, 0.048006415367126465,
0.00016619441157672554, 0.008327833376824856, 1.057657241821289, 1.759
977385518141e-05, 280.7042236328125, 74.2943115234375, 1.0]
---
header:
  seq: 43
  stamp:
    secs: 1476380300
    nsecs: 630851717
  frame_id: usb_cam
objects:
  layout:
    dim: []
    data_offset: 0
    data: [3.0, 202.0, 393.0, 1.0946733951568604, 0.05213480815291405, 0
.00018432267825119197, 0.00999543722718954, 1.0606046915054321, 2.1411
471607279964e-05, 280.6572265625, 74.00088500976562, 1.0]

```

Figure 10: The /object topic values

12. We can compute the new position and orientation from the following equations:

$$H = \begin{bmatrix} h_{00} & h_{01} & h_{02} \\ h_{10} & h_{11} & h_{12} \\ h_{20} & h_{21} & h_{22} \end{bmatrix} \quad \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} = H \begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix} = \begin{bmatrix} h_{00} & h_{01} & h_{02} \\ h_{10} & h_{11} & h_{12} \\ h_{20} & h_{21} & h_{22} \end{bmatrix} \begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix}$$

Figure 11: The equation to compute object position

Here, H is the homography 3x3 matrix, (x_1, y_1) is the object's position in the stored image, and (x_2, y_2) is the computed object position in the current frame.

 You can check out the source code of the `print_object_detected_src` node to get the conversion using a homography matrix. Here is the source code of this node: https://github.com/introlab/find-object/blob/master/src/ros/print_objects_detected_node.cpp.

Running find_object_2d nodes using depth sensors

Using a webcam, we can only find the 2D position and orientation of an object, but what should we use if we need the 3D coordinates of the object? We could simply use a depth sensor like the Kinect and run these same nodes. For interfacing the Kinect with ROS, we need to install some driver packages. The Kinect can deliver both RGB and depth data. Using RGB data, the object detector detects the object, and using the depth value, it computes the distance from the sensor too.

Here are the dependent packages for working with the Kinect sensor:

- If you are using the Xbox Kinect 360, which is the first Kinect, you have to install the following package to get it working:
| \$ sudo apt-get install ros-kinetic-openni-launch
- If you have Kinect version 2, you may need a different driver package, which is available on GitHub. You may need to install it from the source code. The following is the ROS package link of the V2 driver. The installation instructions are also given:https://github.com/code-iai/iai_kinect2

If you are using the Asus Xtion Pro or other PrimeSense device, you may need to install the following driver to work with this detector:

```
|   $ sudo apt-get install ros-kinetic-openni2-launch
```

In this section, we will be working with the Xbox Kinect, which is the first version of Kinect.

Before starting the Kinect driver, you have to plug the USB to your PC and make sure that the Kinect is powered using its adapter. Once everything is done, you can launch the drivers using the following command:

```
|   $ roslaunch openni_launch openni.launch depth_registration:=true
```

1. If the driver is running without errors, you should get the following list of topics:

```

lentin@lentin-Aspire-4755:~$ rostopic list
/camera/depth/camera_info
/camera/depth/disparity
/camera/depth/image
/camera/depth/image/compressed
/camera/depth/image/compressed/parameter_descriptions
/camera/depth/image/compressed/parameter_updates
/camera/depth/image/compressedDepth
/camera/depth/image/compressedDepth/parameter_descriptions
/camera/depth/image/compressedDepth/parameter_updates
/camera/depth/image/theora
/camera/depth/image/theora/parameter_descriptions
/camera/depth/image/theora/parameter_updates
/camera/depth/image_raw
/camera/depth/image_raw/compressed
/camera/depth/image_raw/compressed/parameter_descriptions
/camera/depth/image_raw/compressed/parameter_updates
/camera/depth/image_raw/compressedDepth

```

Figure 12: List of topics from the Kinect openNI driver

2. If you are getting this, start the object detector and mark the object as you did for the 2D object detection. The procedure is the same, but in this case, you will get the 3D coordinates of the object. The following diagram shows the detection of the object and its TF data on Rviz. You can see the side view of the Kinect and the object position in Rviz.

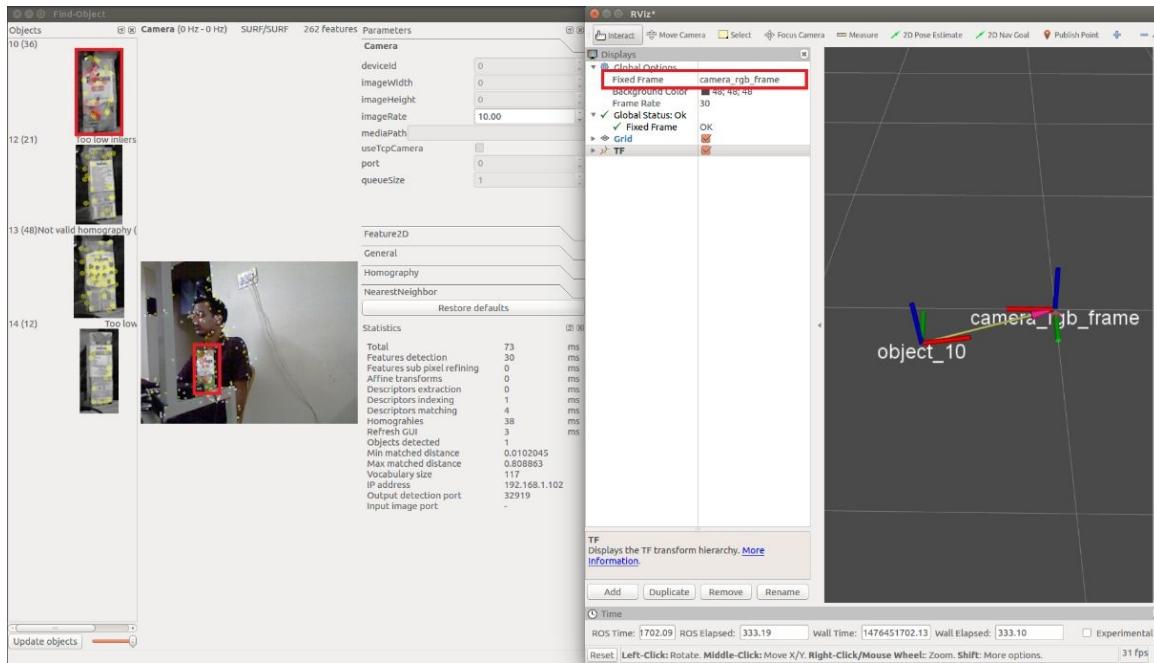


Figure 13: Object detection using Kinect

3. To start the object detection, you have to perform some tweaks in the existing launch file given by this package. The name of the launch file for object detection is `find_object_3d.launch`.



You can directly view this file from the following link: https://github.com/introlab/find-object/blob/master/launch/find_object_3d.launch. This launch file is written for an autonomous robot that detects objects while navigating the surrounding.

4. We can modify this file a little bit because in our case, there is no robot, so we can modify it in such a way that the TF information should be published with respect to Kinect's `camera_rgb_frame`, which is shown in the previous diagram. Here is the launch file definition we want for the demo:

```

<launch>
  <node name="find_object_3d" pkg="find_object_2d"
    type="find_object_2d" output="screen">
    <param name="gui" value="true" type="bool"/>
    <param name="settings_path"
      value="~/.ros/find_object_2d.ini" type="str"/>
    <param name="subscribe_depth" value="true"
      type="bool"/>
    <param name="objects_path" value="" type="str"/>
    <param name="object_prefix" value="object"
      type="str"/>
    <remap from="rgb/image_rect_color"
      to="camera/rgb/image_rect_color"/>
    <remap from="depth_registered/image_raw"
      to="camera/depth_registered/image_raw"/>
    <remap from="depth_registered/camera_info"
      to="camera/depth_registered/camera_info"/>
  </node>
</launch>

```

In this code, we just removed the static transform required for the mobile robot. You can also change the `object_prefix` parameter to name the detected object.

Using the following commands, you can modify this launch file, which is already installed on your system:

```
$ roscd find_object_2d/launch
$ sudo gedit find_object_3d.launch
```

Now, you can remove the unwanted lines of code and save your changes. After saving this launch file, launch it to start detection:

```
$ rosrun find_object_2d find_object_3d.launch
```

You can mark the object and it will start detecting the marked object.

5. To visualize the TF data, you can launch Rviz, make the fixed frame `/camera_link` OR `/camera_rgb_frame`, and add a TF display from the left panel of Rviz.
6. You can run Rviz using the following command:

```
$ rosrun rviz rviz
```

Other than publishing TF, we can also see the 3D position of the object in the detector Terminal. The detected position values are shown in the following screenshot:

```
[ INFO] [1476451737.207323202]: Object_10 [x,y,z] [x,y,z,w] in "/camera_rgb_frame" frame: [0.542000,0.131629,-0.077945] [0.077937,-0.007629,0.996919,-0.004438]
[ INFO] [1476451737.207384747]: Object_10 [x,y,z] [x,y,z,w] in "camera_rgb_optical_frame" frame: [-0.131629,0.077945,0.542000] [0.457895,0.461087,-0.531395,0.543462]
[ INFO] [1476451737.305781577]: Object_10 [x,y,z] [x,y,z,w] in "/camera_rgb_frame" frame: [0.542000,0.130596,-0.077945] [0.059449,0.011080,0.998159,-0.004555]
[ INFO] [1476451737.305822771]: Object_10 [x,y,z] [x,y,z,w] in "camera_rgb_optical_frame" frame: [-0.130596,0.077945,0.542000] [0.477172,0.461538,-0.532067,0.525542]
[ INFO] [1476451737.543368607]: Object_10 [x,y,z] [x,y,z,w] in "/camera_rgb_frame" frame: [0.542000,0.131629,-0.077945] [0.080979,-0.000803,0.996625,-0.013447]
[ INFO] [1476451737.543409748]: Object_10 [x,y,z] [x,y,z,w] in "camera_rgb_optical_frame" frame: [-0.131629,0.077945,0.542000] [0.464145,0.451501,-0.531677,0.545927]
```

Figure 14: Printing the 3D object's position

Getting started with 3D object recognition

In the previous section, we dealt with 2D object recognition using a 2D and 3D sensor. In this section, we will discuss 3D recognition. So what is 3D object recognition? In 3D object recognition, we take the 3D data or point cloud data of the surroundings and 3D model of the object. Then, we match the scene object with the trained model, and if there is a match found, the algorithm will mark the area of detection.

In real-world scenarios, 3D object recognition/detection is much better than 2D because in 3D detection, we use the complete information of the object, similar to human perception. But there are many challenges involved in this process too. Some of the main constraints are computational power and expensive sensors. We may need more expensive computers to process 3D information; also, the sensors for this purpose are costlier.

Some of the latest applications using 3D object detection and recognition are autonomous robots, especially self-driving cars. Self-driving cars have a LIDAR such as Velodyne (<http://velodynelidar.com/>) that can provide a complete 3D point cloud around the vehicle. The computer inside takes the 3D input and run various detectors to find pedestrians, cyclists, and other obstacles for a collision-free ride.

Like we discussed in the beginning, in the Amazon Picking Challenge and other such applications, the picking and placing needs 3D recognition capability. The following figure shows how an autonomous car perceives the world. The data shown around the car is the 3D point cloud, which helps it detect objects and predict a collision-free route.

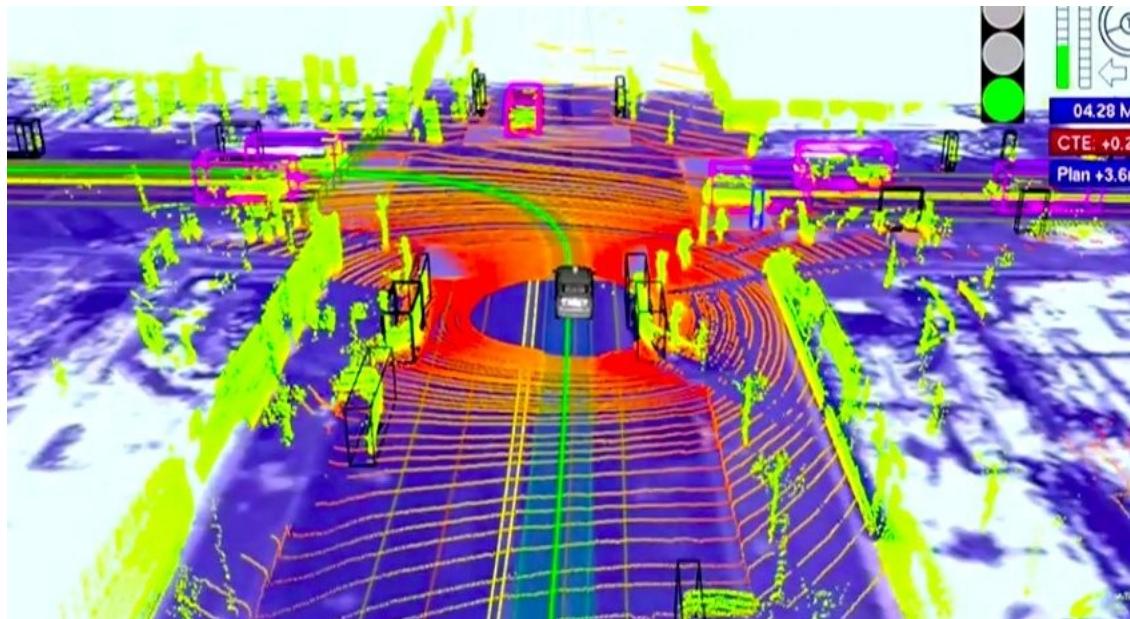


Figure 15: Typical 3D data from an autonomous car

3D object recognition has many applications, and in this section, you are going to see how to perform a basic 3D object recognition using ROS and cheap depth sensors.

Introduction to 3D object recognition packages in ROS

ROS has packages for performing 3D object recognition. One of the popular packages we are dealing with in this section is the **Object Recognition Kitchen (ORK)**. This project was started at Willow Garage mainly for 3D object recognition. The ORK is a generic way to detect any kind of object, whether it be textured, nontextured, transparent, and so on. It is a complete kit in which we can run several object-recognition techniques simultaneously. It is not just a kit for object recognition, but it also provides non-vision aspects, such as database management to store 3D models, input/output handling, robot-ROS integration, and code reuse.



ORK home page: http://wg-perception.github.io/object_recognition_core/. *ORK ROS page:* http://wiki.ros.org/object_recognition

Installing ORK packages in ROS

Here are the installation instructions to set up the `object_recognition` package in ROS. We can install it using prebuilt binaries and source code. The easiest way to install is via binaries.

Here is the command to install ORK packages in ROS:

```
| $ sudo apt-get install ros-kinetic-object-recognition-*
```

If you want to install these packages in ROS Indigo, replace `kinetic` with `indigo`.

This command will install the following ROS packages:

- `object-recognition-core`: This package contains tools to launch several recognition pipelines, train objects, and store models.
- `object-recognition-linemod`: This is an object recognition pipeline that uses linemod from OpenCV. The linemod pipeline is best for rigid body detection.
- `object-recognition-tabletop`: This is a pipeline use for pick-and-place operations from a flat surface
- `object-recognition-tod`: Textured Object Recognition is another pipeline for textured objects that uses features for detection.
- `object-recognition-reconstruction`: This is a basic 3D reconstruction of an object from aligned Kinect data.
- `object-recognition-renderer`: This is code that generates random views of an object.
- `object-recognition-msgs`: This package contains the ROS message and the `actionlib` definition used in `object_recognition_core`.
- `object-recognition-capture`: Capture is a set of tools to capture objects in 3D and perform odometry.
- `object-recognition-transparent-objects`: This is a technique to recognize and estimate poses of transparent objects.
- `object-recognition-ros-visualization`: This package contains Rviz plugins to visualize ORK detection results.

Here are the commands to install the packages from source. This command is basically based on the `rosinstall` tool, which helps set up a list of packages in a single command. You can run this commands from the `/home/<user>` folder.

```
$ mkdir ws && cd ws
$ wstool init src https://raw.github.com/wg-
perception/object_recognition_core/master/doc/source/ork.rosinstall.kinetic.plus
$ cd src && wstool update -j8
$ cd .. && rosdep install --from-paths src -i -y
$ catkin_make
$ source devel/setup.bash
```

You can find more about LINE-MODE from the following link: <http://far.in.tum.de/Main/StefanHinterstoesser> This is the GitHub repository of the object-recognition packages: <https://github.com/wg-perception> Here are the possible issues you may have while working with this package: https://github.com/wg-perception/object_recognition_ros/issues <https://github.com/wg-perception/linemod/issues> https://github.com/wg-perception/object_recognition_core/issues



Detecting and recognizing objects from 3D meshes

After installing these packages, let's start the detection. What are the procedures involved? Here are the main steps:

1. Building a CAD model of the object or capturing its 3D model
2. Training the model
3. Detecting the object using the trained model

The first step in the recognition process is building the 3D model of the desired object. We can do it using a CAD tool, or we can capture the real object using depth-sensing cameras. If the object is rigid, then the best procedure is CAD modelling, because it will have all the 3D information regarding the object. When we try to capture and build a 3D model, it may have errors and the mesh may not look like the actual object because of the accumulation of errors in each stage. After building the object model, it will be uploaded to the object database. The next phase is the training of the uploaded object on the database. After training, we can start the detection process. The detection process will start capturing from the depth sensors and will match with the trained model in the database using different methods, such as **Random Sample Consensus (RANSAC)**. If there is a match, it will mark the area and print the result. We can see the final detection output in Rviz.

Let's see how to add a mesh of an object to the object database. There are ORK tutorial packages that provide meshes of some objects, such as soda bottles. We can use one of these objects and add it to the object database.

Training using 3D models of an object

We can clone the ORK tutorial package using the following command:

```
| $ git clone https://github.com/wg-perception/ork_tutorials
```

You can see that the `ork_tutorials/data` folder contains some mesh files that we can use for object detection. Navigate to that folder and execute the following commands from the same path. The following command will add an entry to the object database:

```
| $ rosrun object_recognition_core object_add.py -n "coke" -d "A universal coke" --commit
```

The object name is mentioned after that `-n` argument and the object description after `-d`. The `--commit` argument is to commit these operations. When the operation is successful, you will get the ID of the object. This ID is used in the next command. The next command is to upload the mesh file of the object to the created entry:

```
| $ rosrun object_recognition_core mesh_add.py <ID_OF_OBJECT> coke.stl --commit
```

Here's an example:

```
| $ rosrun object_recognition_core mesh_add.py cfab1c4804c316ea23c698ecbf0026e4 coke.stl --commit
```

We are mentioning the name of object model--`coke.stl`--in this command, which is in the data folder. We are not mentioning the path here because we are already in that path. If not, we have to mention the absolute path of the model.

If it is successful, you will get output saying the model has been stored in the database.

Do you want to see the uploaded model? Here is the procedure:

1. Install `couchapp`. The object recognition package uses `couchdb` as the database. So we need the following application to view the model from the database:

```
| $ sudo pip install git+https://github.com/couchapp/couchapp.git
```

2. After setting up the application, you can run the following command:

```
| $rosrun object_recognition_core push.sh
```

3. If everything is successful, you will get a message like this:

[INFO] Visit your CouchApp here:

http://localhost:5984/or_web_ui/_design/viewer/index.html

4. Click on the link, and you will get the list of objects and their visualizations in your web browser. Here is a set of screenshots of this web interface:

Figure 16: Web interface for viewing object models

All right! The object model has been properly uploaded to the database.

5. After uploading the model, we have to train it. You can use the following command:

```
$ rosrun object_recognition_core training -c `rospack find object_recognition_linemod`/conf/training.ork
```

6. If the training is successful, you will see a message like this:

```
lentin@lentin-Aspire-4755:~/Desktop/Files/3d_obj_recog/ork_tutorials/data$ rosrun object_recognition_core training -c `rospack find object_recognition_linemod`/conf/training.ork
Training 1 objects.
computing object_id: cfab1c4804c316ea23c698ecbf0026e4
Info, T0: Load /tmp/fileXdkkE5.stl
Info, T0: Found a matching importer for this file format
Info, T0: Import root directory is '/tmp/'
Info, T0: Entering post processing pipeline
Info, T0: Points: 0, Lines: 0, Triangles: 1, Polygons: 0 (Meshes, X = removed)
Error, T0: FindInvalidDataProcess fails on mesh normals: Found zero-length vectors
Info, T0: FindInvalidDataProcess finished. Found issues ...
Info, T0: GenVertexNormalsProcess finished. Vertex normals have been calculated
Error, T0: Failed to compute tangents; need UV data in channel0
Info, T0: JoinVerticesProcess finished | Verts in: 1536 out: 258 | ~83.2%
Info, T0: Cache relevant are 1 meshes (512 faces). Average output ACMR is 0.669922
Info, T0: Leaving post processing pipeline
Loading images 495/5737
```

Figure 17: Training 3D objects

Training from captured 3D models

If you don't have a 3D mesh of the object, you can also create one by capturing the 3D point cloud data and reconstructing the mesh. Here are the steps to capture and build the mesh of an object:

1. Before the capture, we have to print a pattern for better capturing and reconstruction. You can download the pattern from http://wg-perception.github.io/capture/_downloads/capture_board_big_5x3.svg.pdf.

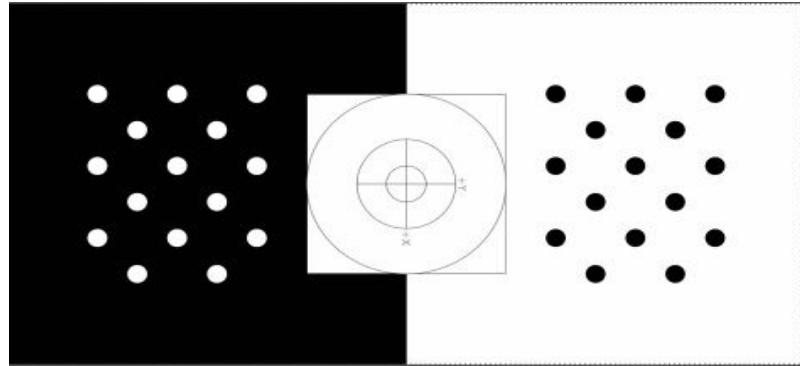


Figure 18: Capture pattern for 3D objects

2. Print the pattern, stick it to some hard board, and place it on a rotating mechanism that you can manually rotate the board along an axis. The object has to be placed at the center of the pattern. The size of the pattern doesn't matter, but the larger the size, the better the detection. We can place the object at the center and place the Kinect where it can detect all the markers. The capture program only capture the object once it detect the markers. Here is an example setup made for the object capture:

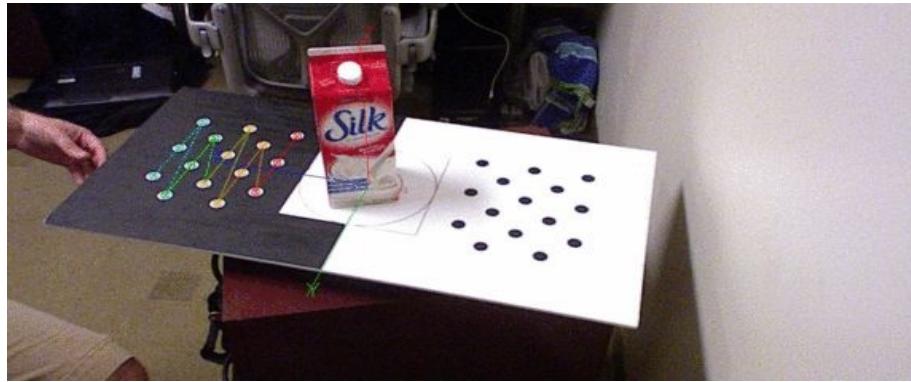


Figure 19: Capturing 3D model of object using patterns

3. If the setup is ready, we can start running the tools to capture the object mode. First, start `roscore`:

```
| $ roscore
```

4. Launch the Kinect driver node, ensuring that Kinect is properly powered on and plugged in to the PC:

```
| $ roslaunch openni_launch openni.launch
```

- Set these parameters for the Kinect ROS driver:

```
$ rosrun dynamic_reconfigure dynparam set /camera/driver depth_registration True  
$ rosrun dynamic_reconfigure dynparam set /camera/driver image_mode 2  
$ rosrun dynamic_reconfigure dynparam set /camera/driver depth_mode 2
```

- The `topic_toolsrelay` parameter basically subscribes to the first topic and republished it in another name. You can run the following two commands on two Terminals:

```
$ rosrun topic_tools relay /camera/depth_registered/image_raw /camera/depth/image_raw  
$ rosrun topic_tools relay /camera/rgb/image_rect_color /camera/rgb/image_raw
```

This command will start the visualization and start capturing the object.

- While running this command, you have to rotate the pattern board to acquire maximum features from the object. Here, `object.bag` is the bag file used to store the captured data.

```
$ rosrun object_recognition_capture capture --seg_z_min 0.01 -o object.bag
```

Here is the screenshot of the capture operation:

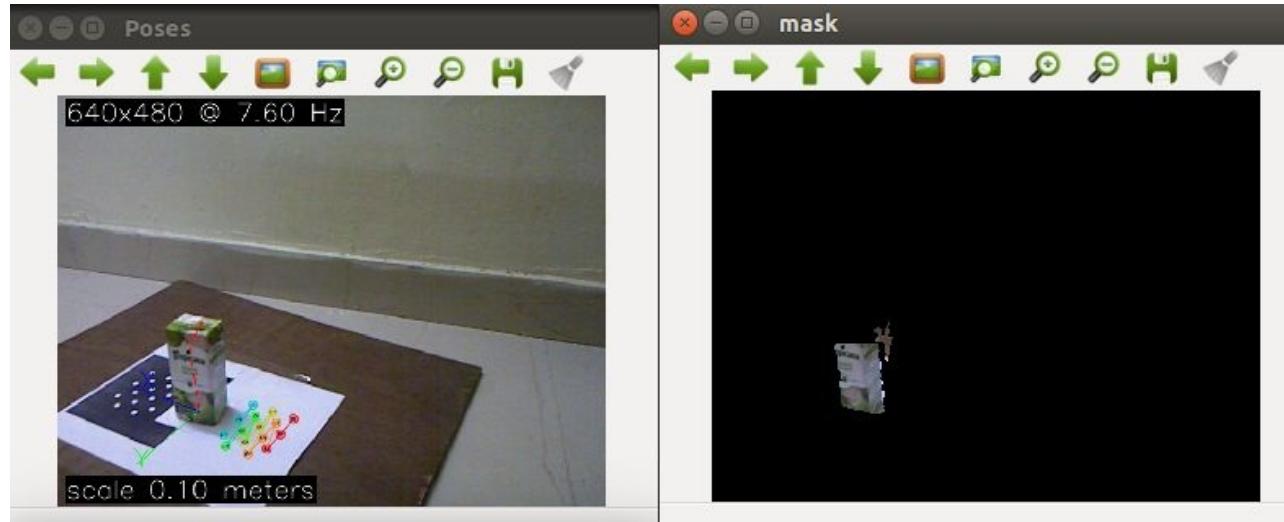


Figure 20: Capturing 3D model

- If the detector gets enough 3D data of an object, it will print that it is satisfied with the data and quit.
- After the capture, we need to upload the data to the database. We have to mention the bag file name, name of the object, and its description. Here is an example command to do that:

```
$ rosrun object_recognition_capture upload -i object.bag -n 'Tropicana' It is a Tropicana --commit
```

- The next phase is the reconstruction of the captured data into a mesh. Here is the command to do that:

```
$ rosrun object_recognition_reconstruction mesh_object --all --visualize --commit
```

- You will see the conversion as shown here:

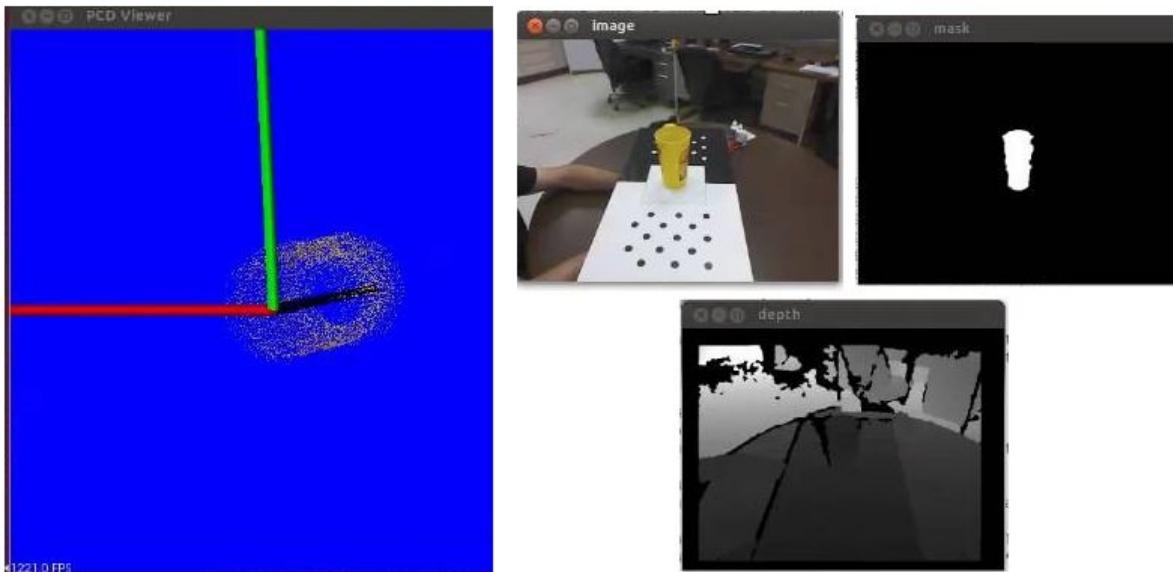


Figure 21: Reconstruction of mesh (with a different object)

12. You can see the point cloud of the captured object and the image during reconstruction. After reconstruction, we can train the models in the database using the following command:

```
$ rosrun object_recognition_core training -c `rospack find object_recognition_linemod`/conf/training.ork
```

13. You can use different pipelines here for training, such as tod, tabletop, or linemod. Here, we've used the linemod pipeline. Each pipeline has its own merits and demerits.
14. After training, we can check whether the object has been uploaded to the database by going to the following link and checking whether it looks like the screenshot shown after it:

http://localhost:5984/_utils/database.html?object_recognition/

| Key | Value |
|--------|--|
| "coke" | <pre>{ "_id": "2ea6c847f0606e07ff0b70b8e0000422", "_rev": "1-5af03414fe1575e5b6abb62e62d272dc", "added": "2016-10-16T15:34:35Z", "description": "A universal coke", "tags": [], "author_email": "", "author_name": "", "object_name": "coke", "Type": "Object" }</pre> |

Figure 22: List of object in database

The next process is recognizing the object using the trained model. Let's discuss how to do that.

Recognizing objects

There are several commands to start recognition using a trained model.

Starting roscore:

```
| $ roscore
```

Starting the ROS driver for Kinect:

```
| $ roslaunch openni_launch openni.launch
```

Setting the ROS parameters for the Kinect driver:

```
| $ rosrun dynamic_reconfigure dynparam set /camera/driver depth_registration True  
| $ rosrun dynamic_reconfigure dynparam set /camera/driver image_mode 2  
| $ rosrun dynamic_reconfigure dynparam set /camera/driver depth_mode 2
```

Republishing the depth and RGB image topics using topic_tools relay:

```
| $ rosrun topic_tools relay /camera/depth_registered/image_raw /camera/depth/image_raw  
| $ rosrun topic_tools relay /camera/rgb/image_rect_color /camera/rgb/image_raw
```

Here is the command to start recognition; we can use different pipelines to perform detection. The following command uses the tod pipeline. This will work well for textured objects.

```
| $ rosrun object_recognition_core detection -c `rospack find object_recognition_tod`/conf/detection.ros.ork --  
| visualize
```

Alternatively, we can use the tabletop pipeline, which can detect objects placed on a flat surface, such as a table itself:

```
| $ rosrun object_recognition_core detection -c `rospack find  
| object_recognition_tabletop`/conf/detection.object.ros.ork
```

You could also use the linemod pipeline, which is the best for rigid object recognition:

```
| $ rosrun object_recognition_core detection -c `rospack find  
| object_recognition_linemod`/conf/detection.object.ros.ork
```

After running the detectors, we can visualize the detections in Rviz. Let's start Rviz and load the proper display type, shown in the screenshot:

```
| $ rosrun rviz rviz
```



Figure 23: Object detection visualized in Rviz

The Fixed Frame can be set to `camera_rgb_frame`. Then, we have to add a `PointCloud2` display with the `/camera/depth_registered/points` topic. To detect the object and display its name, you have to add a new display type called `orkObject`, which is installed along with the object-recognition package. You can see the object being detected, as shown in the previous screenshot.

If it is a tabletop pipeline, it will mark the plane area in which object is placed, as shown in the next screenshot. This pipeline is good for grasping objects from a table, which can work well with the ROS MoveIt! package.

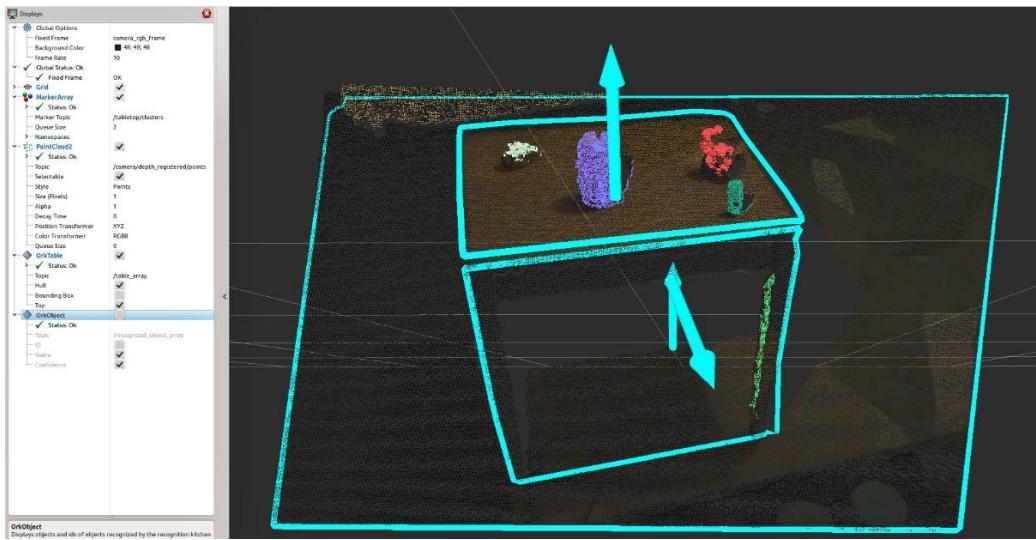


Figure 24: Tabletop detection visualized in Rviz

For visualizing, you need to add `orkTable` with the `/table_array` topic and `MarkerArray` with the `/tabletop/clusters` topic.

We can add any number of objects to the database; detection accuracy depends on the quality of model, quality of 3D input, and processing power of the PC.

Questions

- What is the main difference between object detection and recognition?
- What is 2D and 3D object recognition?
- What are the main functions of the `find_object_2d` package in ROS?
- What are the main steps involved in detecting 3D objects using an object recognition package in ROS?

Summary

In this chapter, we dealt with object detection and recognition. Both these things are extensively used in robotic applications. The chapter started with a popular ROS package for 2D object detection. The package is called `find_2d_object`, and we covered object detection using a webcam and Kinect. After going through a demo using this package, we discussed 3D object recognition using a ROS package called `object_recognition`, which is mainly for 3D object recognition. We saw methods to build and capture the object model and its training procedure. After training, we discussed the steps to start detecting the object using a depth camera. Finally, we visualized the object recognition in Rviz.

In the next chapter, we will deal with interfacing ROS and Google TensorFlow.

Deep Learning Using ROS and TensorFlow

You may have come across *deep learning* many times on the Web. Most of us are not fully aware of this technology, and many people are trying to learn it too. So in this chapter, we are going to see the importance of deep learning in robotics and how we can implement robotics applications using deep learning and ROS.

Here are the main topics we are going to discuss in this chapter:

- Introducing deep learning and its applications
- Deep learning for robotics
- Software frameworks and programming languages for deep learning
- Getting started with Google TensorFlow
- Installing TensorFlow for Python
- Embedding TensorFlow APIs in ROS
- Image recognition using ROS and TensorFlow
- Introduction to scikit-learn
- Implementing SVM using scikit-learn
- Embedding SVM on a ROS node
- Implementing an SVM-ROS application

Introduction to deep learning and its applications

So what actually is deep learning? It is a buzzword in neural network technology. What is a neural network then? An artificial neural network is a computer software model that replicates the behaviour of neurons in the human brain. A neural network is one way to classify data. For example, if we want to classify an image by whether it contains an object or not, we can use this method. There are several other computer software models for classification like logistic regression, **Support Vector Machine (SVM)**; a neural network is one among them.

So why we are not calling it neural network instead of deep learning? The reason is that in deep learning, we use a large number of artificial neural networks. So you may ask, "So why it was not possible before?" The answer: to create a large number of neural networks (multilayer perceptron), we may need a high amount of computational power. So how has it become possible now? It's because of the availability of cheap computational hardware. Will computational power alone do the job? No, we also need a large dataset to train with.

When we train a large set of neurons, it can learn various features from the input data. After learning the features, it can predict the occurrence of an object or anything we have taught to it.

To teach a neural network, we can either use the supervised learning method or go unsupervised. In supervised learning, we have a training dataset with input and its expected output. These values will be fed to the neural network, and the weights of the neurons will be adjusted in such a way that it can predict which output it should generate whenever it gets a particular input data. So what about unsupervised learning? This type of algorithm learns from an input dataset without having corresponding outputs. The human brain can work like supervised or unsupervised way, but unsupervised learning is more predominant in our case.

The main applications of deep neural networks are in the classification and recognition of objects, such as image recognition and speech recognition.

In this section, we are mainly dealing with supervised learning for building deep learning applications for robots.

Deep learning for robotics

Here are the main robotics areas where we apply deep learning:

- **Deep-learning-based object detector:** Imagine a robot wants to pick a specific object from a group of objects. What could be the first step for solving this problem? It should identify the object first, right? We can use image processing algorithms such as segmentation and Haar training to detect an object, but the problem with those techniques is they are not scalable and can't be used for many objects. Using deep learning algorithms, we can train a large neural network with a large dataset. It can have good accuracy and scalability compared to other methods. Datasets such as ImageNet (<http://image-net.org/>), which have a large collection of image datasets, can be used for training. We also get trained models that we can just use without training. We will look at an ImageNet-based image recognition ROS node in an upcoming section.
- **Speech recognition:** If we want to command a robot to perform some task using our voice, what will we do? Will the robot understand our language? Definitely not. But using deep learning techniques, we can build more accurate speech recognition system compared to the existing **Hidden Markov Model (HMM)** based recognizer. Companies such as Baidu (<http://research.baidu.com/>) and Google (<http://research.google.com/pubs/SpeechProcessing.html>) are trying hard to create a global speech recognition system using deep learning techniques.
- **SLAM and localization:** Deep learning can be used to perform SLAM and localization of mobile robots, which perform much better than conventional methods.
- **Autonomous vehicles:** The deep learning approach in self-driving cars is a new way of controlling the steering of vehicles using a trained network in which sensor data can be fed to the network and corresponding steering control can be obtained. This kind of network can learn by itself while driving.
- **Deep reinforcement learning:** Do you want to make your robot act like a human? Then use this technique. Reinforcement learning is a kind of machine learning technique that allows machines and software agents to automatically determine the ideal behavior within a specific context, in order to maximize its performance. By combining it with deep learning, robots can truly behave as truly intelligent agents that can solve tasks that are considered challenging by humans.

One of the companies doing a lot in deep reinforcement learning is DeepMind owned by Google. They have built a technique to master the Atari 2600 games to a superhuman level with only the raw pixels and score as inputs (<https://deepmind.com/research/dqn/>). AlphaGo is another computer program developed by DeepMind, which can even beat a professional human Go player (<https://deepmind.com/research/alphago/>).

Deep learning libraries

Here are some of the popular deep learning libraries used in research and commercial applications:



Figure 1: Popular deep learning libraries

- **TensorFlow**: This is an open source software library for numerical computation using data flow graphs. The TensorFlow library is designed for machine intelligence and developed by the Google Brain team. The main aim of this library is to perform machine learning and deep neural network research. It can be used in a wide variety of other domains as well (<https://www.tensorflow.org/>).
- **Theano**: This is an open source Python library (<http://deeplearning.net/software/theano/>) that enables us to optimize and evaluate mathematical expressions involving multidimensional arrays efficiently. Theano is primarily developed by the machine learning group at the University of Montreal , Canada.
- **Torch**: Torch is again a scientific computing framework with wide support for machine learning algorithms that puts GPUs first. It's very efficient, being built on the scripting language LuaJIT and has an underlying C/CUDA implementation (<http://torch.ch>).
- **Caffe**: Caffe (<http://caffe.berkeleyvision.org>) is a deep learning library made with a focus on modularity, speed, and expression. It is developed by the **Berkeley Vision and Learning Centre (BVLC)**.

Getting started with TensorFlow

As we discussed, TensorFlow is an open source library mainly designed for fast numerical computing. This library mainly works with Python and was released by Google. TensorFlow can be used as a foundation library to create deep learning models.

We can use TensorFlow both for research and development and in production systems. The good thing about TensorFlow is it can run on a single CPU all the way to a large-scale distributed system of hundreds of machines. It also works well on GPUs and mobile devices.

You can check out the Tensorflow library at the following link:

<https://www.tensorflow.org/>

Installing TensorFlow on Ubuntu 16.04 LTS

Installing TensorFlow is not a tedious task if you have a fast Internet connection. The main tool we need to have is `pip`. It is a package management system used to install and manage software packages written in Python.



You may get latest installation instruction for Linux from following link: https://www.tensorflow.org/install/install_linux

Here is the command to install `pip` on Ubuntu:

```
| $ sudo apt-get install python-pip python-dev
```

After installing `pip`, you have to execute the following command to set a BASH variable called `TF_BINARY_URL`. This is for installing the correct binaries for our configuration. The following variable is for the Ubuntu 64 bit, Python 2.7, CPU only version:

```
| $ export TF_BINARY_URL=https://storage.googleapis.com/tensorflow/linux/cpu/tensorflow-0.11.0-cp27-none-linux_x86_64.whl
```

If you have an NVIDIA GPU, you may need a different binary. You may also need to install CUDA toolkit 8.0 cuDNN v5 for installing this:

```
| $ export TF_BINARY_URL=https://storage.googleapis.com/tensorflow/linux/gpu/tensorflow-0.11.0-cp27-none-linux_x86_64.whl
```

Installing TensorFlow with NVIDIA acceleration: <http://www.nvidia.com/object/gpu-accelerated-applications-tensorflow-installation.html> <https://alliseesolutions.wordpress.com/2016/09/08/install-gpu-tensorflow-from-sources-w-ubuntu-16-04-and-cuda-8-0-rc/> Installing cuDNN: <https://developer.nvidia.com/cudnn> For more Python distributions and other OS configuration, check out the following link: https://www.tensorflow.org/versions/r0.11/get_started/os_setup.html

After defining the BASH variable, use the following command to install the binaries for Python 2:

```
| $ sudo pip install --upgrade $TF_BINARY_URL
```

If everything works fine, you will get the following kind of output in Terminal:

```

robot@robot-pc:~$ sudo pip install --upgrade $TF_BINARY_URL
The directory '/home/robot/.cache/pip/http' or its parent directory is not owned
by the current user and the cache has been disabled. Please check the permissions
and owner of that directory. If executing pip with sudo, you may want sudo's
-H flag.
The directory '/home/robot/.cache/pip' or its parent directory is not owned by t
he current user and caching wheels has been disabled. check the permissions and
owner of that directory. If executing pip with sudo, you may want sudo's -H flag
.
Collecting tensorflow==0.11.0rc1 from https://storage.googleapis.com/tensorflow/
linux/cpu/tensorflow-0.11.0rc1-cp27-none-linux_x86_64.whl
  Downloading https://storage.googleapis.com/tensorflow/linux/cpu/tensorflow-0.1
1.0rc1-cp27-none-linux_x86_64.whl (39.8MB)
    100% |████████████████████████████████| 39.8MB 42kB/s
Collecting mock>=2.0.0 (from tensorflow==0.11.0rc1)
  Downloading mock-2.0.0-py2.py3-none-any.whl (56kB)
    100% |███████████████████████████████| 61kB 122kB/s
Collecting protobuf==3.0.0 (from tensorflow==0.11.0rc1)
  Downloading protobuf-3.0.0-py2.py3-none-any.whl (342kB)
    100% |███████████████████████████████| 348kB 176kB/s
Collecting numpy>=1.11.0 (from tensorflow==0.11.0rc1)
  Downloading numpy-1.11.2-cp27-cp27mu-manylinux1_x86_64.whl (15.3MB)
    67% |███████████████████████████████| 10.4MB 321kB/s eta 0:00:16

```

Figure 2: Installing TensorFlow on Ubuntu 16.04 LTS

If everything has been properly installed on your system, you can check it using a simple test.

Open a Python Terminal, execute the following lines, and check whether you are getting the results shown in the following screenshot. We will look at an explanation of the code in the next section.

```

robot@robot-pc:~$ python
Python 2.7.11+ (default, Apr 17 2016, 14:00:29)
[GCC 5.3.1 20160413] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import tensorflow as tf
>>> hello = tf.constant('Hello, TensorFlow!')
>>> sess = tf.Session()
>>> print(sess.run(hello))
Hello, TensorFlow!
>>> a = tf.constant(12)
>>> b = tf.constant(34)
>>> print(sess.run(a+b))
46
>>>

```

Figure 3: Testing a TensorFlow installation on Ubuntu 16.04 LTS

Here is our hello world code in TensorFlow

```

import tensorflow as tf
hello = tf.constant('Hello, TensorFlow!')
sess = tf.Session()
print (sess.run(hello))
a = tf.constant(12)
b = tf.constant(34)
print(sess.run(a+b))

```

TensorFlow concepts

Before you start programming using TensorFlow functions, you should understand its concepts. Here is the block diagram of TensorFlow concepts demonstrated using addition operation in Tensorflow.

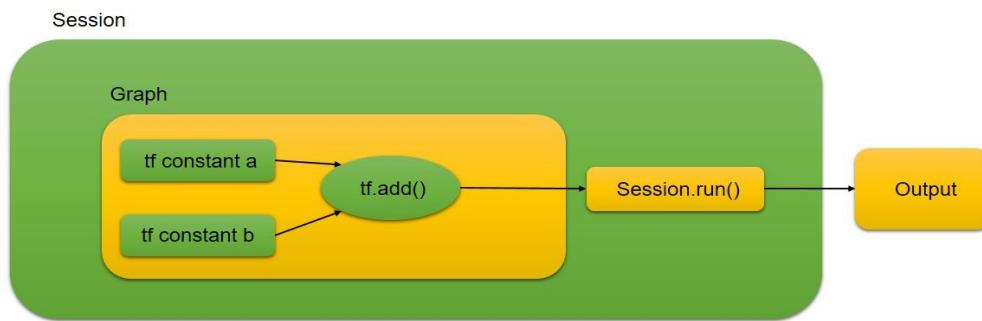


Figure 4: Block diagram of Tensorflow concepts

Let's look at each concepts:

Graph

In TensorFlow, all computations are represented as graphs. A graph consists of nodes. The nodes in a graph are called operations or **ops**. An op or node can take tensors. Tensors are basically typed multi-dimensional arrays. For example, an image can be a tensor. So, in short, the TensorFlow graph has the description of all the computation required.

In the preceding example, the ops of the graphs are as follows:

```
hello = tf.constant('Hello, TensorFlow!')
a = tf.constant(12)
b = tf.constant(34)
```

These `tf.constant()` methods create a constant op that will be added as a node in the graph. You can see how a string and integer are added to the graph.

Session

After building the graph, we have to execute it, right? For computing a graph, we should put it in a session. A `Session` class in TensorFlow places all ops or nodes onto computational devices such as CPU or GPU.

Here is how we create a `Session` object in TensorFlow:

```
| sess = tf.Session()
```

For running the ops in a graph, the `Session` class provides methods to run the entire graph:

```
| print(sess.run(hello))
```

It will execute the op called `hello` and print "Hello, TensorFlow" in Terminal.

Variables

During execution, we may need to maintain the state of the ops. We can do so by using `tf.Variable()`. Let's check out an example declaration of `tf.Variable()`:

This line will create a variable called `counter` and initialize it to scalar value `0`.

```
| state = tf.Variable(0, name="counter")
```

Here are the ops to assign a value to the variable:

```
| one = tf.constant(1)
| update = tf.assign(state, one)
```

If you are working with variables, we have to initialize them all at once using the following function:

```
| init_op = tf.initialize_all_variables()
```

After initialization, we have to run the graph for putting this into effect. We can run the previous ops using the following code:

```
| sess = tf.Session()
| sess.run(init_op)
| print(sess.run(state))
| sess.run(update)
```

Fetches

To fetch the outputs from the graph, we have to execute the `run()` method, which is inside the `Session` object. We can pass the ops to the `run()` method and retrieve the output as tensors:

```
a = tf.constant(12)
b = tf.constant(34)
add = tf.add(a,b)
sess = tf.Session()
result = sess.run(add)
print(result)
```

In the preceding code, the value of `result` will be `12+34`.

Feeds

Until now, we have been dealing with constants and variables. We can also feed tensors during the execution of a graph. Here we have an example of feeding tensors during execution. For feeding a tensor, first we have to define the feed object using the `tf.placeholder()` function. After defining two feed objects, we can see how to use it inside `sess.run()`:

```
x = tf.placeholder(tf.float32)
y = tf.placeholder(tf.float32)

output = tf.mul(input1, input2)

with tf.Session() as sess:
    print(sess.run([output], feed_dict={x:[8.], y:[2.]}))

# output:
# [array([ 16.], dtype=float32)]
```

Writing our first code in TensorFlow

Let's start coding using TensorFlow. We are again going to write basic code that performs matrix operations such as matrix addition, multiplication, scalar multiplication and multiplication with a scalar from 1 to 99. The code is written for demonstrating basic capabilities of TensorFlow, which we have discussed previously.

Here is the code for all these operations:

```
import tensorflow as tf
import time

matrix_1 = tf.Variable([[1,2,3],[4,5,6],[7,8,9]],name="mat1")
matrix_2 = tf.Variable([[1,2,3],[4,5,6],[7,8,9]],name="mat2")

scalar = tf.constant(5)
number = tf.Variable(1, name="counter")

add_msg = tf.constant("\nResult of matrix addition\n")
mul_msg = tf.constant("\nResult of matrix multiplication\n")
scalar_mul_msg = tf.constant("\nResult of scalar multiplication\n")
number_mul_msg = tf.constant("\nResult of Number multiplication\n")

mat_add = tf.add(matrix_1,matrix_2)
mat_mul = tf.matmul(matrix_1,matrix_2)
mat_scalar_mul = tf.mul(scalar,mat_mul)
mat_number_mul = tf.mul(number,mat_mul)

init_op = tf.initialize_all_variables()
sess = tf.Session()
tf.device("/cpu:0")
sess.run(init_op)

for i in range(1,100):
    print "\nFor i =",i
    print(sess.run(add_msg))
    print(sess.run(mat_add))

    print(sess.run(mul_msg))
    print(sess.run(mat_mul))

    print(sess.run(scalar_mul_msg))
    print(sess.run(mat_scalar_mul))

    update = tf.assign(number,tf.constant(i))
    sess.run(update)

    print(sess.run(number_mul_msg))
    print(sess.run(mat_number_mul))

    time.sleep(0.1)

sess.close()
```

As we know, we have to import the `tensorflow` module to access its APIs. We are also importing the `time` module to provide a delay in the loop:

```
import tensorflow as tf
import time
```

Here is how to define variables in TensorFlow. We are defining `matrix_1` and `matrix_2` variables, two 3x3 matrices:

```
| matrix_1 = tf.Variable([[1,2,3],[4,5,6],[7,8,9]],name="mat1")
| matrix_2 = tf.Variable([[1,2,3],[4,5,6],[7,8,9]],name="mat2")
```

In addition to the preceding matrix variables, we are defining a constant and a scalar variable called `counter`. These values are used for scalar multiplication operations. We will change the value of `counter` from 1 to 99, and each value will be multiplied with a matrix:

```
| scalar = tf.constant(5)
| number = tf.Variable(1, name="counter")
```

The following is how we define strings in TF. Each string is defined as a constant.

```
| add_msg = tf.constant("\nResult of matrix addition\n")
| mul_msg = tf.constant("\nResult of matrix multiplication\n")
| scalar_mul_msg = tf.constant("\nResult of scalar multiplication\n")
| number_mul_msg = tf.constant("\nResult of Number multiplication\n")
```

The following are the main ops in the graph doing the computation. The first line will add two matrices, second line will multiply those same two, the third will perform scalar multiplication with one value, and the fourth will perform scalar multiplication with a scalar variable.

```
| mat_add = tf.add(matrix_1,matrix_2)
| mat_mul = tf.matmul(matrix_1,matrix_2)
| mat_scalar_mul = tf.mul(scalar,mat_mul)
| mat_number_mul = tf.mul(number,mat_mul)
```

If we have TF variable declarations, we have to initialize them using the following line of code:

```
| init_op = tf.initialize_all_variables()
```

Here, we are creating a `Session()` object:

```
| sess = tf.Session()
```

This is one thing we hadn't discussed earlier. We can perform the computation on any device according to our priority. It can be a CPU or GPU. Here, you can see that the device is a CPU:

```
| tf.device("/cpu:0")
```

This line of code will run the graph to initialize all variables:

```
| sess.run(init_op)
```

In the following loop, we can see the running of a TF graph. This loop puts each op inside the `run()` method and fetches its results. To be able to see each output, we are putting a delay on the loop:

```
| for i in range(1,100):
|   print "\nFor i =",i
|   print(sess.run(add_msg))
|   print(sess.run(mat_add))
|
|   print(sess.run(mul_msg))
|   print(sess.run(mat_mul))
```

```
print(sess.run(scalar_mul_msg))
print(sess.run(mat_scalar_mul))

update = tf.assign(number,tf.constant(i))
sess.run(update)

print(sess.run(number_mul_msg))
print(sess.run(mat_number_mul))

time.sleep(0.1)
```

After all this computation, we have to release the `session()` object to free up the resources:

```
| sess.close()
```

The following is the output:

```
For i = 99

Result of matrix addition

[[ 2  4  6]
 [ 8 10 12]
 [14 16 18]]

Result of matrix multiplication

[[ 30  36  42]
 [ 66  81  96]
 [102 126 150]]

Result of scalar multiplication

[[150 180 210]
 [330 405 480]
 [510 630 750]]

Result of Number multiplication

[[ 2970  3564  4158]
 [ 6534  8019  9504]
 [10098 12474 14850]]
```

Figure 5: Output of basic TensorFlow code

Image recognition using ROS and TensorFlow

After discussing the basics of TensorFlow, let's start discussing how to interface ROS and TensorFlow to do some serious work. In this section, we are going to deal with image recognition using these two.

There is a simple package to perform image recognition using TensorFlow and ROS. Here is the ROS package to do this:

<https://github.com/qboticslabs/rostensorflow>

This package was forked from <https://github.com/OTL/rostensorflow>. The package basically contains a ROS Python node that subscribes to images from the ROS webcam driver and performs image recognition using TensorFlow APIs. The node will print the detected object and its probability.



This code was developed using TensorFlow tutorials from the following link: https://www.tensorflow.org/versions/r0.11/tutorials/image_recognition/index.html.

The image recognition is mainly done using a model called deep convolution network. It can achieve high accuracy in the field of image recognition. An improved model we are going to use here is Inception-v3 (<https://arxiv.org/abs/1512.00567>).



*This model is trained for the **ImageNet Large Scale Visual Recognition Challenge (ILSVRC)** (<http://image-net.org/challenges/LSVRC/2016/index>) using data from 2012.*

When we run the node, it will download a trained Inception-v3 model to the computer and classify the object according to the webcam images. You can see the detected object's name and its probability in Terminal.

There are a few prerequisites to run this node. Let's go through the dependencies.

Prerequisites

For running a ROS image recognition node, you should install the following dependencies. The first is `cv-bridge`, which helps us convert a ROS image message into OpenCV image data type and vice versa. The second is `cv-camera`, which is one of the ROS camera drivers. Here's how to install them:

```
| $sudo apt-get install ros-kinetic-cv-bridge ros-kinetic-cv-camera
```

The ROS image recognition node

You can download the ROS image recognition package from GitHub; it's also available in the section's code bundle. The `image_recognition.py` program can publish detected results in the `/result` topic, which is of the `std_msgs/String` type and is subscribed to image data from the ROS camera driver from the `/image` (`sensor_msgs/Image`) topic.

So how does `image_recognition.py` work?

First take a look at the main modules imported to this node. As you know, `rospy` has ROS Python APIs. The ROS camera driver publishes ROS image messages, so here we have to import `Image` messages from `sensor_msgs` to handle those image messages. To convert a ROS image to the OpenCV data type and vice versa, we need `cv_bridge` and, of course, the `numpy`, `tensorflow`, and `tensorflow_imagenet` modules to classify of images and download the Inception-v3 model from tensorflow.org. Here are the imports:

```
import rospy
from sensor_msgs.msg import Image
from std_msgs.msg import String
from cv_bridge import CvBridge
import cv2
import numpy as np
import tensorflow as tf
from tensorflow.models.image.imagenet import classify_image
```

The following code snippet is the constructor for a class called `RosTensorFlow()`:

```
class RosTensorFlow():
    def __init__(self):
```

The constructor call has the API for downloading the trained Inception-v3 model from tensorflow.org:

```
|     classify_image.maybe_download_and_extract()
```

Now, we are creating a TensorFlow `Session()` object, then creating a graph from a saved `GraphDef` file, and returning a handle for it. The `GraphDef` file is available in the code bundle.

```
|     self._session = tf.Session()
|     classify_image.create_graph()
```

This line creates a `cv_bridge` object for the ROS-OpenCV image conversion:

```
|     self._cv_bridge = CvBridge()
```

Here are the subscriber and publisher handles of the node:

```
self._sub = rospy.Subscriber('image', Image, self.callback,
queue_size=1)
self._pub = rospy.Publisher('result', String, queue_size=1)
```

Here are some parameters used for recognition thresholding and the number of top predictions:

```
self.score_threshold = rospy.get_param('~score_threshold', 0.1)
self.use_top_k = rospy.get_param('~use_top_k', 5)
```

Here is the image call back in which a ROS image message is converted to OpenCV data type:

```
def callback(self, image_msg):
    cv_image = self._cv_bridge.imgmsg_to_cv2(image_msg, "bgr8")
    image_data = cv2.imencode('.jpg', cv_image)[1].tostring()
```

The following code runs the softmax tensor by feeding image_data as input to the graph. The 'softmax:0' part is a tensor containing the normalized prediction across 1,000 labels.

```
softmax_tensor =
self._session.graph.get_tensor_by_name('softmax:0')
```

The 'DecodeJpeg/contents:0' line is a tensor containing a string providing JPEG encoding of the image:

```
predictions = self._session.run(
    softmax_tensor, {'DecodeJpeg/contents:0': image_data})
predictions = np.squeeze(predictions)
```

The following section of code will look for a matching object string and its probability and publish it through the topic called /result:

```
node_lookup = classify_image.NodeLookup()
top_k = predictions.argsort()[-self.use_top_k:][::-1]
for node_id in top_k:
    human_string = node_lookup.id_to_string(node_id)
    score = predictions[node_id]
    if score > self.score_threshold:
        rospy.loginfo('%s (score = %.5f)' % (human_string,
score))
        self._pub.publish(human_string)
```

The following is the main code of this node. It simply initializes the class and calls the main() method inside the RostensorFlow() object. The main method will spin() the node and execute a callback whenever an image comes into the /image topic.

```
def main(self):
    rospy.spin()
if __name__ == '__main__':
    rospy.init_node('rostensorflow')
    tensor = RostensorFlow()
    tensor.main()
```

Running the ROS image recognition node

Let's go through how we can run the image recognition node.

First, you have to plug a UVC webcam, which we used in *Chapter 20, Face Detection and Tracking Using ROS, OpenCV and Dynamixel Servos* Run `roscore`:

```
| $ roscore
```

Run the webcam driver:

```
| $ rosrun cv_camera cv_camera_node
```

Run the image recognition node, simply using the following command:

```
| $ python image_recognition.py image:=/cv_camera/image_raw
```

When we run the recognition node, it will download the inception model and extract it into the `/tmp/imagenet` folder. You can do it manually by downloading inception-v3 from the following link:

<http://download.tensorflow.org/models/image/imagenet/inception-2015-12-05.tgz>

You can copy this file into the `/tmp/imagenet` folder:



Figure 6: Inception model in the `/tmp/imagenet` folder

You can see the result by echoing the following topic:

```
| $ rostopic echo /result
```

You can view the camera images using following command:

```
| $ rosrun image_view image_view image:= /cv_camera/image_raw
```

Here is the output from the recognizer. The recognizer detects the device as a cell phone.

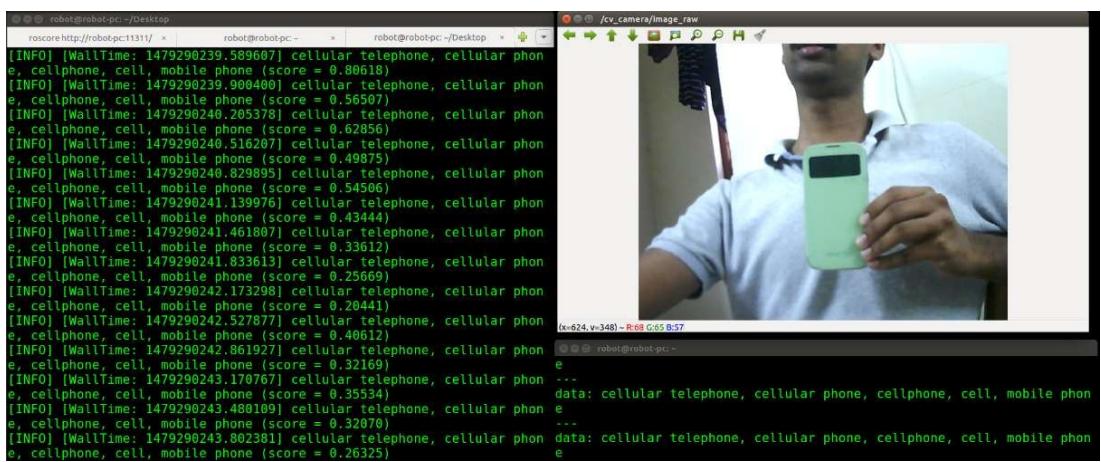


Figure 7: Output from recognizer node

In the next detection, the object is detected as a water bottle:

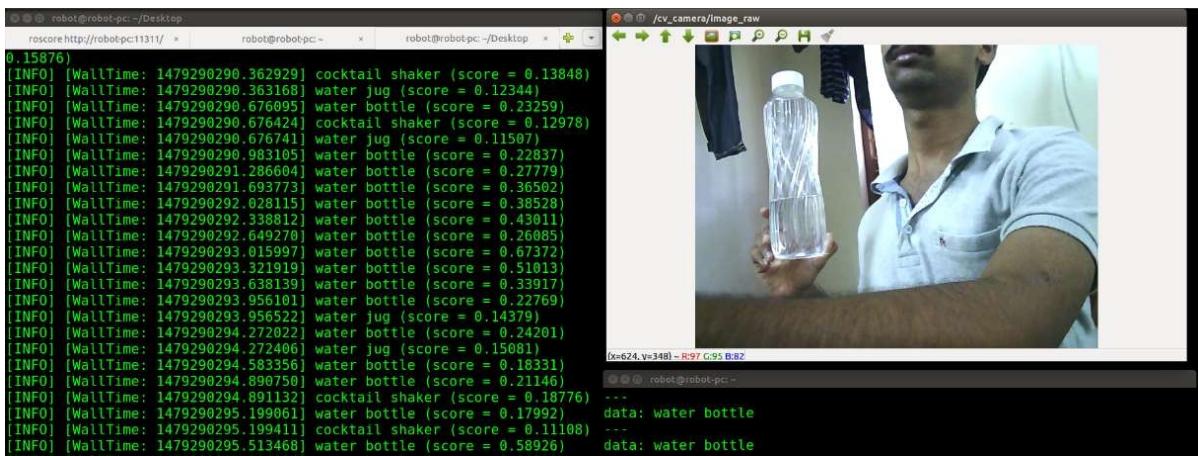


Figure 8: Output from recognizer node detecting a water bottle

Introducing to scikit-learn

Until now, we have been discussing deep neural networks and some of their applications in robotics and image processing. Apart from neural networks, there are a lot of models available to classify data and predict using them.

Generally, in machine learning, we can teach the model using supervised or unsupervised learning. In supervised learning, we train the model against a dataset, but in unsupervised, it discover groups of related observations called clusters instead.

There are lots of libraries available for working with other machine learning algorithms. We'll look at one such library called `scikit-learn`; we can play with most of the standard machine learning algorithms and implement our own application using it.

`scikit-learn` (<http://scikit-learn.org/>) is one of the most popular open source machine learning libraries for Python. It provides an implementation of algorithms for performing classification, regression, and clustering. It also provides functions to extract features from a dataset, train the model, and evaluate it.

`scikit-learn` is an extension of a popular scientific python library called **SciPy** (<https://www.scipy.org/>). `scikit-learn` strongly binds with other popular Python libraries, such as NumPy and matplotlib. Using NumPy, we can create efficient multidimensional arrays, and using matplotlib, we can visualize the data.

`scikit-learn` is well documented and has wrappers for performing **Support Vector Machine (SVM)** and natural language processing functions.

Installing scikit-learn on Ubuntu 16.04 LTS

Installing scikit-learn on Ubuntu is easy and straightforward. You can install it either using `apt-get install` or `pip`.

Here is the command to install scikit-learn using `apt-get install`:

```
| $ sudo apt-get install python-sklearn
```

We can install it using `pip` using the following command:

```
| $ sudo pip install scikit-learn
```

After installing scikit-learn, we can test the installation by doing following commands in Python Terminal.

```
| >>> import sklearn  
| >>> sklearn.__version__  
| '0.17'
```

Congratulations, you have successfully set up scikit-learn!

Introducing to SVM and its application in robotics

We have set up scikit-learn, so what is next? Actually, we are going to discuss a popular machine learning technique called SVM and its applications in robotics. After discussing the basics, we can implement a ROS application using SVM.

So what is SVM? SVM is a supervised machine learning algorithm that can be used for classification or regression. In SVM, we plot each data item in n -dimensional space along with its value. After plotting, it performs a classification by finding a hyper-plane that separates those data points. This is how the basic classification is done!

SVM can perform better for small datasets, but it does not do well if the dataset is very large. Also, it will not be suitable if the dataset has noisy data.

SVM is widely used in robotics, especially in computer vision for classifying objects and also for classifying various kinds of sensor data in robots.

In the next section, we will see how we can implement SVM using scikit-learn and make an application using it.

Implementing an SVM-ROS application

Following shows the aim of this project.

In this application we are going to classify a sensor data in three ways. The sensor values are assumed to be in between 0 to 30,000 and we are having a dataset which is having the sensor value mapping. For example, for a sensor value, you can assign the value belongs to 1, 2, or 3. To test the SVM, we are making another ROS node called virtual sensor node, which can publish value in between 0 to 30000. The trained SVM model can able to classify the virtual sensor value. This method can be adopted for any kind of sensors for classifying its data.

Before embedding SVM in ROS, here's some basic code in Python using `sklearn` to implement SVM.

The first thing is importing the `sklearn` and `numpy` modules. The `sklearn` module has the `svm` module, which is going to be used in this code, and `numpy` is used for making multi-dimensional arrays:

```
| from sklearn import svm  
| import numpy as np
```

For training SVM, we need an input (predictor) and output (target); here, `x` is the input and `y` is the required output:

```
| x = np.array([[-1, -1], [-2, -1], [1, 1], [2, 1]])  
| y = np.array([1, 1, 2, 2])
```

After defining `x` and `y`, just create an instance of **SVM Classification (SVC) object**.

Feed `x` and `y` to the SVC object for training the model. After feeding `x` and `y`, we can feed an input that may not be in `x`, and it can predict the `y` value corresponding to the given input:

```
| model = svm.SVC(kernel='linear', C=1, gamma=1)  
| model.fit(x,y)  
| print(model.predict([-0.8,-1]))
```

The preceding code will give an output of 1.

Now, we are going to implement a ROS application that does the same thing. Here, we are creating a virtual sensor node that can publish random values from 0 to 30,000. The ROS-SVM node will subscribe to those values and classify them using the preceding APIs. The learning in SVM is done from a CSV data file.

You can view the complete application package in section code; it's called `ros_ml`. Inside the `ros_ml/scripts` folder, you can see nodes such as `ros_svm.py` and `virtual_sensor.py`.

First, let's take a look at the virtual sensor node. The code is very simple and self-explanatory. It simply generates a random number between 0 and 30,000 and publishes it to the `/sensor_read` topic:

```
|#!/usr/bin/env python  
|import rospy
```

```

from std_msgs.msg import Int32
import random

def send_data():
    rospy.init_node('virtual_sensor', anonymous=True)
    rospy.loginfo("Sending virtual sensor data")
    pub = rospy.Publisher('sensor_read', Int32, queue_size=1)
    rate = rospy.Rate(10) # 10hz

    while not rospy.is_shutdown():
        sensor_reading = random.randint(0,30000)
        pub.publish(sensor_reading)
        rate.sleep()

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

```

The next node is `ros_svm.py`. This node reads from a data file from a data folder inside the `ros_ml` package. The current data file is named `pos_readings.csv`, which contains the sensor and target values. Here is a snippet from that file:

```

5125,5125,1
6210,6210,1
.....
10125,10125,2
6410,6410,2
5845,5845,2
.....
14325,14325,3
16304,16304,3
18232,18232,3
.....

```

The `ros_svm.py` node reads this file, trains the SVC, and predicts each value from the virtual sensor topic. The node has a class called `classify_Data()`, which has methods to read the CSV file and train and predict it using scikit APIs.

We'll step through how these nodes are started:

Start `roscore`:

```
| $ roscore
```

Switch to the script folder of `ros_ml`:

```
| $ roscd ros_ml/scripts
```

Run the ROS SVM classifier node:

```
| $ python ros_svm.py
```

Run the virtual sensor in another Terminal:

```
| $ rosrun ros_ml virtual_sensor.py
```

Here is the output we get from the SVM node:

```
[INFO] [WallTime: 1478795301.108411] Input = 3578 Prediction = 2
[INFO] [WallTime: 1478795301.208386] Input = 25929 Prediction = 3
[INFO] [WallTime: 1478795301.308298] Input = 2692 Prediction = 1
[INFO] [WallTime: 1478795301.408412] Input = 18532 Prediction = 3
[INFO] [WallTime: 1478795301.508338] Input = 23827 Prediction = 3
[INFO] [WallTime: 1478795301.608325] Input = 9525 Prediction = 2
[INFO] [WallTime: 1478795301.708314] Input = 16329 Prediction = 3
[INFO] [WallTime: 1478795301.808376] Input = 6160 Prediction = 2
[INFO] [WallTime: 1478795301.908370] Input = 3789 Prediction = 2
[INFO] [WallTime: 1478795302.008349] Input = 23066 Prediction = 3
[INFO] [WallTime: 1478795302.108361] Input = 11637 Prediction = 3
[INFO] [WallTime: 1478795302.208381] Input = 29517 Prediction = 3
[INFO] [WallTime: 1478795302.308438] Input = 25526 Prediction = 3
[INFO] [WallTime: 1478795302.408303] Input = 9828 Prediction = 2
[INFO] [WallTime: 1478795302.508341] Input = 14718 Prediction = 3
[INFO] [WallTime: 1478795302.608403] Input = 19248 Prediction = 3
[INFO] [WallTime: 1478795302.708343] Input = 29411 Prediction = 3
[INFO] [WallTime: 1478795302.808402] Input = 8170 Prediction = 2
[INFO] [WallTime: 1478795302.908326] Input = 8343 Prediction = 2
[INFO] [WallTime: 1478795303.008411] Input = 18324 Prediction = 3
[INFO] [WallTime: 1478795303.108378] Input = 14846 Prediction = 3
[INFO] [WallTime: 1478795303.208400] Input = 20526 Prediction = 3
[INFO] [WallTime: 1478795303.308316] Input = 18781 Prediction = 3
[INFO] [WallTime: 1478795303.408407] Input = 1716 Prediction = 1
[INFO] [WallTime: 1478795303.508380] Input = 28444 Prediction = 3
[INFO] [WallTime: 1478795303.608389] Input = 7272 Prediction = 2
[INFO] [WallTime: 1478795303.708434] Input = 22650 Prediction = 3
```

Figure 9: ROS - SVM node output

Questions

- What basically is a neural network?
- What is deep learning?
- Why do we use TensorFlow for deep learning?
- What are the main concepts in TensorFlow?
- Why do we use scikit-learn for machine learning?

Summary

In this chapter, we mainly discussed the various machine learning techniques and libraries that can be interfaced with ROS. We started with the basics of machine learning and deep learning. Then we started working with TensorFlow, which is an open source Python library mainly for performing deep learning. We discussed basic code using TensorFlow and later combined those capabilities with ROS for an image recognition application. After discussing Tensorflow and deep learning, we discussed another Python library called scikit-learn used for machine learning applications. We saw what SVM is and saw how to implement it using scikit-learn. Later, we implemented a sample application using ROS and scikit-learn for classifying sensor data.

In the next chapter, we will discuss ROS on Android and MATLAB.

ROS on MATLAB and Android

As you all know, MATLAB is one of the powerful numerical computation tools available for research, education, and commercial applications. Also, we don't need any more explanation of the Android OS, which is one of the most popular mobile operating systems. In this chapter, we will mainly work with the ROS interface of MATLAB and Android.

By combining the capabilities of MATLAB and Android in ROS, we can create powerful robotic projects. Here are the main topics we will discuss in this chapter:

- Getting started with the ROS-MATLAB interface
- Communicating from MATLAB to a ROS network
- Controlling a ROS robot from MATLAB
- Getting started with Android and its ROS interface
- Installing the ROS-Android interface
- Playing with ROS-Android applications
- ROS-Android code walk through
- Creating a basic application using the ROS-Android interface

Getting started with the ROS-MATLAB interface

The ROS-MATLAB interface is a useful interface for researchers and students for prototyping their robot algorithms in MATLAB and testing it on ROS-compatible robots. The robotics system toolbox in MATLAB provides the interface between MATLAB and ROS. We can prototype our algorithm and test it on a ROS-enabled robot or in robot simulators such as Gazebo and V-REP (<http://www.coppeliarobotics.com/downloads.html>). From MATLAB, we can publish or subscribe to a topic, such as a ROS node, and we can make it a ROS master. The MATLAB-ROS interface has most of the ROS functionalities that we need.

Here is a block diagram shows how MATLAB is communicating with a robot which is running on ROS.

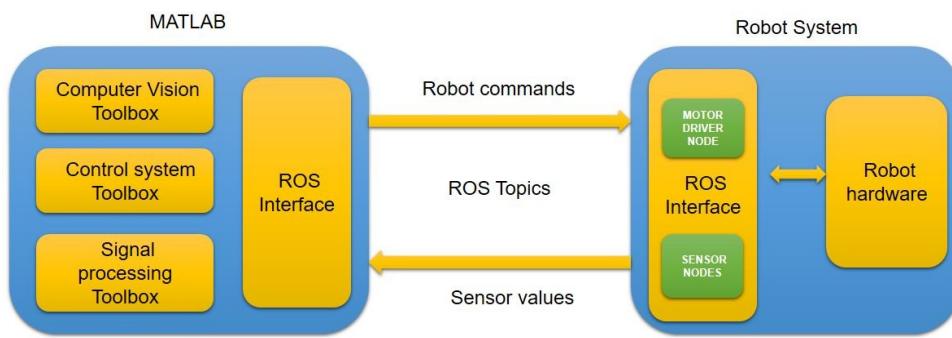


Figure 1: The MATLAB - Robot communication diagram

From the preceding figure, you can understand, MATLAB is equipped with powerful toolboxes such as computer vision, control system and signal processing. We can fetch the data from robot through ROS interface and process using these toolbox. After processing sensor data, we can also send control commands to robot. These communications are all occurs via ROS-MATLAB interface.

Here are some of the main features of the ROS - MATLAB interface:

- It can seamlessly connect to another ROS network and can work with various robot capabilities. We can also visualize the robot sensor data in MATLAB.
- We can create ROS nodes, publishers, and subscribers directly from MATLAB and Simulink.
- We can send and receive topics using ROS messages and ROS custom messages.
- It has full support for ROS services.
- We get ROS functionality from any OS platform (Windows, Linux, or Mac).
- We can make MATLAB the ROS master.
- We can import ROS bag files and analyze, visualize, and post-process logged data.
- It provides full-fledged communication with robot simulators such as Gazebo and V-REP for offline programming.
- We can create standalone ROS C++ nodes from a Simulink model.

Setting Robotics Toolbox in MATLAB

Here is the link to download a trial or purchase the Robotics Toolbox in MATLAB (<https://in.mathworks.com/products/robotics.html>). This toolbox is compatible with MATLAB version 2013 onward. If you don't have MATLAB, you can test the chapter's code using a trial version; if you have it, buy or download a trial version of Robotic Toolbox.

Basic ROS functions in MATLAB

After setting up Robotics Toolbox in MATLAB, we can start working on the important functions of MATLAB that are used to interact with a ROS network. Let's look at them with examples.

Initializing a ROS network

Before running a ROS node, we have to run the `roscore` command, right? The `roscore` command will start a ROS master, and other ROS nodes can find each other through it. In MATLAB, instead of the `roscore` command, we can use the `rosinit` function to start a ROS master.

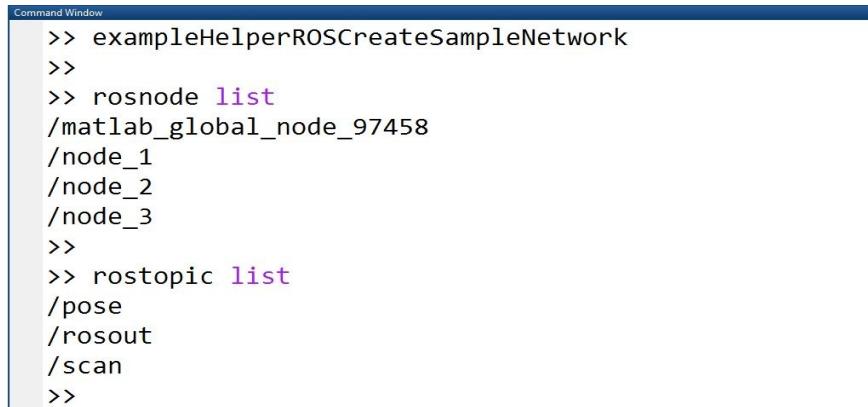
```
>> rosinit
Initializing ROS master on http://DESKTOP-IOQ6CMI:11311/.
Initializing global node /matlab_global_node_97458 with NodeURI http://DESKTOP-IOQ6CMI:53329/
>>
>>
>>
```

Figure 2 : The `rosinit` function in MATLAB

The `rosinit` function can start a ROS master and a global node that is connected to the master. Here, we can see that MATLAB itself can act as a ROS master and other nodes can connect to it. We can also connect to a ROS network from MATLAB. We'll cover that in the next section. In such a setup, the ROS master is running on a different system, either on a ROS robot or ROS PC. Let's try some of the ROS commands in MATLAB to list ROS nodes, topics, and all that. The good thing about the MATLAB - ROS interface is that the commands of Robotics Toolbox are similar to the actual ROS bash commands. Let's go through a few commands to list out ROS parameters.

Listing ROS nodes, topics, and messages

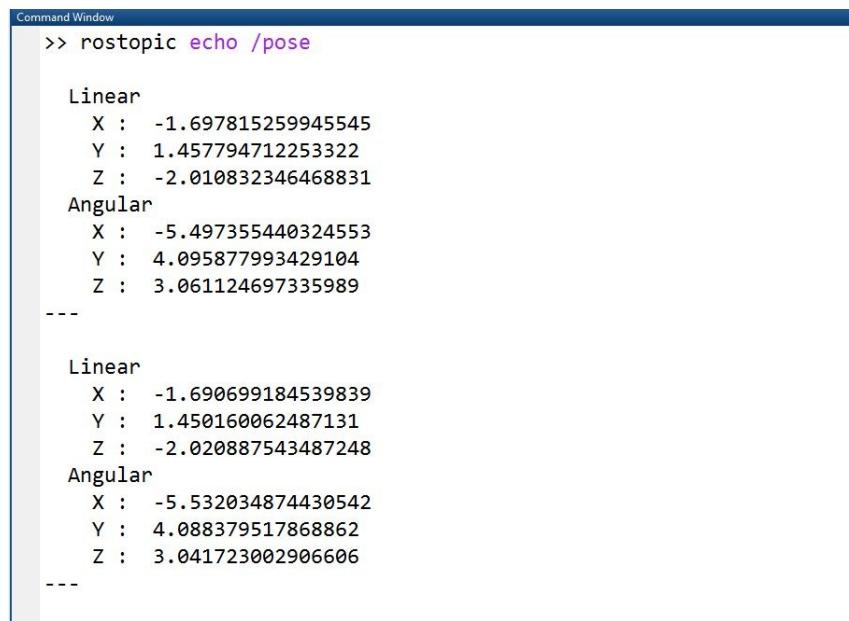
The commands to inspect nodes, topics, and messages are similar to ROS bash commands. MATLAB provides a command to start sample ROS nodes that can publish topics. You can just call `exampleHelperROSCreateSampleNetwork` to start these nodes.



```
Command Window
>> exampleHelperROSCreateSampleNetwork
>>
>> rosnode list
/matlab_global_node_97458
/node_1
/node_2
/node_3
>>
>> rostopic list
/pose
/rosout
/scan
>>
```

Figure 3: ROS-MATLAB commands

You can see that the usage of `rosnode` and `rostopic` is the same as with real ROS commands. You can even echo the `rostopic` using `rostopic echo /topic_name`. Here is one example, in which we are echoing a topic called `/pose`:



```
Command Window
>> rostopic echo /pose

Linear
X : -1.697815259945545
Y : 1.457794712253322
Z : -2.010832346468831
Angular
X : -5.497355440324553
Y : 4.095877993429104
Z : 3.061124697335989
---

Linear
X : -1.690699184539839
Y : 1.450160062487131
Z : -2.020887543487248
Angular
X : -5.532034874430542
Y : 4.088379517868862
Z : 3.041723002906606
---
```

Figure 4: ROS topic echo output

You can get the complete list of ROS commands in MATLAB using the `help` command.

Here is the syntax for doing so:

```
| >> help robotics.ros
```

This is the screenshot of the list of commands with MATLAB for ROS:

Command Window

```
>> help robotics.ros
ros (Robot Operating System)
  rosinit          - Initialize the ros system
  rosshutdown      - Shut down the ros system

  rosmessage       - Create a ros message
  rospublisher     - Create a ros publisher
  rossubscriber    - Create a ros subscriber
  rossvcclient     - Create a ros service client
  rossvcserver     - Create a ros service server
  rosactionclient   - Create a ros action client
  rostype           - View available ros message types

  rosaction         - Get information about actions in the ros network
  rosmsg            - Get information about messages and message types
  rosnode           - Get information about nodes in the ros network
  rosservice         - Get information about services in the ros network
  rostopic          - Get information about topics in the ros network

  rosbag             - Open and parse a rosbag log file
  rosparam           - Get and set values on the parameter server
  rosrate            - Execute loop at fixed frequency using ros time
  rostf              - Receive, send, and apply ros transformations

  rosduration        - Create a ros duration object
  rostime            - Access ros time functionality
```

Figure 5: List of ROS-MATLAB commands

Communicating from MATLAB to a ROS network

We have worked with some MATLAB commands and we've understood that we can communicate with ROS from MATLAB. But the previous commands were executed in a MATLAB terminal by making MATLAB the ROS master. But what do we do when we need to communicate with a ROS network or a ROS-based robot? The method is simple.

Assuming your PC has MATLAB and the ROS PC/robot is connected to the same network. It can be connected either through LAN or Wi-Fi. If the PC and robot are connected to the same network, both should have identical IP addresses. The first step is to find each device's IP address.

If your MATLAB installation is in Windows, you can open Command Prompt window by simply searching for `cmd` in the search window; then, enter the `ipconfig` command. This will list the network adapters and their details:

```
Wireless LAN adapter Wi-Fi:  
  
    Connection-specific DNS Suffix . :  
    Link-local IPv6 Address . . . . . : fe80::b05d:3405:9b99:8736%9  
    IPv4 Address . . . . . : 192.168.1.101  
    Subnet Mask . . . . . : 255.255.255.0  
    Default Gateway . . . . . : 192.168.1.1
```

Figure 6: Wi-Fi adapter details and its IP in a MATLAB system

Here you can see that the PC running MATLAB and the ROS system are connected through Wi-Fi, and the IP is marked. If you are using MATLAB from Linux, you can use the `ifconfig` command instead of `ipconfig`. You can also get the IP of the ROS-running PC, which could be a Linux PC, using the same command.

```
wlx00177c2e2869 Link encap:Ethernet HWaddr 00:17:7c:2e:28:69  
    inet addr:192.168.1.102 Bcast:192.168.1.255 Mask:255.255.255.0  
    inet6 addr: fe80::24f:8bd5:fb19:828f/64 Scope:Link  
        UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1  
        RX packets:953 errors:0 dropped:0 overruns:0 frame:0  
        TX packets:391 errors:0 dropped:0 overruns:0 carrier:0  
        collisions:0 txqueuelen:1000  
        RX bytes:115747 (115.7 KB) TX bytes:147426 (147.4 KB)
```

Figure 7: Wi-Fi adapter details and IP of ROS system

So in this case, the IP address of the MATLAB system is `192.168.1.101` and that of the ROS system is `192.168.1.102`. Here is how the network looks like:

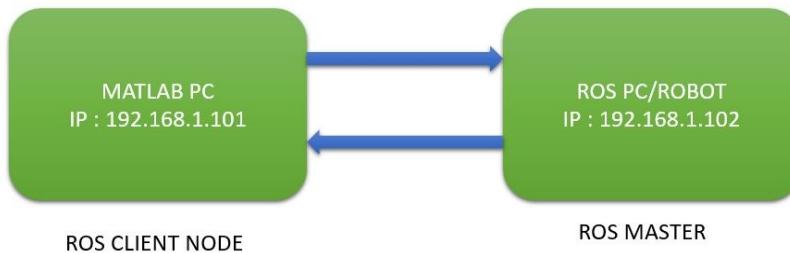


Figure 8: Connecting MATLAB to a ROS network

Connecting from MATLAB to the ROS network is pretty easy. First, we have to set the `ROS_MASTER_URI` variable, which is the IP of the ROS PC/robot where the ROS master is running. You have to mention the port along with the IP; the default port is `11311`.

Before connecting to the ROS network, be sure that you run `roscore` on the ROS PC/robot. MATLAB can connect to the ROS network if there is a ROS master running on it.

The following command helps us connect to the ROS network:

```

>> setenv('ROS_MASTER_URI','http://192.168.1.102:11311')
>> rosinit

```

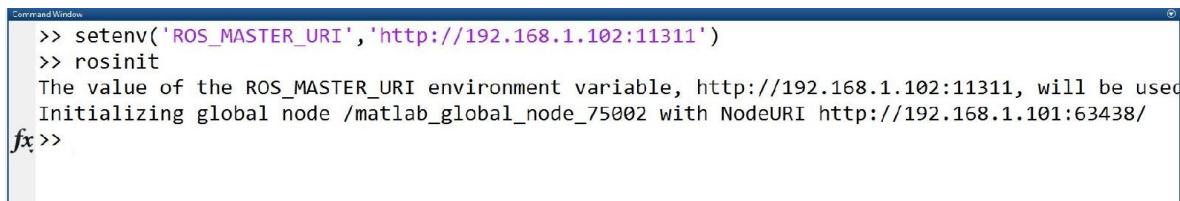


Figure 9: Connecting to ROS network

You can also do this using following command:

```

>> rosinit('192.168.1.102', 'NodeHost', '192.168.1.101')

```

Here, the first argument is the ROS network IP and next one is the IP of the host. If the connection is successful, we will get a message like in preceding screenshot.

After connecting to the network, run an example node on the ROS PC/robot. You can use following node for testing:

```

$ rosrun roscpp_tutorials talker

```

This node basically publishes string data (`std_msgs/String`) to the `/chatter` topic. You can see the node output from the following screenshot:

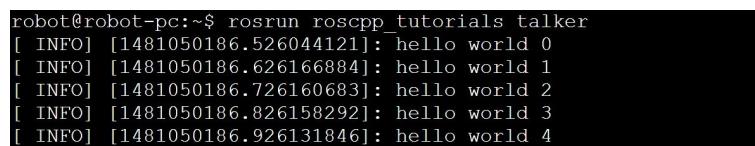


Figure 10: roscpp talker node

Now list the topics in MATLAB and see the magic!

```
| >> rostopic list
```

You will see something like the following screenshot:

```
| >> rostopic list  
| /chatter  
| /rosout  
| /rosout_agg  
fx>> |
```

Figure 11: roscpp talker node

We can also publish values from MATLAB to ROS. Let's see how.

This will connect to the ROS network:

```
| >> setenv('ROS_MASTER_URI', 'http://192.168.1.102:11311')  
| >> rosinit
```

This will create a handle for the ROS publisher. The publisher topic name is `/talker` and message type is `std_msgs/String`.

```
| >> chatpub = rospublisher('/talker', 'std_msgs/String');
```

This line will create a new message definition:

```
| >> msg = rosmessage(chatpub);
```

Here, we are putting data into the message:

```
| >> msg.Data = 'Hello, From Matlab';
```

Now let's send the message through the topic:

```
| >> send(chatpub, msg);
```

With this command, we are latching the message to the topic:

```
| >> latchpub = rospublisher('/talker', 'IsLatching', true);
```

After executing these commands in MATLAB, check the topic list from the ROS PC and echo it. You will get the same message, like this:

```
robot@robot-pc:~$ rostopic list  
/rosout  
/rosout_agg  
/talker  
robot@robot-pc:~$ robot@robot-pc:~$ rostopic echo /talker  
data: Hello, From Matlab  
---
```

Figure 12: Listing rostopic from MATLAB on a ROS PC

Controlling a ROS robot from MATLAB

Here is an interesting MATLAB GUI application that uses ROS APIs to remotely control a robot. The final application will look like the following:

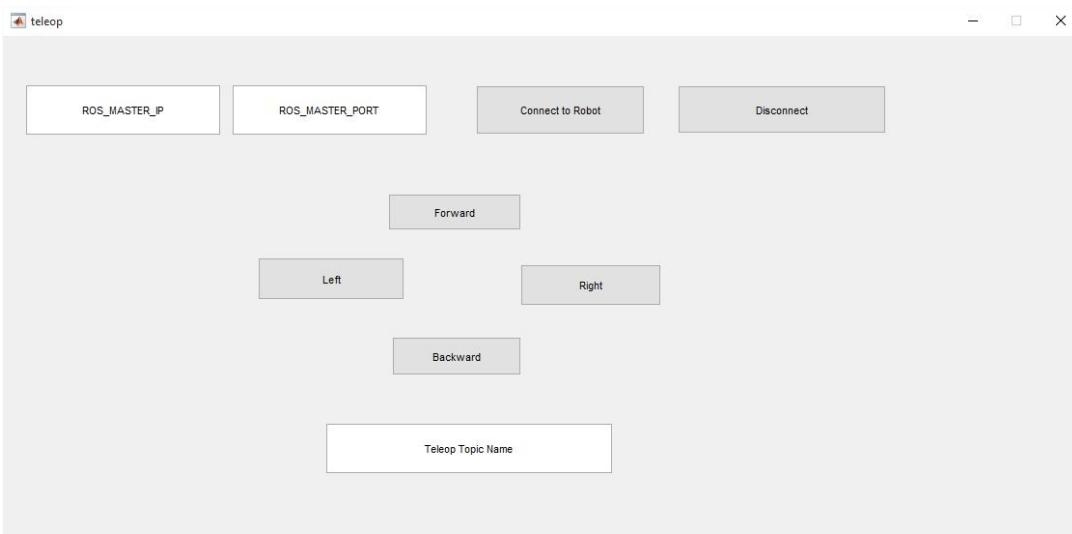


Figure 13: MATLAB-ROS application GUI

In this application, we can put in the ROS master IP, port, and the teleop topic of the robot in its GUI itself. When we press the connect button, the MATLAB application will connect to the ROS network. Now, we can move the robot by pressing the Forward, Backward, Left, and Right buttons.

Here is the design block diagram of this application:

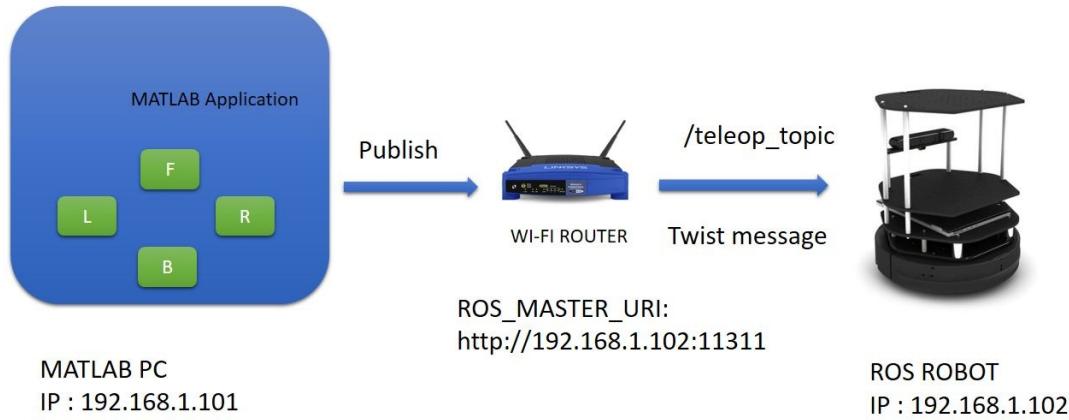


Figure 14: MATLAB-ROS application design block diagram

So let's look at how we can build an application like this.

Here are some of the frequently asking questions in ROS-MATLAB interface

1. How to run multiple ROS nodes in MATLAB?

Yes, we can run multiple ROS nodes in MATLAB. The following command in MATLAB

will give you an example to do it.

```
| >>>openExample('robotics/RunMultipleROSNodesToPerformDifferentTasksExample')
```

2. Does MATLAB support launch files?

No, there is no XML kind launch files in MATLAB, but we can start and end nodes in a MATLAB script which will work almost like a launch file.

3. What features exist in both MATLAB and ROS?

For example plotting data, any recommendations for the use of each?

4. There are plotting tools available in ROS and MATLAB. The tools such as `rqt_gui` help to plot different kind of data which are coming as topics. If you want to play with data and its analysis, MATLAB is the good choice.

Designing the MATLAB GUI application

MATLAB provides easy ways to design a GUI. Here is one popular method to create a GUI using **GUIDE development environment (GUIDE)** (<https://in.mathworks.com/discovery/matlab-gui.html>). To start GUIDE in MATLAB, just type `guide` in your MATLAB command line:

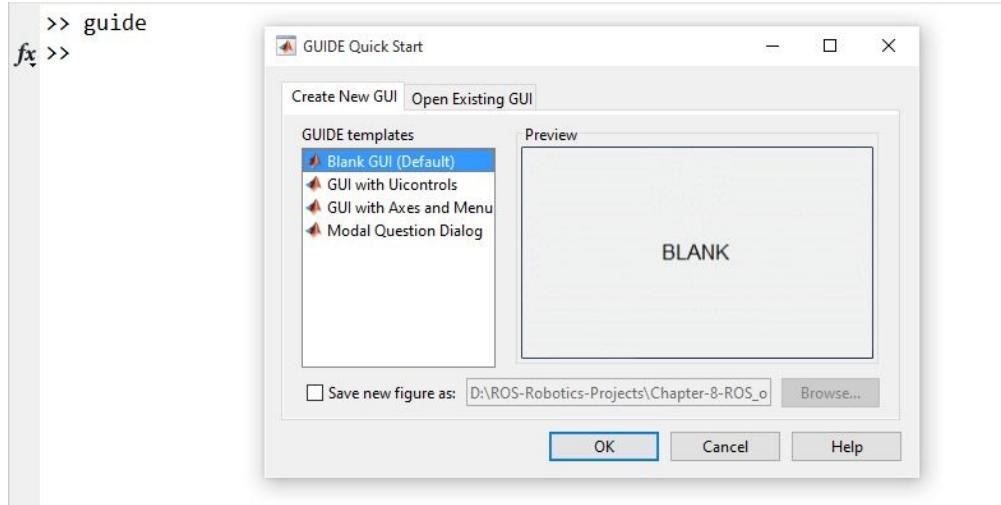


Figure 15: MATLAB GUI wizard

You can select a Blank GUI and press OK. You will get a blank GUI, and you can add buttons and text boxes according to your requirements. The following figure shows the basic GUI elements in GUIDE. You can see an empty GUI form and toolbox. We can just drag components from the toolbox to the form. For example, if we need a push button and text edit box, we can just drag and drop those items to the empty form and align them on the form:

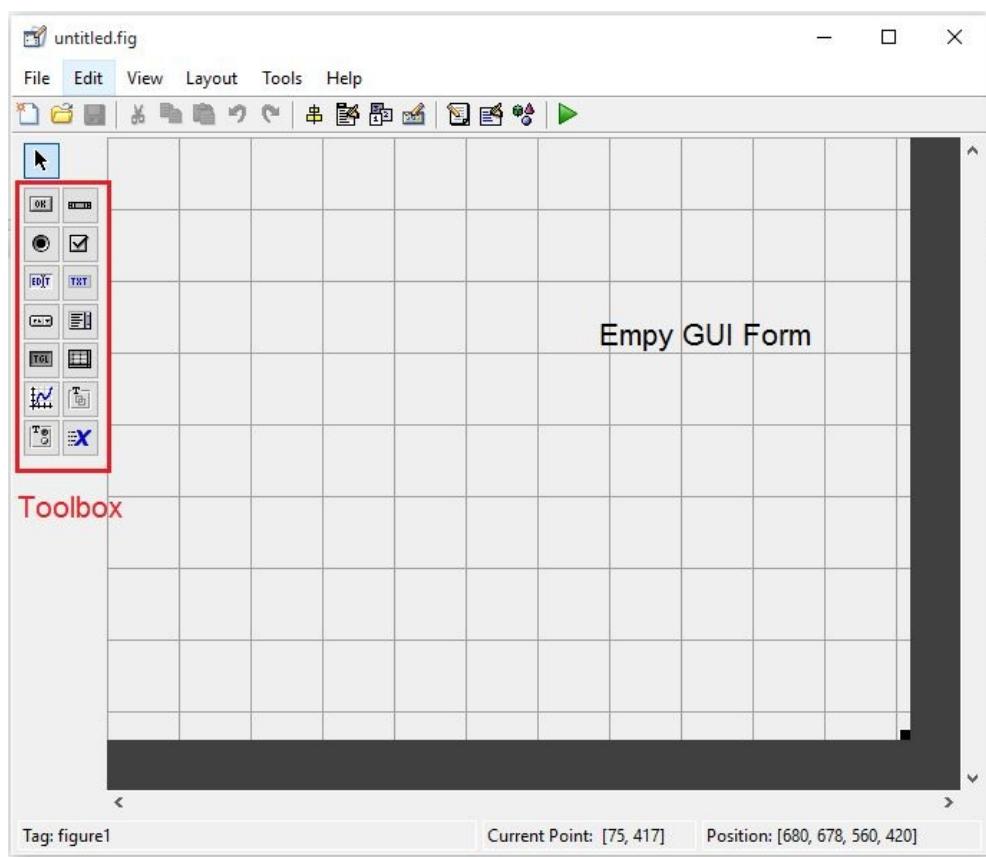


Figure 16: MATLAB GUI empty form

After assigning buttons, we have to generate a callback function for them, which will be executed once the button is pressed (or the text edit box is changed). You can create the callback function from the option highlighted in the following figure. When you save it, you will get a *.m file too. This is the MATLAB code file, in which we are going to write the callback functions.

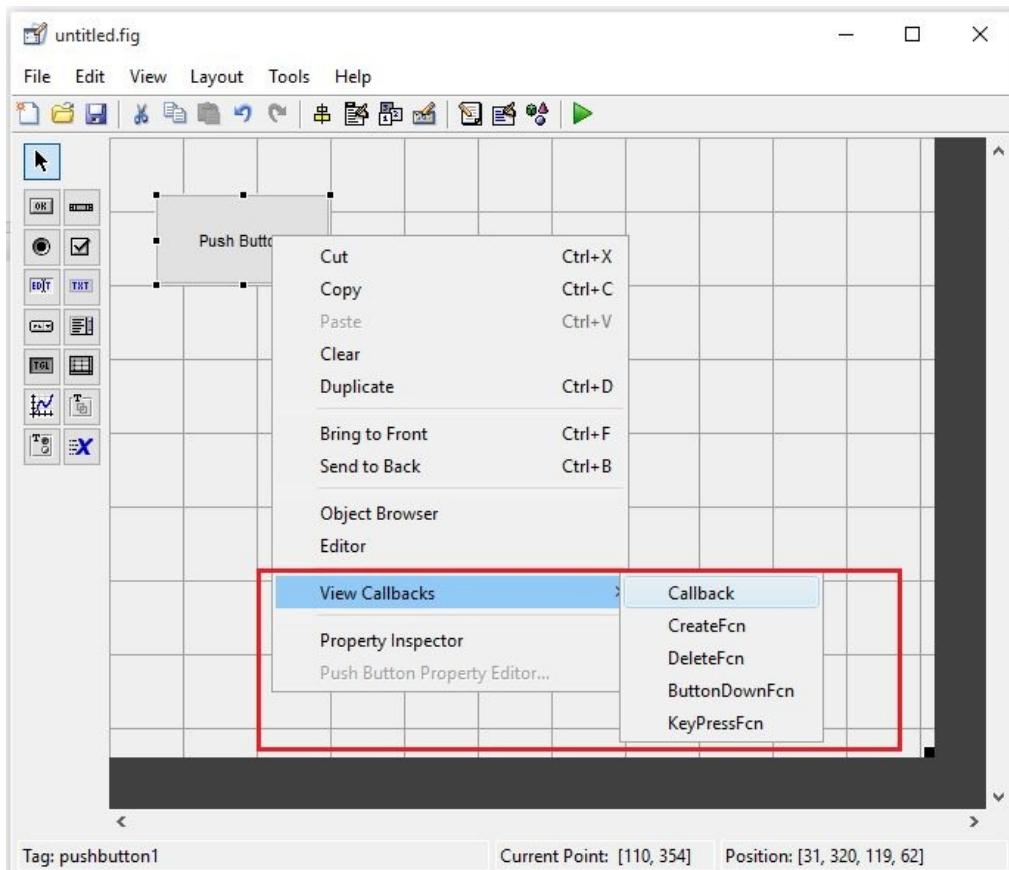


Figure 17: Inserting callback functions

The preceding figure shows how to insert a callback for each button. Right-click on the button and press the Callback option. You'll see the empty callback function for this button:

```
% --- Executes on button press in pushbutton1.
function pushbutton1_Callback(hObject, eventdata, handles)
% hObject    handle to pushbutton1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
```

Figure 18: An empty callback function

In the next section, we will discuss the content of each callback of the application.

Explaining callbacks

You can get the complete code from chapter_8_codes/Matlab/teleop.m. Let's look at the content and functions of each callback. The first callback we are going to see is for the ROS MASTER IP edit box:

```
function edit1_Callback(hObject, eventdata, handles)
global ros_master_ip
ros_master_ip = get(hObject,'String')
```

When we enter an IP address from the ROS network in this edit box, it will store the IP address as a string in a global variable called `ros_master_ip`. If you don't enter the IP, then a default value is loaded, defined outside the callback.

Here are the initial values of `ros_master_ip`, `ros_master_port`, and `teleop topic`.

```
ros_master_ip = '192.168.1.102';
ros_master_port = '11311';
teleop_topic_name = '/cmd_vel_mux/input/teleop';
```

If we don't provide any values in the textbox, these initial values get loaded.

The next GUI element is for obtaining the ROS MASTER PORT. This is the callback of this edit box:

```
function edit2_Callback(hObject, eventdata, handles)
global ros_master_port
ros_master_port = get(hObject,'String')
```

In this function too, the port from the edit box is stored as string type in a global variable called `ros_master_port`.

The next edit box is for obtaining the `teleop_topic_name`. Here is its callback function definition:

```
function edit3_Callback(hObject, eventdata, handles)
global teleop_topic_name
teleop_topic_name = get(hObject,'String')
```

Similar to `ros_master_port` and `port`, this too is stored as string in a global variable.

After obtaining all these values, we can press the Connect to Robot button for connecting to the ROS robot/ROS PC. If the connection is successful, you can see proper messages in the command line. Here are the callback definitions of the Connect to Robot button:

```
function pushbutton6_Callback(hObject, eventdata, handles)
global ros_master_ip
global ros_master_port
global teleop_topic_name
global robot
global velmsg

ros_master_uri =
strcat('http://',ros_master_ip,':',ros_master_port)
setenv('ROS_MASTER_URI',ros_master_uri)

rosinit
```

```
robot = rospublisher(teleop_topic_name, 'geometry_msgs/Twist');
velmsg = rosmessage(robot);
```

This callback will set the `ROS_MASTER_URI` variable by concatenating `ros_master_ip` and the port. Then, it initialize the connection by calling `rosinit`. After connecting, it will create a publisher of `geometry_msgs/Twist`, which is for sending the command velocity. The topic name is the name that we give in the edit box.

After successful connection, we can control the robot by pressing keys such as Forward, Backward, Left, and Right.

The speeds of linear and angular velocity are initialized as follows:

```
global left_spinVelocity
global right_spinVelocity

global forwardVelocity
global backwardVelocity

left_spinVelocity = 2;
right_spinVelocity = -2;
forwardVelocity = 3;
backwardVelocity = -3;
```

Let's look at the function definition of Forward first:

```
function pushbutton4_Callback(hObject, eventdata, handles)
global velmsg
global robot
global teleop_topic_name
global forwardVelocity
velmsg.Angular.Z = 0;
velmsg.Linear.X = forwardVelocity;

send(robot,velmsg);
latchpub = rospublisher(teleop_topic_name, 'IsLatching', true);
```

What it basically does is it publishes a linear velocity and latches it on the topic. In the `backward` callback, we are providing a negative linear velocity. In the `left` and `right` callbacks, we are only providing an angular velocity.

After doing all this, we can save the figure file, which is the `.fig` and `.m` file, which is the MATLAB file.

Running the application

You can load your own application or the application that came along with the section. Here's how to run the application:

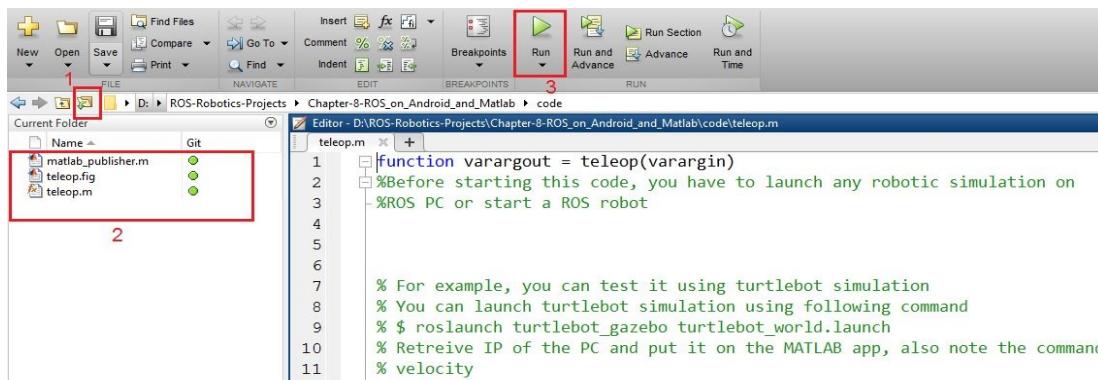


Figure 19: Running MATLAB application

First, you have to click on the Browse button, marked 1, to go to the application folder. If you are in the application folder, you can see the application files listed in the folder marked 2. After obtaining the files, double-click on the application file, which will pop up in the editor, and click on the Run button, marked 3.

Now, you will get the GUI and can fill the input arguments. After filling it all in, press the Enter key; only then it will give the value to the main code. You can fill the form like shown in the following screenshot. You can see the main GUI entries here.



Figure 20: Running the MATLAB application

Before connecting to the ROS robot, confirm whether robot or robot simulation is running on the ROS PC. For doing a test, you can start a TurtleBot simulation on the ROS PC using the following command:

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

The teleop topic of TurtleBot is `/cmd_vel_mux/input/teleop`, which we have already provided in the application.

After starting the simulation, you can connect to the MATLAB application by pressing the Connect to Robot button. If the connection is successful, you can see that the robot is moving when you press the corresponding buttons, as shown here:

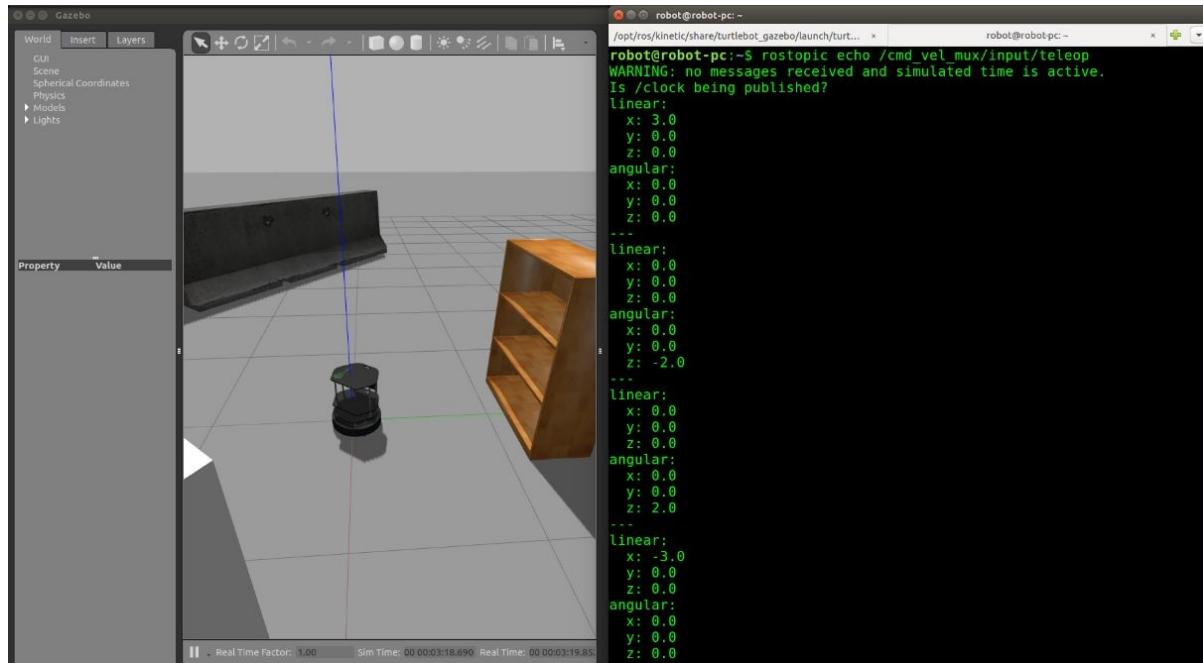


Figure 21: Controlling a robot in a ROS network

You can echo the command velocity topic using the following command:

```
| $ rostopic echo /cmd_vel_mux/input/teleop
```

After working with the robot, you can press the Disconnect button to disconnect from the ROS network.

You can clone the book code using the following command
\$ git clone https://github.com/qboticslabs/ros_robotics_projects

Getting started with Android and its ROS interface

There exists a cool interface between ROS and Android. As you know, Android is one of most popular operating systems in mobile devices. Just imagine: if we can access all features of a mobile devices on the ROS network, we can build robots using it, right? We can build Android apps with ROS capabilities and can make any kind of robot using it, its scope is unlimited.

The following shows how the communication between android device and ROS robot is happening. The figure shows an example Android-ROS application which can teleoperate robot from an android device. Each android application should inherit from **RosActivity** which is getting from Android-ROS interface, then only we can access ROS API's in our application. We can see more about the API's after this section.

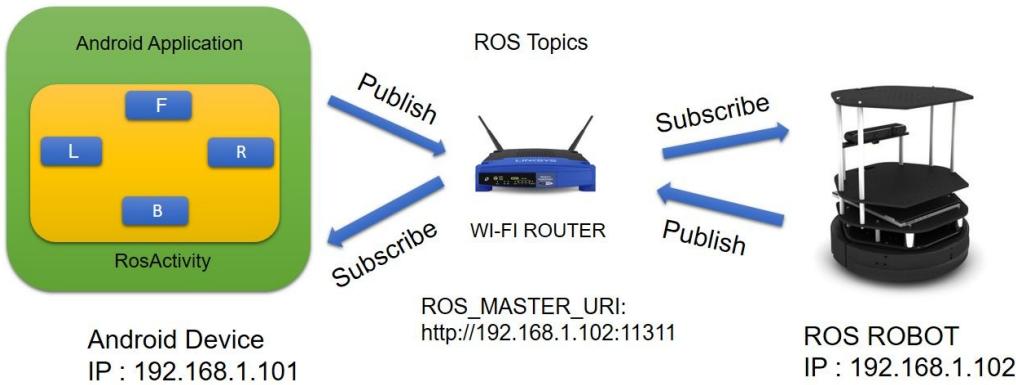


Figure 22: Android - ROS teleop interface

The core backend of the Android ROS library is RosJava (<http://wiki.ros.org/rosjava>), which is an implementation of ROS in Java. There is also Android core libraries (http://wiki.ros.org/android_core) which are built using RosJava API's. Using Android-ROS APIs we can create ROS nodes and ROS master, but compared to actual ROS API's in C++/Python, the Android-ROS features are less.

So what is the importance of Android-ROS interface? The main reason is, an Android device is like a mini computer, which is having all the sensors and other peripherals. We may can use an android device itself as a robot too. So if there is a ROS interface, we may can expand its capabilities by performing high level functionalities like navigation, mapping and localization. Nowadays, android devices are having high quality cameras, so we can even do image processing application via ROS interface.

Smartphone based robots are already in the market, we can also expand existing robots features by using ROS interface.

In the next section, we are going to see how to set up the ROS-Android interface on your PC and generate the **Android Package Kit (APK)** file, which can be directly installed on Android devices. Let's start setting up the ROS-Android interface on Ubuntu.

Before setting up Android, there is a long list of prerequisites that have to be satisfied for compiling and building it. Let's see what they are.

Installing rosjava

As you know, Android is based on Java, so it need `rosjava` to work.

Here are the methods to install `rosjava`.

Installing from the Ubuntu package manager

Here is the command to install `rosjava` from the package manager:

```
| $ sudo apt-get install ros-<rosversion_name>-rosjava
```

For example, for Kinetic, use the following command:

```
| $ sudo apt-get install ros-kinetic-rosjava
```

For Indigo, you can use the following command:

```
| $ sudo apt-get install ros-indigo-rosjava
```

Installing from source code

Here is the procedure to install `rosjava` from source code:

First, you have to create a catkin workspace folder called `rosjava`:

```
| $ mkdir -p ~/rosjava/src
```

Initialize the workspace and clone the source files using the following command:

```
| $ wstool init -j4 ~/rosjava/src https://raw.githubusercontent.com/rosjava/rosjava/indigo/rosjava.rosinstall
```

If you getting any issues related to `wstool`, you can install it using the following command:

```
| $ sudo apt-get install python-wstool
```

Switch to the `rosjava` workspace:

```
| $ cd ~/rosjava
```

Install the dependencies of the `rosjava` source files:

```
| $ rosdep update  
| $ rosdep install --from-paths src -i -y
```

After installing the dependencies, build the entire workspace:

```
| $ catkin_make
```

After successfully building the repository, you can execute the following command to add the `rosjava` environment inside bash:

```
| $ echo 'source ~/rosjava/devel/setup.bash' >> ~/.bashrc
```



For more reference, you can check the following link: <http://wiki.ros.org/rosjava/Tutorials/kinetic/Installation>

The next step is to set up `android-sdk` in Ubuntu. We can do it in two ways. One is through the Ubuntu package manager and the other is from the prebuilt binaries we can get from the Android website.

Let's see how to install the `android-sdk` using command line

Installing android-sdk from the Ubuntu package manager

Installing `android-sdk` using a command is pretty simple. Here is the command to install it. It may not be the latest version.

```
| $ sudo apt-get install android-sdk
```

Install the latest version of `android-sdk` available on the Android website. For building `android-ros` applications, you only need to install `android-sdk`; an IDE is not mandatory.

Installing android-sdk from prebuilt binaries

Here is the link for downloading latest `android-sdk` version from the website:

<https://developer.android.com/studio/index.html#downloads>

You only need to download the Android tools for building `android-ros` apps:

Get just the command line tools

If you do not need Android Studio, you can download the basic Android command line tools below. You can use the included `sdkmanager` to download other SDK packages.

These tools are included in Android Studio.

| Platform | SDK tools package | Size | SHA-1 checksum |
|----------|---|-------------------------------|--|
| Windows | tools_r25.2.3-windows.zip | 292 MB (306,745,639 bytes) | b965decb234ed793eb9574bad8791c50ca574173 |
| Mac | tools_r25.2.3-macosx.zip | 191 MB (200,496,727 bytes) | 0e88c0bdb8f8ee85cce248580173e033a1bbc9cb |
| Linux | tools_r25.2.3-linux.zip | 264 MB (277,861,433 bytes) | aafe7f28ac51549784efc2f3bdfc620be8a08213 |

See the [SDK tools release notes](#).

Figure 23: Standalone Android SDK

You can download and extract this into the `home` folder, and you have to set up environment variables to access the SDK tools.

Let's look at the variables you have to append on your `.bashrc` file. You can set them using the following commands.

Here is how we set the `ANDROID_HOME` variable, which is required while building `android-ros` applications. You can set your own SDK location here:

```
| $ export ANDROID_HOME=~/android-sdk-linux
```

This command will help you access Android commands from bash:

```
| $ export PATH=${PATH}:~/android-sdk-linux/tools  
| $ export PATH=${PATH}:~/android-sdk-linux/platform-tools
```

To run those Android commands, we also need to install the 32-bit Ubuntu libraries. It is required for running most Android tool commands.

We can install it using the following command:

```
| $ sudo dpkg --add-architecture i386  
| $ sudo apt-get update  
| $ sudo apt-get install libc6:i386 libncurses5:i386 libstdc++6:i386 lib32z1 libbz2-1.0:i386
```

You can also refer to the following instructions to set `android-sdk` in Linux:

<https://developer.android.com/studio/install.html?pkg=tools>

Congratulations, you are almost there!

Now, you can run the following command to start Android SDK manager:

```
| $ android
```

From the pop-up window, you have to install the following Android platforms and their build tools to make the ROS-Android interface work:

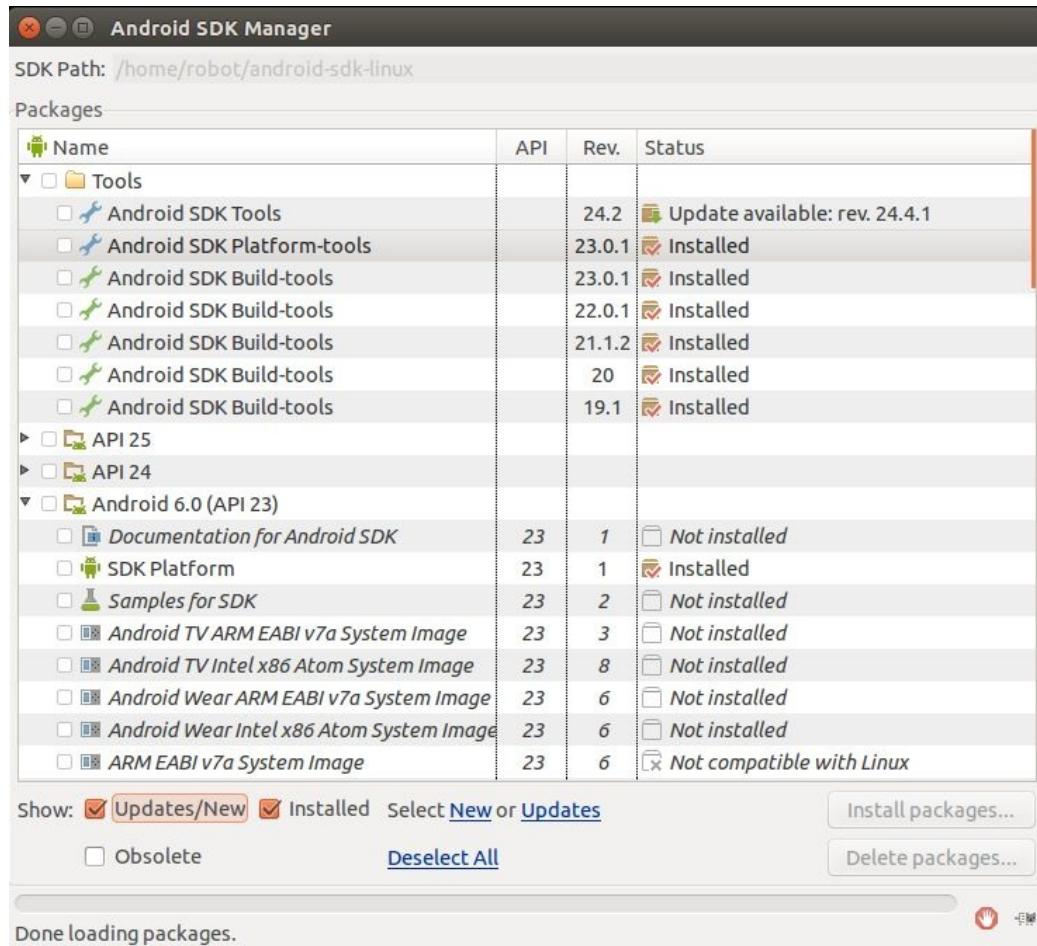


Figure 24: Android SDK manager

Here is the list of things you may need to install in Android SDK manager:

- SDK Platforms
 - Android 2.3.3 (API 10)
 - Android 4.0.3 (API 15)
 - Android 5.0.1 (API 21)
 - Android 6.0 (API 23)
- Android SDK Build-tools
 - Revision: 19.1
 - Revision: 20
 - Revision: 21.1.2
 - Revision: 22.0.1
 - Revision: 23.0.1
- Android SDK Platform-tools
 - Revision: 23.0.1

- Android SDK Tools
 - Revision: 24.2

Your configuration may vary; this was the configuration used to build the apps for this section.

With this, we should have met all the dependencies for the Android-ROS interface.

Let's clone the Android-ROS source code.

Installing the ROS-Android interface

If all the dependencies are satisfied, you can easily build the ROS-Android interface and build a bunch of Android-ROS applications. Here is how we can do that:

Initially, we have to create a workspace folder for the Android interface. We can name it `android_core`:

```
| $ mkdir -p ~/android_core
```

After creating this folder, you can initialize the workspace using the following command:

```
| $ wstool init -j4 ~/android_core/src https://raw.github.com/rosjava/rosjava/indigo/android_core.rosinstall
```

Now switch to the workspace and build the workspace using `catkin_make`:

```
| $ cd ~/android_core  
| $ catkin_make
```

After building the workspace successfully, you can source it by adding it to `.bashrc`:

```
| $ echo 'source ~/android_core /devel/setup.bash' >> ~/.bashrc
```

You are now done with setting up the `android_core` package in ROS. So what do you get after building this workspace? You will get a bunch of Android-ROS applications that can be installed on your Android device. You will also get the Android-ROS library, which we can use in our custom application

 *For more reference, you can check following link: <http://wiki.ros.org/android/Tutorials/kinetic>.*

Playing with ROS-Android applications

In this section, we will see how to install the ROS-Android application generated from the preceding build process on your Android phone.

Let's take the `android_core` folder and search for `.apk` files; you may get a bunch of applications, as shown in the following figure:

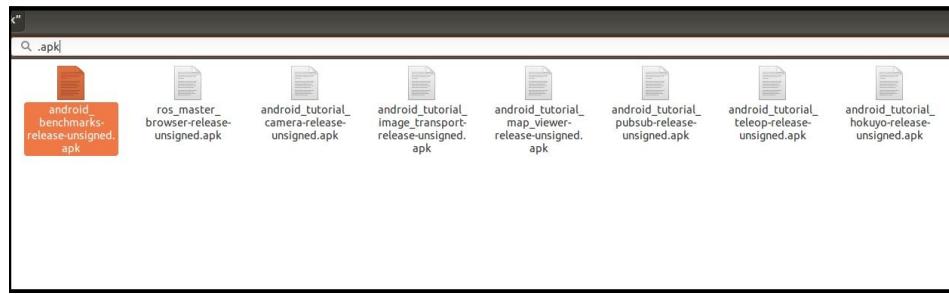


Figure 25: List of generated APK files

You can copy the APK files and install them on your phone.

Troubleshooting

You may get errors while installing these APK files. One of the errors is shown in the following screenshot:

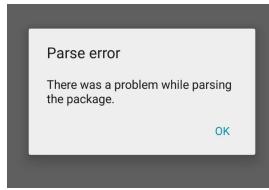


Figure 26: Parse error during installation of APK

Here are the tips to solve this issue:

The first step is to enable installation from Unknown sources, as shown in section 1 of the following figure:

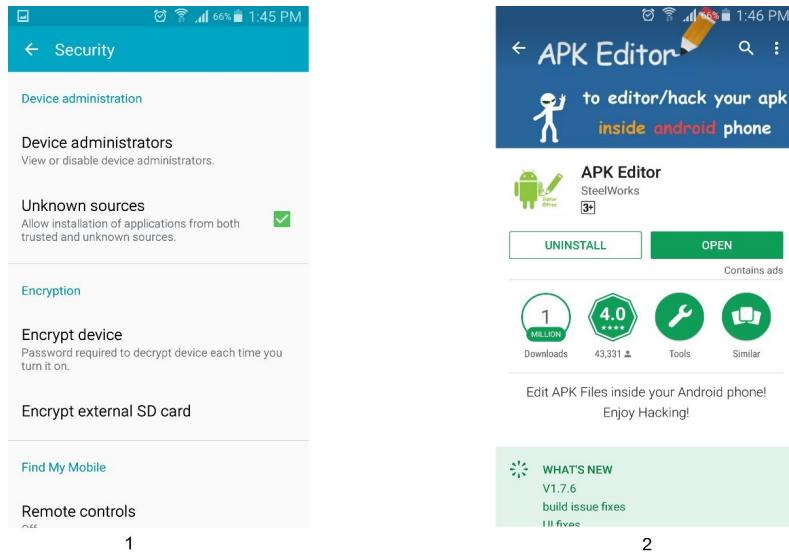


Figure 27: Tips to solve parse error

Install an Android app called **APK Editor**, which can be downloaded from following the link:

<https://play.google.com/store/apps/details?id=com.gmail.heagoo.apkeditor&hl=en>

You can also buy the Pro version, which you may require in the future. Here's the link to the Pro version:

<https://play.google.com/store/apps/details?id=com.gmail.heagoo.apkeditor.pro>

What this app does is enable us to edit the APK that we created and do more stuff with it. For example, the APK that we built was unsigned; using this app, we can sign it. We can also change the minimum and target SDK using the app.

Here is how we can edit the APK and install our APKs:

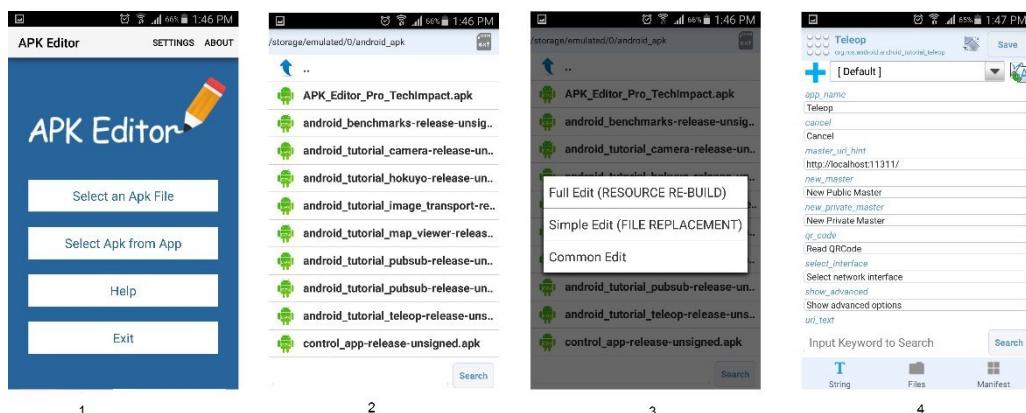


Figure 28: Working with APK Editor

What we need to do with this app is simple. Just choose an APK from this app, click on the Full Edit option, and save it. After saving, you can see a wizard that shows an option for installing our app:

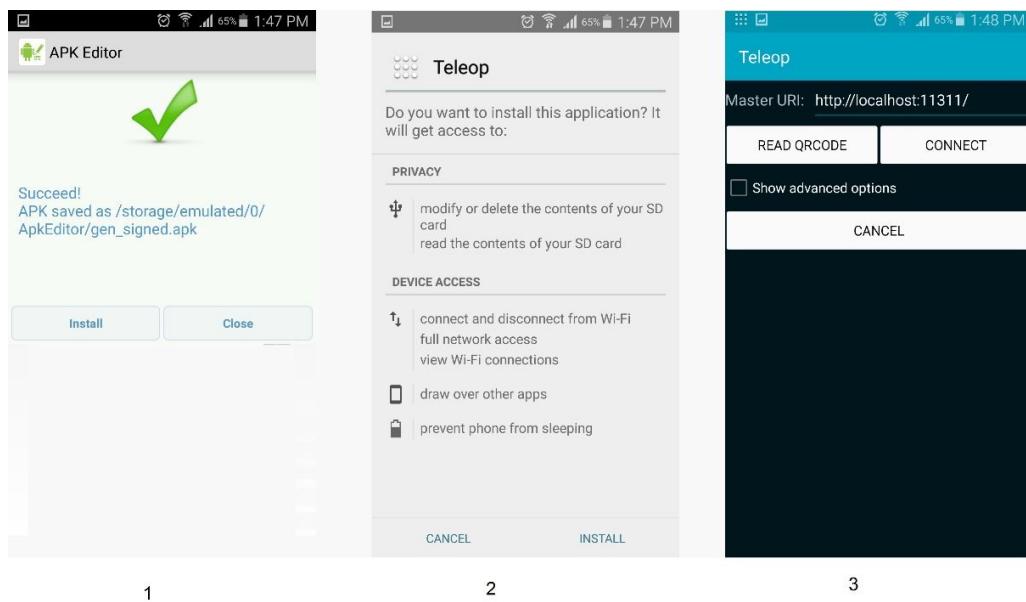


Figure 29: Installing the ROS-Android app

Once you've installed the application successfully, we can work with the ROS-Android examples.

Android-ROS publisher-subscriber application

You can first find the publisher-subscriber application with the name `androidTutorial_pubsub-release.apk`. Install it using the preceding procedure, and let's learn how we can work with it.

You can open the PubSub Tutorial application, and you'll see the following window marked 1:

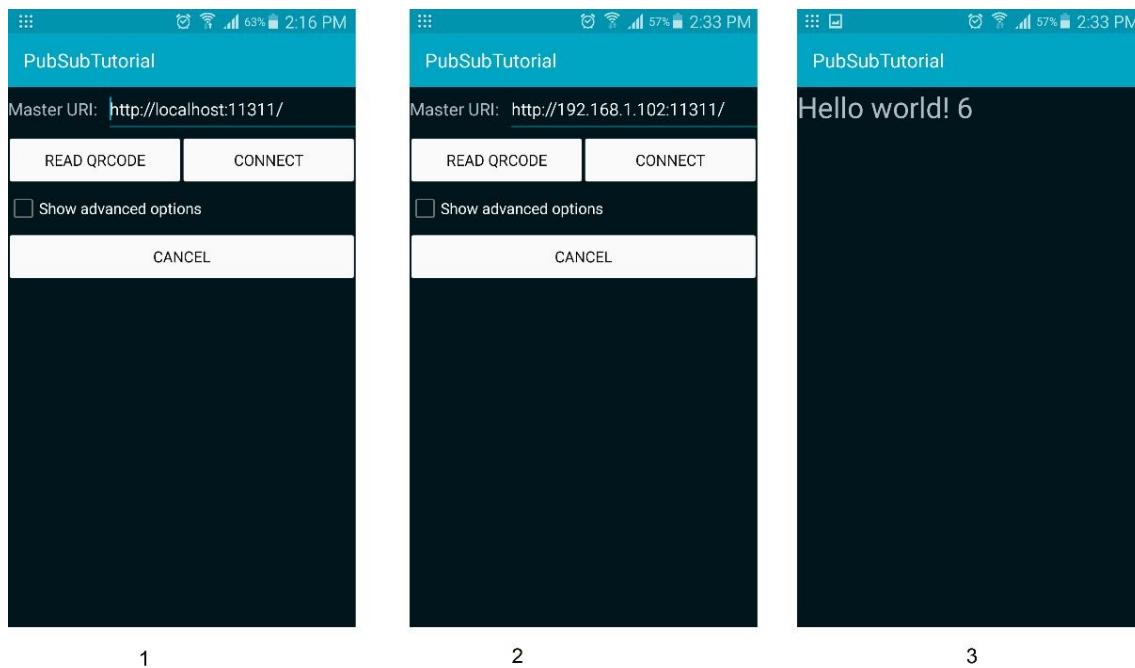


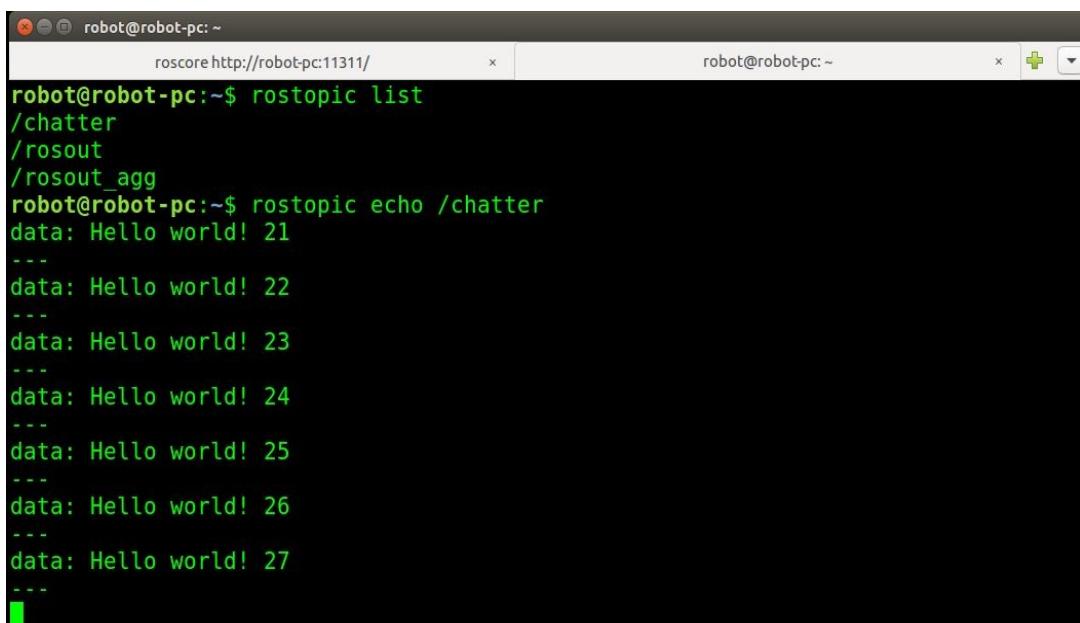
Figure 30: PubSub Tutorial ROS-Android app

Assuming you have connected your Android device and ROS PC over Wi-Fi and in the same network, launch `roscore` on your ROS PC and note its IP too.

In the first window of the app, you have to provide the `ROS_MASTER_URI`. In that, you can replace the '`localhost`' variable with your ROS PC IP address; here, it is `192.168.1.102`.

When you press the Connect button, the app tries to reach the ROS master, which is running on the ROS PC, and if it is successful, it will start publishing the `Hello World` message to a topic called `/chatter`.

Now you can check the ROS PC and list the topics; you'll see the `/chatter` topic, and you can also echo the topic, as shown in the following screenshot:



The screenshot shows a terminal window with two tabs. The left tab is titled 'roscore http://robot-pc:11311/' and the right tab is titled 'robot@robot-pc: ~'. The terminal content is as follows:

```
robot@robot-pc:~$ rostopic list
/chatter
/rosout
/rosout_agg
robot@robot-pc:~$ rostopic echo /chatter
data: Hello world! 21
---
data: Hello world! 22
---
data: Hello world! 23
---
data: Hello world! 24
---
data: Hello world! 25
---
data: Hello world! 26
---
data: Hello world! 27
---
```

Figure 31: Echoing the /chatter topic on ROS PC

The teleop application

One of the commonly used apps in this list is the Android teleop application. Using this app, you can control the ROS robot from your Android phone.

Like the previous app, the setup is the same, and using a virtual joystick in this app, we can control the movement and rotation of the robot. Here are the screenshots of the app:

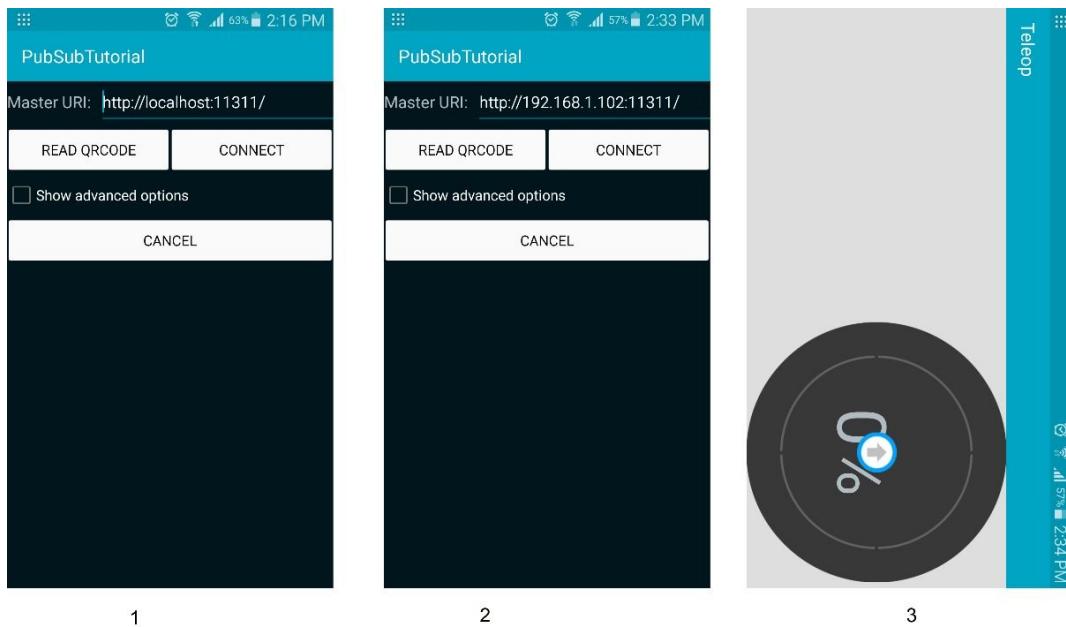
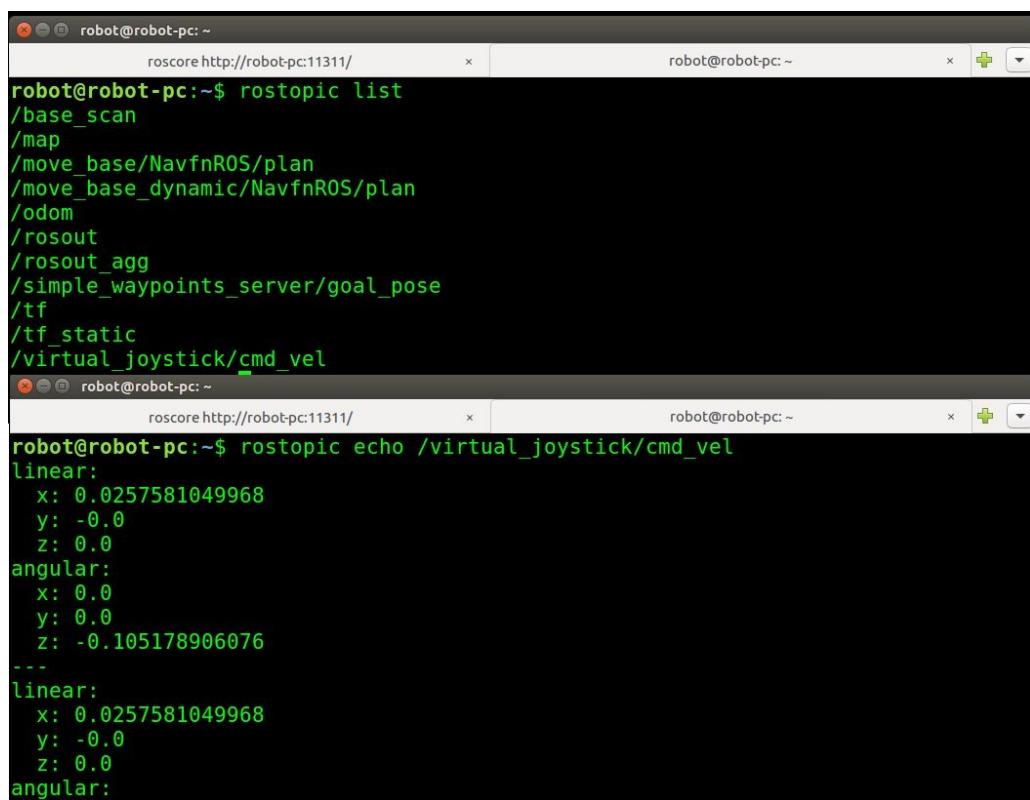


Figure 32: Android teleop application

Here are the topics and output that we may get on the ROS PC or robot. You can see a bunch of topics, actually, which are useful for robot navigation. Now we only need the command velocity topic:



The image shows two terminal windows side-by-side. The left window has the title 'roscore http://robot-pc:11311/' and displays the command 'rostopic list' followed by a list of ROS topics. The right window has the title 'robot@robot-pc: ~' and displays the command 'rostopic echo /virtual_joystick/cmd_vel' followed by the message content.

```
robot@robot-pc:~$ rostopic list
/base_scan
/map
/move_base/NavfnROS/plan
/move_base_dynamic/NavfnROS/plan
/odom
/rosout
/rosout_agg
/simple_waypoints_server/goal_pose
/tf
/tf_static
/virtual_joystick/cmd_vel
robot@robot-pc:~$ rostopic echo /virtual_joystick/cmd_vel
linear:
  x: 0.0257581049968
  y: -0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: -0.105178906076
---
linear:
  x: 0.0257581049968
  y: -0.0
  z: 0.0
angular:
```

Figure 33: Android teleop application data

The ROS Android camera application

The final application that we are going to check out is the Android-ROS camera application. This application will stream Android phone camera images over ROS topics. You can install the Camera Tutorial app on your Android device and try to connect to the ROS master. If the connection is successful, you will see the camera view open up on the mobile device.

Now, check the ROS PC, and you can visualize the camera topic from the phone. Here is the command to perform the visualization:

```
| $ rosrun image_view image_view image:=/camera/image _image_transport:=compressed
```

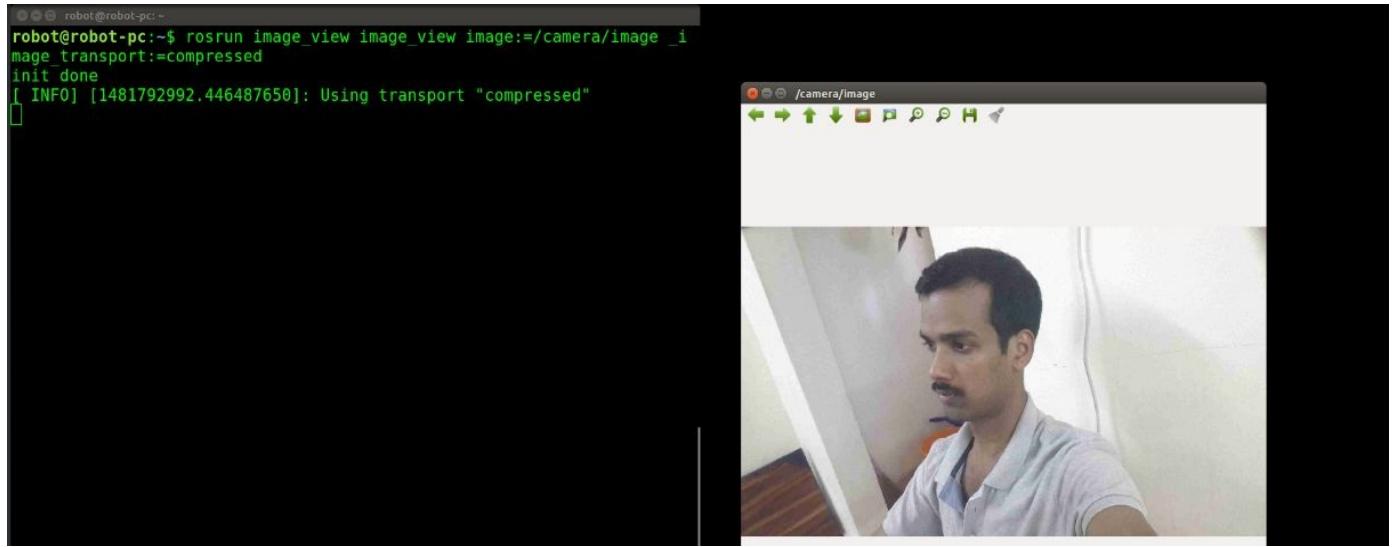


Figure 34: Android-ROS camera app

Making the Android device the ROS master

In the previous test, we made the Android ROS application a ROS node; we can also configure it as a ROS master. Show the advanced option from the app and click on PUBLIC MASTER. Now the app itself act as the ROS master. You can connect from your PC to the Android device as a node.

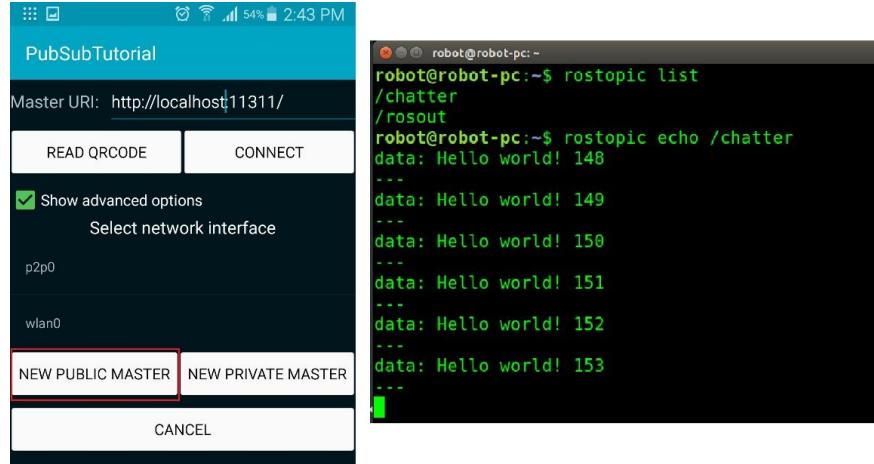


Figure 35: Android-ROS app as ROS MASTER

For listing topics on the ROS PC, you have to set `ROS_MASTER_URI` inside the `.bashrc` file:

Here, I have defined `ROS_MASTER_URI` like this:

```
| $ export ROS_MASTER_URI=http://192.168.1.100:11311
```

The IP is the IP address of the Android device.

Code walkthrough

Let's check out the Android-ROS application code for the basic publisher-subscriber app. You can get it from `~/android_core/android_tutorial_pubsub/src`. You'll see a file called `MainActiviy.java`, and now I'll explain the code.

In the beginning of the code, you can see the package name and required Android modules for this application. The important modules are `RosActivity` and `NodeConfiguration`. These will help us create a new ROS node in an Android activity (<https://developer.android.com/guide/components/activities.html>).

```
package org.ros.android.android_tutorial_pubsub;

import android.os.Bundle;
import org.ros.android.MessageCallable;
import org.ros.android.RosActivity;
import org.ros.android.view.RosTextView;
import org.ros.node.NodeConfiguration;
import org.ros.node.NodeMainExecutor;
import org.ros.rosjava_tutorial_pubsub.Talker;
```

Here is where the Android `MainActivity` starts, which is inherited from `RosActivity`. It is also creates a `Talker` object for publishing topics.

```
public class MainActivity extends RosActivity {

    private RosTextView<std_msgs.String> rostextView;
    private Talker talker;

    public MainActivity() {
        // The RosActivity constructor configures the notification
        // title and ticker
        // messages.
        super("Pubsub Tutorial", "Pubsub Tutorial");
    }
}
```

This is one of the important callback functions whenever an activity is initialized. We have to define the essential components of activities inside this function.

In this code, we are creating a `rostextView` for `ROS_MASTER_URI` and also performing topic creation and creating a callback for sending the ROS message through the topic:

```
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
    rostextView = (RosTextView<std_msgs.String>)
        findViewById(R.id.text);
    rostextView.setTopicName("chatter");
    rostextView.setMessageType(std_msgs.String._TYPE);
    rostextView.setMessageToStringCallable(new
        MessageCallable<String, std_msgs.String>() {
            @Override
            public String call(std_msgs.String message) {
                return message.getData();
            }
        });
}
```

The following function is inherited from `RosActivity`, and what it does is when the `MainActivity` gets initialized, it will run as a thread and query for `ROS_MASTER_URI`. If it gets the URI, it will start a ROS node

itself.

```
protected void init(NodeMainExecutor nodeMainExecutor) {  
    talker = new Talker();  
  
    // At this point, the user has already been prompted to either  
    // enter the URI  
    // of a master to use or to start a master locally.  
  
    // The user can easily use the selected ROS Hostname in the  
    // master chooser  
    // activity.  
    NodeConfiguration nodeConfiguration =  
    NodeConfiguration.newPublic(getRosHostname());  
    nodeConfiguration.setMasterUri(getMasterUri());  
    nodeMainExecutor.execute(talker, nodeConfiguration);  
    // The RosTextView is also a NodeMain that must be executed in  
    // order to  
    // start displaying incoming messages.  
    nodeMainExecutor.execute(rosTextView, nodeConfiguration);  
}  
}
```

You can see more code of ROS-Android applications from the `android_core` package.

Creating basic applications using the ROS-Android interface

We have covered Android - ROS applications provided from the ROS repository. So how can we create our own application using it? Let's take a look.

First, we have to create a separate workspace for our application. Here, it is named `myandroid`:

```
| $ mkdir -p ~/myandroid/src
```

Switch to the workspace's `src` folder:

```
| $ cd ~/myandroid/src
```

Create a package called `android_foo` that depends on `android_core`, `rosjava_core`, and `std_msgs`:

```
| $ catkin_create_android_pkg android_foo android_core rosjava_core std_msgs
```

Switch into `android_foo` and add sample libraries to check whether the project is building properly:

```
$ cd android_foo  
$ catkin_create_android_project -t 10 -p com.github.ros_java.android_foo.bar bar  
$ catkin_create_android_library_project -t 13 -p com.github.ros_java.android_foo.barlib barlib  
$ cd ../../
```

And finally, you can build the empty project using `catkin_make`:

```
| $ catkin_make
```

If it is building properly, you can add a custom project, such as a `bar` project. The custom project should be inside the `android_foo` folder, and it should be included in the `settings.gradle` file, which is in the `android_foo` folder.

Here is how we can do that. You need to include our app, named `my_ros_app`, in this file to build it. For the application source code, you can modify one of the existing ROS-Android applications' code and write new lines:

```
include 'my_ros_app'  
include 'bar'  
include 'barlib'
```

Also, inside `my_ros_app`, you should include the ROS Android dependencies in the `build.gradle` file; otherwise, the package will not build properly. Here is a sample `build.gradle` file. You can also mention the minimum SDK, target SDK, and compiled SDK versions in the same file.

```
dependencies {  
    compile 'org.ros.android_core:android_10:[0.2,0.3)'  
    compile 'org.ros.android_core:android_15:[0.2,0.3)'  
}
```

```
apply plugin: 'com.android.application'

android {
    compileSdkVersion 15

    defaultConfig {
        minSdkVersion 15
        applicationId "org.ros.android.my_ros_app"
        targetSdkVersion 21
        versionCode 1
        versionName "1.0"
    }
}
```

If you've entered all this information correctly, you can build your own custom ROS-Android application.



For more reference, you can check the following link: http://wiki.ros.org/rosjava_build_tools/Tutorials/indigo/Creating%20Android%20Packages

Troubleshooting tips

There are chances of getting errors while building packages. Errors are mainly because of missing Android platform or build tools. If any platforms are missing, you can install them through the Android SDK manager.

Questions

- What are the main features of MATLAB Robotics Toolbox?
- How to set up MATLAB as a ROS master?
- What is the main backend of the ROS-Android interface?
- How to set up a ROS-Android application as a ROS master?

Summary

In this chapter, we mainly discussed two important interfaces of ROS: MATLAB and Android. Both are very popular platforms, and this chapter will be very useful if you are working on the interfacing of ROS with MATLAB and Android. In MATLAB interfacing, we covered Robotics Toolbox and APIs to connect to ROS networks. Using these APIs, we built a GUI application to teleoperate a ROS robot. In the Android-ROS interfacing section, we saw how to set up and build Android-ROS applications from a Linux PC. After that, we successfully built ROS-Android applications and saw demos of important applications. We also saw the Android-ROS application code and its functions, and finally, we saw how to build a custom Android-ROS application.

Building an Autonomous Mobile Robot

An autonomous mobile robot can move from its current position to the goal position autonomously with the help of mapping and localizing algorithms. ROS provides some powerful packages to prototype an autonomous mobile robot from scratch. Some of the packages used in an autonomous robot are the ROS navigation stack, `gmapping`, and `amcl`. Combining these packages, we can build our own autonomous mobile robot. In this chapter, we will see a DIY autonomous mobile robot platform that works using ROS. This project is actually the updated version of the work mentioned in my first book, *Learning Robotics Using Python*, Packt Publishing (<http://learn-robotics.com>). In this chapter, we will mainly go through designing and building the simulation of a robot, then the hardware of robot, and finally the software framework. The chapter will be an abstract of all these things, since explaining everything in a single chapter will be a tedious task.

The following are the main topics we will discuss on this chapter:

- Robot specification and design overview
- Designing and selecting motors and wheels for the robot
- Building a 2D and 3D model of the robot body
- Simulating the robot model in Gazebo
- Designing and building actual robot hardware
- Interfacing robot hardware with ROS
- Setting up the ROS navigation stack and `gmapping` packages
- Final run

Robot specification and design overview

Here are the main specifications of the robot we are going to design in this chapter:

- A maximum payload of 2 kg
- Body weight of 3 kg
- A maximum speed of 0.35 m/s
- Ground clearance of 3 cm
- Two hours of continuous operation
- Differential drive configuration
- Circular base footprint
- Autonomous navigation and obstacle avoidance
- Low-cost platform

We are going to design a robot that satisfies all these specifications.

Designing and selecting the motors and wheels for the robot

The robot we are going to design should have a differential drive configuration, and from the preceding specification, we can first determine the motor torque values. From the payload value and robot body weight, we can easily compute the motor torque.

Computing motor torque

Let's calculate the torque required to move this robot.

The number of wheels is four, including two caster wheels. The number of wheels undergoing actuation is only two. We can assume the coefficient of friction is 0.6 and of wheel radius is 4.5 cm. We can use the following formula:

$$\text{Total weight of robot} = \text{Weight of robot} + \text{Payload}$$

Weight of the robot: $3 \times 9.8 \approx 30 \text{ N}$ ($W = mg$)

Payload: $2 \times 9.8 \approx 20 \text{ N}$

Total weight: $30 + 20 = 50 \text{ N}$

This total weight should be split among the four wheels of the robot, so we can write it as $W = 2 \times N_1 + 2 \times N_2$, where N_1 is the weight acting on each robot wheel and N_2 is the weight acting on each caster wheel. The configuration of wheels of the robot is shown in *Figure 1*. The **C1** and **C2** shows the caster wheels of the robot and **M1** and **M2** shows the motor position in which wheels can attach on the slots just near to the motor shaft.

If the robot is stationary, the motors attached to the wheels have to exert maximum torque to get moving. This is the maximum torque equation:

$$\mu \times N \times r - T = 0$$

Here, μ is the coefficient of friction, N is the average weight acting on each wheel, r is the radius of the wheels, and T is the maximum torque to get moving.

We can write $N = W/2$ since the weight of the robot is equally distributed among all four wheels, but two are only actuated. We are taking $W/2$ as the average weight here.

$$\text{We can write } 0.6 \times (50/2) \times 0.045 - T = 0$$

Hence, $T = 0.675 \text{ N-m}$ or 6.88 kg-cm . We can use a standard value, 10 kg-cm .

Calculation of motor RPM

From the specification, we get to know that the maximum speed of the robot is 0.35 m/s. We took the wheel radius as 4.5 cm in the preceding section, and one of the other specifications we need to satisfy is ground clearance. The specified ground clearance is 3 cm, so this wheel is satisfying those requirements too. We can find the **rotations per minute (RPM)** of the motors using the following equation:

$$RPM = ((60 \times Speed) / (3.14 \times Diameter\ of\ wheel))$$

$$RPM = (60 \times 0.35) / (3.14 \times 0.09) = 21 / 0.2826 = 74\ RPM$$

We can choose a standard *80 RPM* or *100 RPM* for this robot.

Design summary

After designing, we have the following design values:

- Motor RPM: 80
- Motor torque: 10 kg-cm
- Wheel diameter: 9 cm

Building 2D and 3D models of the robot body

Chassis design is the next step in designing the robot. We can create the 2D drawing of the robot and then draw a 3D model of it. The only specification need to satisfy is that the robot's base footprint should be circular. Here, we are discussing a drawing that is satisfying this condition. If your requirements are different, you may need to modify your design accordingly. Now let's look at some illustrations of the robot's footprint.

The base plate

Following figure shows the base footprint of our robot:

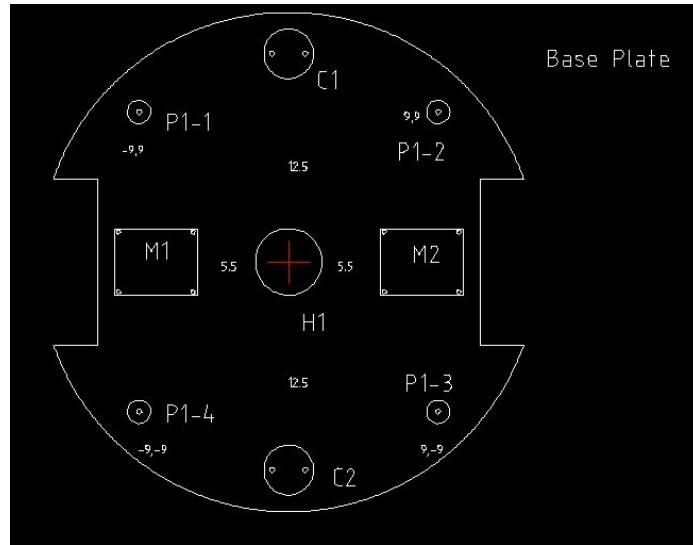


Figure 1: Base plate of the robot

The preceding figure shows the base footprint of our robot. You can see that it is circular and there are two slots on the left and right for attaching motors and wheels. **M1** and **M2** are the positions of the motor body, and the shaft will be in the slots. The motors can be put on the top of the plate or on the bottom. Here, we are attaching the motors to the bottom of this plate. The wheels should be inside these two slots. We have to make sure that the slot length is greater than the wheel diameter. You can see **C1** and **C2**, which are the positions where we are attaching the caster wheels. Caster wheels are freely rotating wheels without any actuation. We can select available caster wheels for this purpose. Some caster wheels may have issues moving on uneven terrain. In that case, we may need to use a caster wheel with spring suspension. This ensures that it always touches the ground even when the terrain is slightly uneven.

You can also see parts such as **P1-1** and **P1-4**, which are the poles from the base plate. If we want to attach an additional layer above the base plate, we can use these poles as the pillars. Poles can be hard plastic or steel, which are fixed to the base plate and have a provision to attach a hollow tube on them. Each pole is screwed on to the base plate.

The center of the base plate is hollow; this will be useful when we have to take wires from the motors. Mainly, we will attach the electronic board required for the robot to this plate.

Here are the dimensions of base plate and each part:

| Parts of base plate | Dimensions (length x height) or (radius) in cm |
|---------------------|--|
| M1 and M2 | 5 x 4 |
| C1 and C2 | Radius = 1.5 |
| S (screw) | 0.15 |

| | |
|------------------------------|------------------------------|
| P1-1, P1-2, P1-3, P1-4 | Outer radius 0.7, Height 3.5 |
| Left and right wheel section | 2.5 x 10 |
| Base plate | Radius = 15 |

The pole and tube design

The following figure shows how to make a pole and tube for this robot. Again, this design is all up to you. You can design customized poles too:

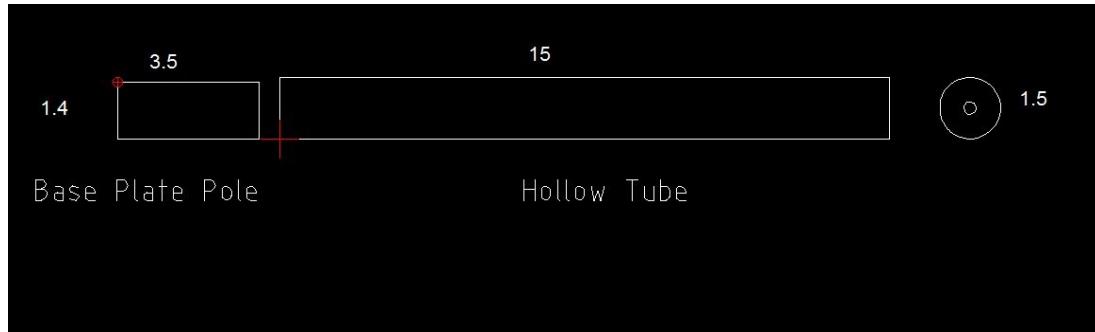


Figure 2: Pole and tube dimension of the robot

From the preceding figure, you can see the dimension of the pole and tube. It's **3.5 cm** by **1.4 cm**. The poles that we've used here are basically hard plastic. We are using hollow tubes to connect to the poles and extend them for the second layer. The length of the hollow tube is **15 cm**, and it has a slightly bigger diameter than the poles, that is, **1.5 cm**. Only then will we be able to insert this tube into the pole. A hard plastic piece is inserted at one side of the hollow tube, which helps connect the next layer.

The motor, wheel, and motor clamp design

You can choose a motor and wheels that satisfy the design criteria. Most of the standard motors come with clamps. The motor can be connected to the base plate using this clamp. If you don't have one, you may need to make it. This is the drawing of a standard clamp that goes with one of the motors:

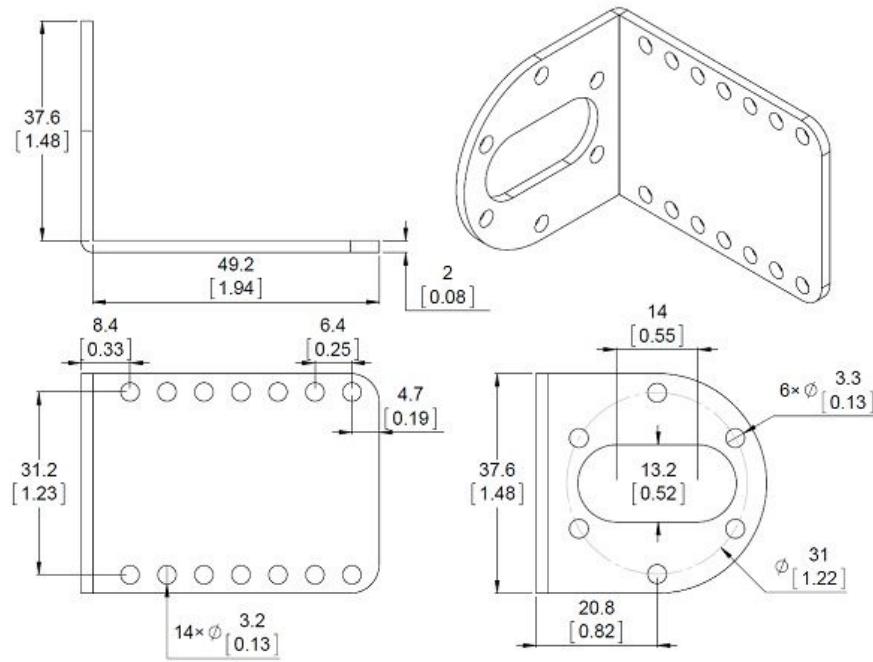


Figure 3: The clamp design

The clamp can be fixed on the base plate, and the motor shaft can be put through the clamp slot which is perpendicular to the clamp base.

The caster wheel design

You can use any caster wheel that can be move freely on the ground. The main use of caster wheels is distributing the weight of the robot and balancing it. If you can use a spring suspension on the caster wheel, it can help you navigate the robot on uneven terrain.

Here are some caster wheels that you can use for this robot:

<http://www.robotshop.com/en/robot-wheel.html>.

Middle plate and top plate design

If you want a more layers for the robot, you can simply make circular plates and hollow tubes which are compatible with the base plate. Here you can see middle plate design and the tubes used to connect it to the base plate:

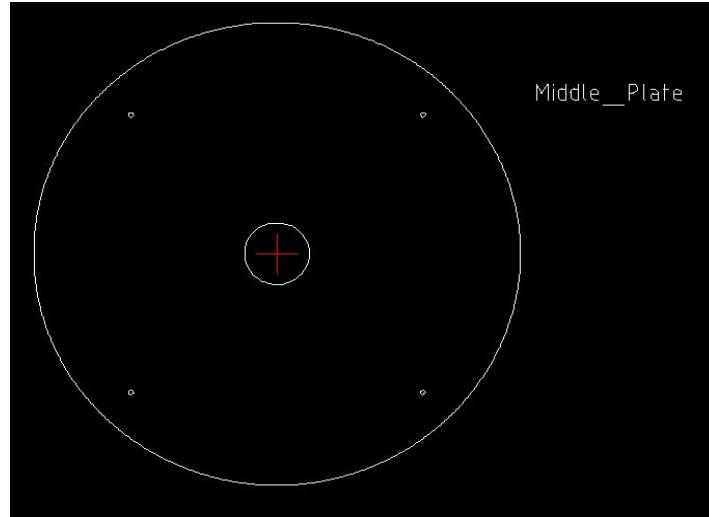


Figure 4: The middle plate design

The middle plate is simply a circular plate having screw holes to connect it to the tubes from the base plate. We can use following kind of hollow tubes to connect the base plate tubes and middle plate.

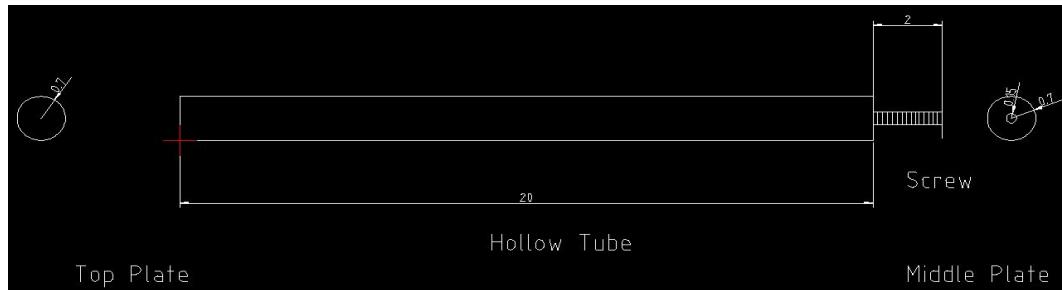


Figure 5: The hollow tube from the second plate

Here you can see that a screw is mounted on one side of the tube; the screw can be used to connect tubes to the base plate. We can mount the top plate on top of the tube too.

The top plate

Here is a diagram of the top plate:

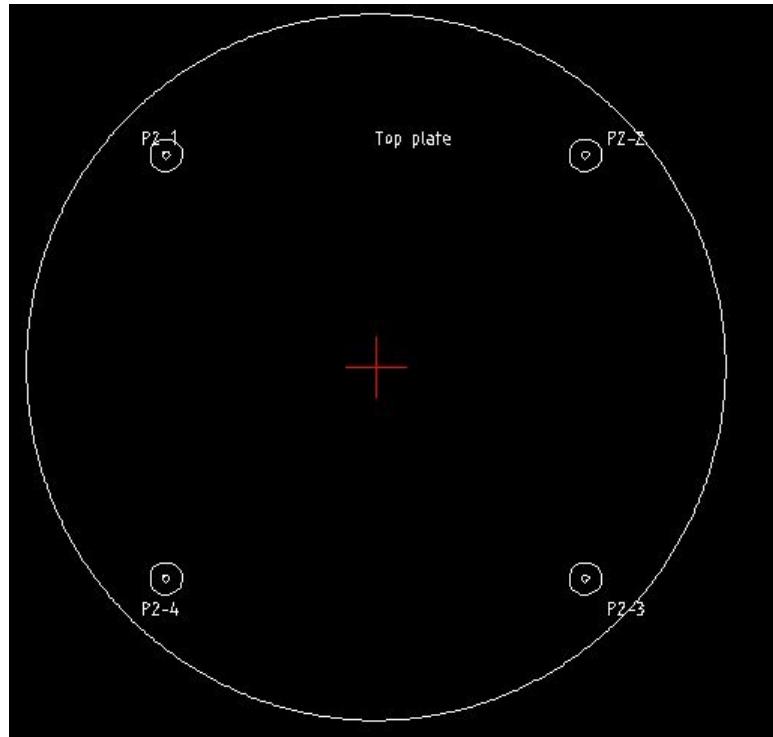


Figure 6: The top plate

The top plate can be placed in a hollow tube. If we want to put anything on top of the robot, we can put it on the top plate. On the middle plate, we can put vision sensors, PC, and so on for processing.

These are the main structural elements that we need for this robot. These drawing can be develop in any CAD software like AutoCAD and LibreCAD. AutoCAD is a proprietary software whereas LibreCAD is free (<http://librecad.org/cms/home.html>). We have used LibreCAD for developing the preceding sketches.

You can simply install LibreCAD in Ubuntu using the following command:

```
| $ sudo apt-get install librecad
```

In the next section, we can see how we can model the robot in 3D. The 3D modeling is mainly using for robot simulation.

3D modeling of the robot

The 3D modeling of the robot can be done in any 3D CAD software. You can use popular commercial software such as AutoCAD, SOLIDWORKS, and CATIA or free software such as Blender. The design can be customized according to your specification. Here, you can see a 3D model of the robot built using Blender. Using the 3D model, we can perfect the robot's design without building the actual hardware. We can also create the 3D simulation of the robot using this model. The following screenshot shows the 3D model of a robot designed using Blender:

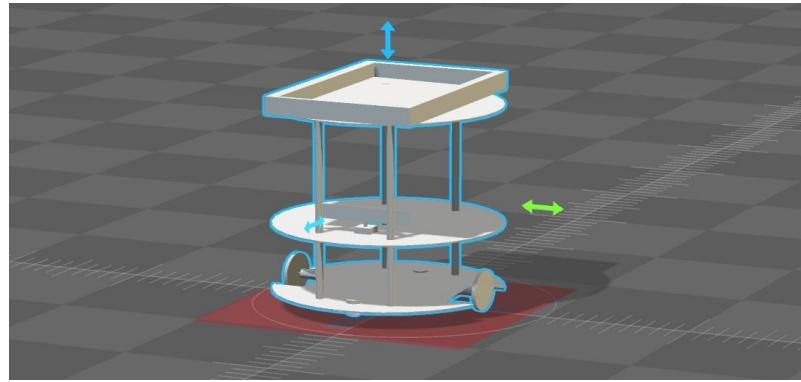


Figure 7: The 3D model

You can check out this model at [chapter_9_codes/chefbot](#).

Simulating the robot model in Gazebo

After modeling the robot, the next stage that we have to do is simulation. The simulation is mainly for mimicking the behavior of designed robot. For the simulation, normally we are putting ideal parameters to the simulated model. When we do the actual robot, there can be some changes from the simulated parameters. We can simulate the robot using Gazebo. Before simulating the robot, it will be good if you understand the mathematical model of a differential robot. The mathematical representation will give you more insight about the working of robot. We are not going to implement the robot controllers from scratch. Instead of that, we are using existing one.

Mathematical model of a differential drive robot

As you may know, robot kinematics is the study of motion without considering the forces that affect the motion, and robot dynamics is the study of the forces acting on a robot. In this section, we will discuss the kinematics of a differential robot.

Typically, a mobile robot or vehicle can have six **degrees of freedom (DOF)**, which are represented as x , y , z , roll, pitch, and yaw. The x , y , and z degrees are translation, and roll, pitch, and yaw are rotation values. The roll movement of robot is sideways rotation, pitch is forward and backward rotation, and yaw is the heading and orientation of the robot. A differential robot moves along a 2D plane, so we can say it will have only three DOF, such as x , y , and θ , where θ is the heading of the robot and points along the forward direction of the robot.

The following figure shows the coordinate system of a differential-drive robot:

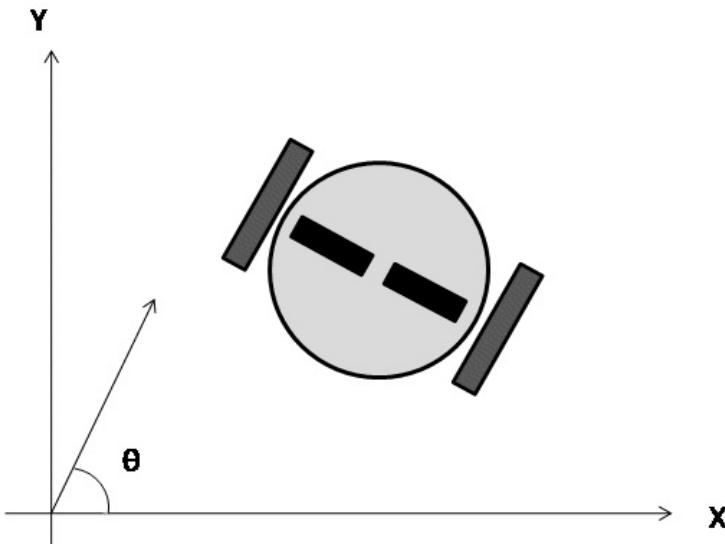


Figure 8: The coordinate system representation of a differential-drive robot

So how to control this robot? It has two wheels, right? So the velocity of each wheel determines the new position of the robot. Let's say V_{left} and V_{right} are the respective wheel velocities, (x, y, θ) is the standing position of the robot at time t , and (x', y', θ') is the new position at time $t + \delta t$, where δt is a small time interval. Then, we can write the standard forward kinematic model of a differential robot like this:

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

Figure 9: Forward kinematics model of a differential drive robot

Here are the unknown variables in the preceding equation:

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

$$ICC = [x - R \sin\theta, y + R \cos\theta]$$

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

nl and nr are encoder counts for left and right wheels. l is the length of the wheel axis and $step$ is the distance covered by the wheel in each encoder ticks.

ICC stands for **instantaneous center of curvature**, and it is the common point for rotation of the robot wheels.

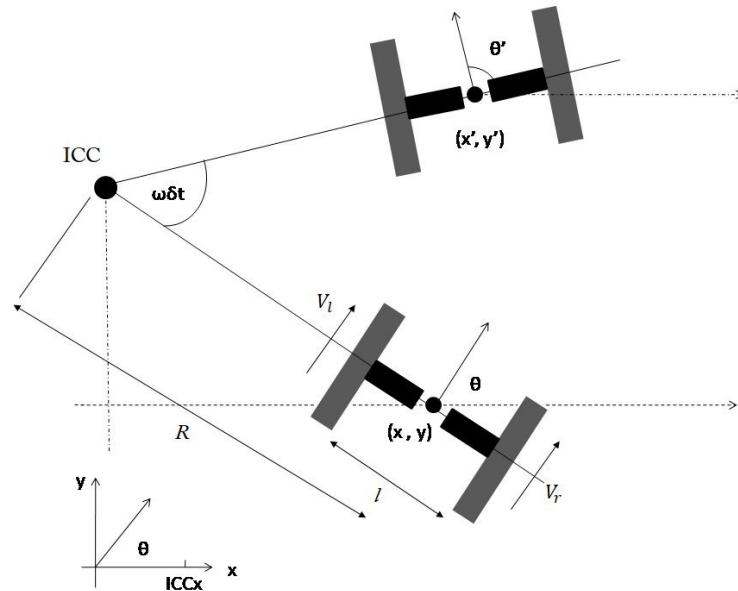


Figure 10: Forward kinematic diagram of differential drive

You can also refer the equations of inverse kinematics of mobile robotics from the following reference.



For more information, check the publication titled *Kinematics Equations for Differential Drive and Articulated Steering*, ISSN-0348-0542 and the first author is Thomas Hellstrom.

So we've seen the kinematics equations of this robot; the next stage is to simulate the robot.

Simulating Chefbot

The robot in the section is actually designed for carrying food and delivering to the customers in a hotel. It is called Chefbot. Now let's see what are the steps involved for simulating Chefbot. We are using the Gazebo simulator along with ROS for simulating the capabilities of a robot. We'll look at the basic teleoperation of the mapping and localization of a robot in Gazebo.

Building the URDF model of Chefbot

The first step in the simulation is building a robot model compatible with ROS. The URDF (<http://wiki.ros.org/urdf>) file is the robot model that is compatible with ROS. We are not going to discuss how to write a URDF model; instead, we will see the important sections we have to focus on while creating the URDF of a robot.

Inserting 3D CAD parts into URDF as links

Creating URDF is a time-consuming task; in this section, we will learn how to create a URDF package for a robot and insert 3D CAD models as a robot link in URDF. The first step is to create a robot description package; in this case, `chapter_9_codes/chefbot_code/chefbot_description` is our robot model ROS package. This package contains all the URDF files and 3D mesh files required for a robot. The `chefbot_description/meshes` folder has some 3D models that we designed earlier. These 3D models can be inserted into the URDF file. You can check the existing URDF file from `chefbot_description/urdf`. Here is a snippet that inserts a 3D model into URDF, which can act as a robot link. The code snippet can be found in `urdf/chefbot_base.urdf.xacro`.

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

Here, you can see we are inserting the `base_plate.dae` mesh into the URDF file.

Inserting Gazebo controllers into URDF

After inserting the link and assigning joints, we need to insert Gazebo controllers for simulating differential drive and the depth camera plugin, which is done with software models of actual robots. Here is a snippet of the differential drive Gazebo plugin. You can find this code snippet in `urdf/chefbot_base_gazebo.urdf.xacro`.

```
&lt;gazebo>
  &lt;plugin name="kobuki_controller"
filename="libgazebo_ros_kobuki.so">

    &lt;publish_tf>1&lt;/publish_tf>
    &lt;left_wheel_joint_name>wheel_left_joint
    &lt;/left_wheel_joint_name>
    &lt;right_wheel_joint_name>wheel_right_joint
    &lt;/right_wheel_joint_name>
    &lt;wheel_separation>.30&lt;/wheel_separation>
    &lt;wheel_diameter>0.09&lt;/wheel_diameter>
    &lt;torque>18.0&lt;/torque>
    &lt;velocity_command_timeout>0.6&lt;/velocity_command_timeout>

    &lt;imu_name>imu&lt;/imu_name>
  &lt;/plugin>
&lt;/gazebo>
```

In this plugin, we are providing the designed values of the robot, such as motor torque, wheel diameter, and wheel separation. The differential drive plugin that we are using here is `kobuki_controller`, which is used in the TurtleBot simulation.

After creating this controller, we need to create a depth sensor plugin for mapping and localization. Here is the code snippet to simulate the Kinect, a depth sensor. You can find the code snippet from `urdf/chefbot_gazebo.urdf.xacro`.

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

In the depth sensor plugin, we can provide necessary design values inside it for simulating the same behavior.



You can clone the section code using the following command: `$ git clone`



https://github.com/qboticslabs/ros_robotics_projects

Running the simulation

To simulate the robot, you may need to satisfy some dependencies. The differential robot controller used in our simulation is of Turtlebot. So we have to install Turtlebot packages to get those plugins and run the simulation:

```
| $ sudo apt-get install ros-kinetic-turtlebot-simulator ros-kinetic-turtlebot-navigation ros-kinetic-create-node  
| ros-kinetic-turtlebot-bringup ros-kinetic-turtlebot-description
```

You can also install ROS packages such as `chefbot_bringup`, `chefbot_description`, `chefbot_simulator` to start the simulation. You can copy these package into your ROS workspace and launch the simulation using the following command:

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

If everything is working properly, you will get this window, which has the designed robot:

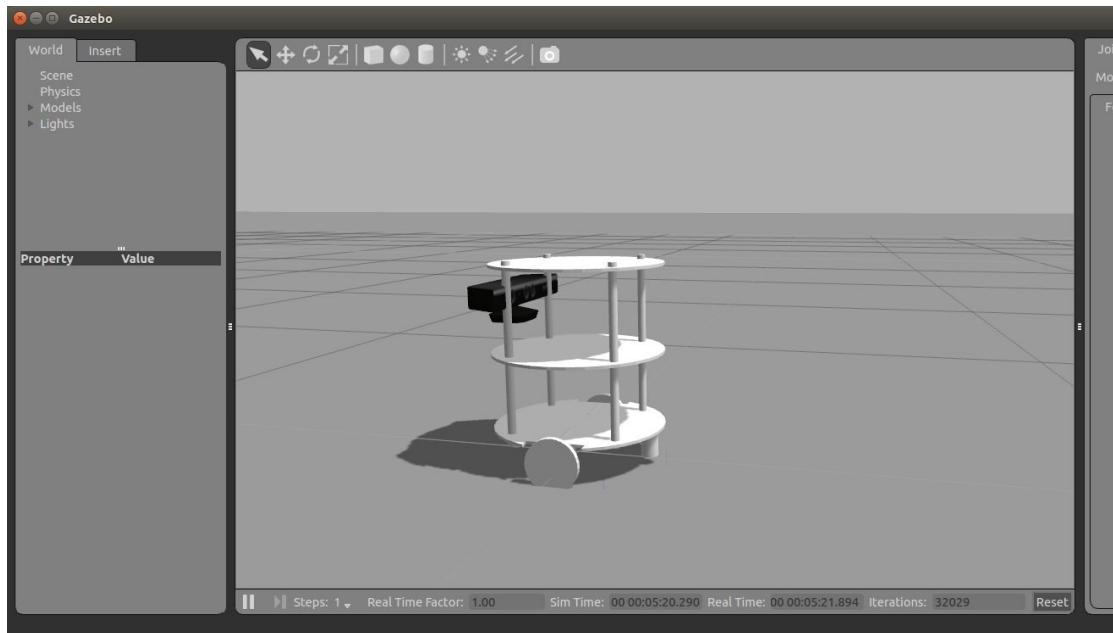


Figure 11: Simulation of Chefbot in Gazebo

You can move the robot around using a teleop node. You can start teleop using the following command:

```
| $ roslaunch chefbot_bringup keyboard_teleop.launch
```

You can move the robot with your keyboard, using the keys shown in the following screenshot:

```
Control Your Turtlebot!
-----
Moving around:
  u    i    o
  j    k    l
  m    ,    .

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

CTRL-C to quit

currently:      speed 0.2      turn 1
```

Figure 12: Keyboard teleop

If you can move the robot using teleop, you can now implement its remaining capabilities.

Mapping and localization

Now we can perform mapping and localization of the simulated robot. Mapping is done using the ROS `gmapping` package, which is based on the **Simultaneous Localization and Mapping (SLAM)** algorithm, and localization is done using the `amcl` **Adaptive Monte Carlo Localization (AMCL)** package, which has an implementation of the AMCL algorithm.

In this section, we will launch a new simulated world and see how to map and localize in the world.

Mapping

Here is the command to start the simulated world that has our robot:

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

This will launch the world as shown in the following screenshot. The environment is similar to a hotel conference room with tables placed in it:

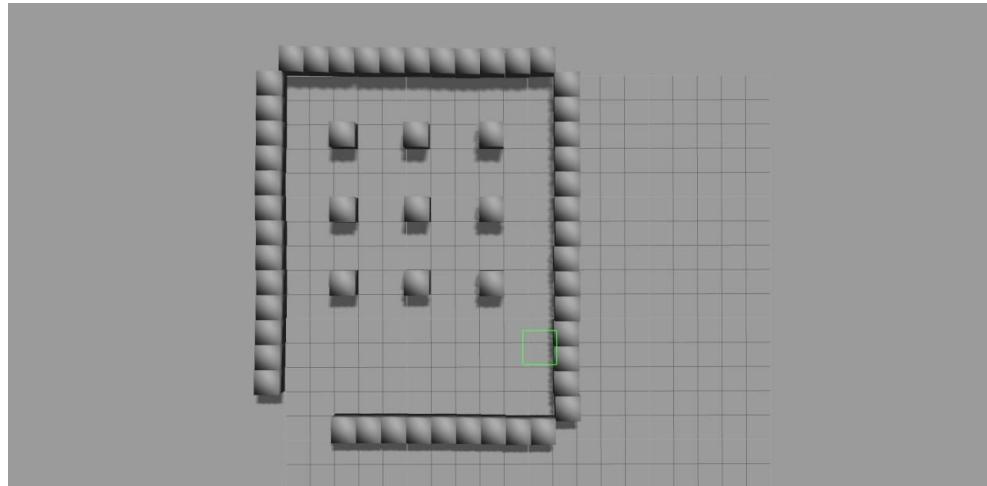


Figure 13: Hotel environment in Gazebo

To start mapping the environment, we can use the following launch file. This will start the `gmapping` node and finally create the map file.

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

After launching `gmapping` nodes, we can start Rviz for visualizing the map building done by the robot. The following command will start Rviz with necessary settings to view the map file:

```
| $ roslaunch chefbot_bringup view_navigation.launch
```

You can start the teleop node and move around the world; this will create a map like the following:

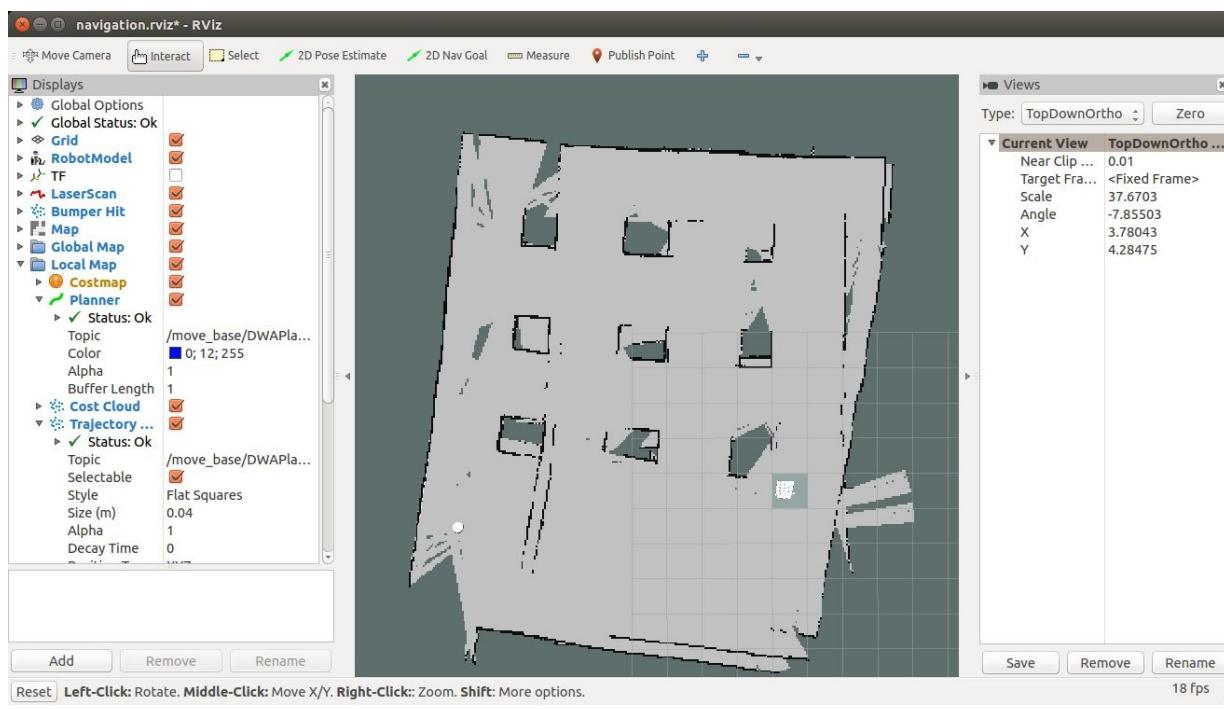


Figure 14: The map visualized in Rviz

After building the map, we can save it using the following command:

```
| $ rosrun map_server map_saver -f ~/hotel_world
```

This will save the map in the `home` folder with the name `hotel_world`.

Congratulations; you have successfully built the map of the world and saved it. The next step is to use this map and navigate autonomously around the world. We need the `amcl` package to localize on the map. Combining this with the `amcl` package and ROS navigation, we can autonomously move around the world.

Navigation and localization

Close all the Terminals we have used for mapping, and launch the simulated world in Gazebo using the following command:

```
| $ roslaunch chefbot_gazebo chefbot_world.launch
```

Start localization using the following command:

```
| $ roslaunch chefbot_gazebo amcl_demo.launch map_file:=~/home/<user_name>/hotel_world.yaml
```

This will load the saved map and `amcl` nodes. To visualize the robot, we can start Rviz using the following command:

```
| $ roslaunch chefbot_bringup view_navigation.launch
```

Now, we can start navigating the robot autonomously. You can click on the 2D Nav Goal button and click on the map to set the destination. When we set the position, the robot will autonomously move from its starting point to the destination, as shown here:

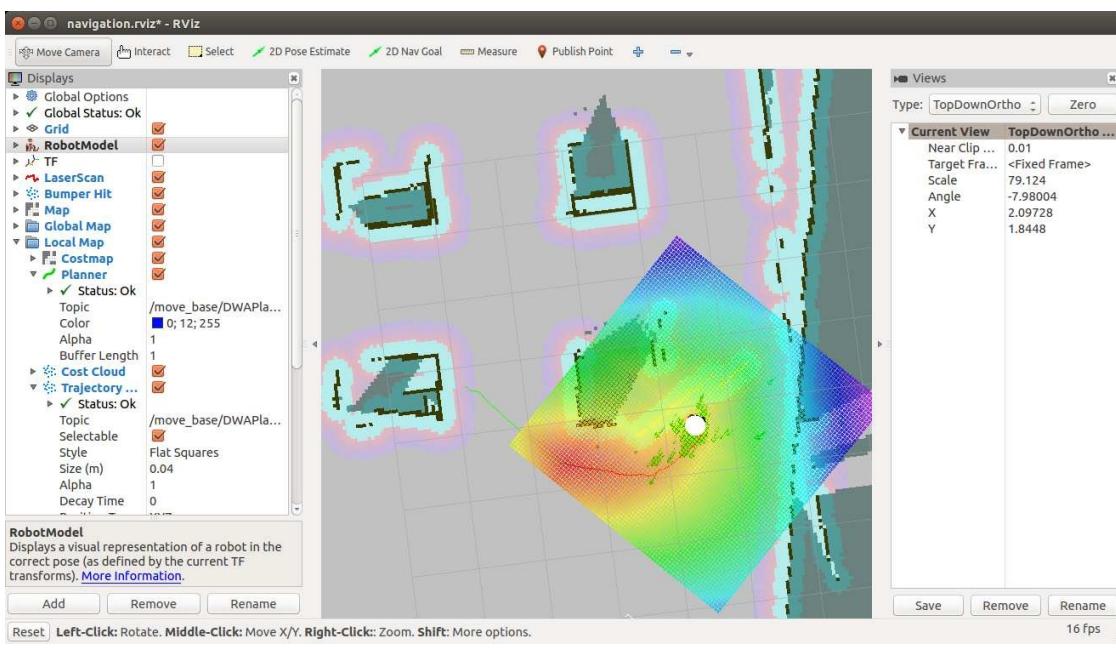


Figure 15: Visualizing autonomous navigation with AMCL particles

Congratulations! You have successfully set up the robot simulation and performed autonomous navigation using the simulator. Now let's see how we can create the actual robot hardware and program it.

Designing and building actual robot hardware

Let's build the actual hardware of this robot. We need components that satisfy our design values and additional vision sensors to perform SLAM and AMCL. Here is the list::

| No | Component name | Link |
|----|--|---|
| 1 | DC gear motor with encoder | https://www.pololu.com/product/2824 |
| 2 | Motor driver | https://www.pololu.com/product/708 |
| 3 | Tiva C 123 or 129 Launchpad | http://www.ti.com/tool/EK-TM4C123GXL OR http://www.ti.com/tool/EK-TM4C1294XL |
| 4 | Ultrasonic sensor | http://www.robotshop.com/en/hc-sr04-ultrasonic-range-finder.html |
| 5 | MPU 6050 (IMU) | http://www.robotshop.com/en/mpu-6050-6-dof-gyro-accelerometer-imu.html http://www.robotshop.com/en imu-breakout-board-mpu-9250.html |
| 6 | OpenNI compatible depth sensor (Astra Pro) | https://orbbec3d.com/product-astra-pro/ |
| 7 | Intel NUC | http://www.intel.in/content/www/in/en/nuc/products-overview.html |
| 8 | 12V, 10AH battery | Any battery with the specifications provided |

Let's discuss the use of each hardware part of the robot.

Motor and motor driver

The motors are controlled using a motor driver circuit. Adjusting the speed of the motors will adjust the speed of the robot. The motor drivers are basically H-bridges that are used to control the speed and direction of the motors. We are using motors and drivers from Pololu. You can check them out from the link in the table.

Motor encoders

Motor encoders are sensors that provide a count corresponding to the speed of the robot wheel. Using the encoder counter, we can compute the distance travelled by each wheel.

Tiva C Launchpad

The Tiva C Launchpad is the embedded controller board used to control the motor and interface with other sensors. The board we are using here is running at 80 MHz and on 256 KB of flash memory. We can program this board using the Arduino language, called **Wiring** (<http://wiring.org.co/>).

Ultrasonic sensor

The ultrasonic sensor is used to detect nearby obstacles, if any, in front of the robot. This sensor is an optional one; we can enable or disable it in the embedded controller code.

MPU 6050 The IMU of the robot is used improve the odometry data from the robot. The odometry data provides the current robot position and orientation with respect to its initial position. Odometry data is important while building a map using SLAM.

OpenNI depth sensor

To map the environment, we will need a laser scanner or a depth sensor. Laser scanner data is one of the inputs to the SLAM node. One of the latest depth sensors we can use is the Orbbec Astra Pro (<https://orbbec3d.com/product-astra-pro/>). You can also use a Kinect for this purpose. Using the `depthimage_to_laserscan` (http://wiki.ros.org/depthimage_to_laserscan) ROS package, we can convert the depth value to laser scan data.

Intel NUC

To run ROS and its packages, we need a computer. A compact PC we can use is the Intel NUC. It can smoothly run all the packages needed for our robot.

Interfacing sensors and motors with the Launchpad

In this section, we will see how to interface each sensor with the Launchpad. The Launchpad can be used to interface motor controllers and also to interface sensors. Here is a block diagram showing how to connect the Launchpad and sensors:

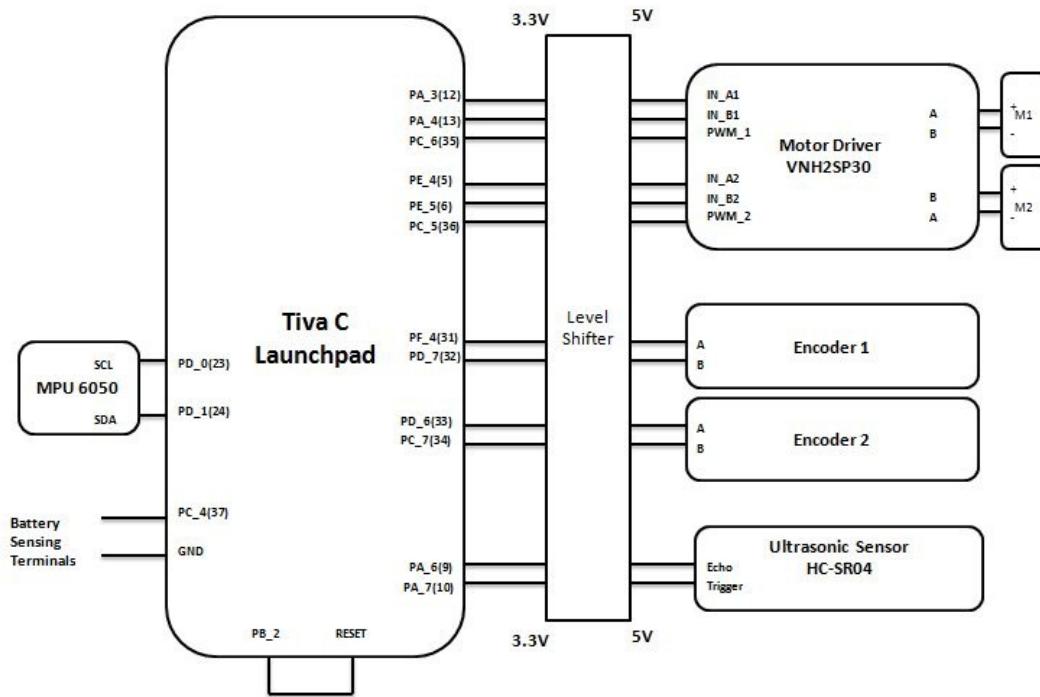


Figure 16: Interconnection between the Launchpad and sensors

The Launchpad works on 3.3V (CMOS) logic, so we may need a logic level shifter to convert from 3.3V to 5V and vice versa. In board like Arduino UNO is having 5V level, so it can directly interface to motor driver without any need of level shifter. Most of the ARM based controller boards are working in 3.3V, so level shifter circuit will be essential while interfacing to a 5V compatible sensor or circuit.



You can clone the section code using the following command: `$ git clone https://github.com/qboticslabs/ros_robotics_projects`

Programming the Tiva C Launchpad

The programming of the Tiva C Launchpad is done using the Energia IDE, which is the customized version of the Arduino IDE. You can download it from <http://energia.nu/>. As with Arduino, you can choose the serial port of the board and the board name.

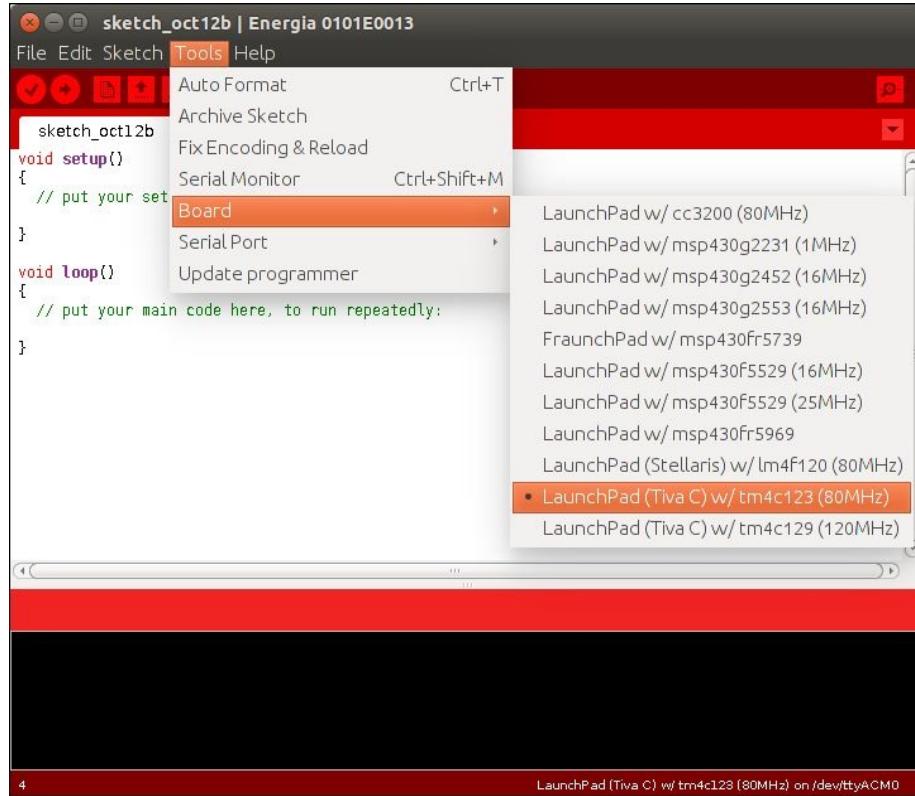


Figure 17: Energia IDE

The embedded code is placed in the `chapter_9_codes/ chefbot_code/tiva_c_energia_code_final` folder. Let's look at some important snippets from the main embedded code.

Here are headers files of the main code. We need to include the following MPU 6050 headers to reading values from it. The `MPU6050` library for Energia is also given along with the section's code:

```
#include "Wire.h"
#include "I2Cdev.h"
#include "MPU6050_6Axis_MotionApps20.h"
```

The `Messenger` library is used to handle serial data from the PC:

```
#include <Messenger.h>
#include <limits.h>
```

In the following code, the first line is the object of the `MPU6050` class for handling data from the IMU, and the second one is the object of the `Messenger` library for handling serial input:

```
MPU6050 accelgyro(0x68);
Messenger Messenger_Handler = Messenger();
```

The following is the main `setup()` function of the code. This will initialize all sensors and motors of the robot. The `setup()` function will initialize the serial port with a baud rate of `115200` and initialize encoders, motors, ultrasonic, `MPU6050`, and the messenger object. You can see the definition of each function in the code itself.

```
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);

}
```

The following is the main `loop()` function of the code. It will read sensor values and send motor speed commands to the motor driver. The speed commands are received from the PC.

```
void loop()
{
    //Read from Serial port
    Read_From_Serial();

    //Send time information through serial port
    Update_Time();

    //Send encoders values through serial port
    Update_Encoders();

    //Send ultrasonic values through serial port
    Update_Ultra_Sonic();

    //Update motor values with corresponding speed and send speed
    //values through serial port
    Update_Motors();

    //Send MPU 6050 values through serial port
    Update_MP6050();

    //Send battery values through serial port
    Update_Battery();

}
```

We can compile the code and upload it into the board using Energia. If the upload is successful, we can communicate with the board using the `miniterm.py` tool.

Assume that the serial port device is `/dev/ttyACM0`. First, change the permission using following command:

```
| $ sudo chmod 777 /dev/ttyACM0
```

We can communicate with the board using the following command:

```
| $ miniterm.py /dev/ttyACM0 115200
```

If everything is successful, you will get values like these:

```
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 18: The serial port values from the board

The messages that you are seeing can be decoded like this: the first letter denotes the device or parameter. Here is what the letters mean:

| Letter | Device or parameter |
|--------|---------------------|
| b | Battery |
| t | Time |
| e | Encoder |
| u | Ultrasonic sensor |
| s | Motor speed |
| i | IMU value |

The serial messages are separated by spaces and tabs so that each value can be decoded easily.

If we are getting serial messages, we can interface the board with ROS.



The latest ROS Tiva C Launchpad interface can be found here: http://wiki.ros.org/roserial_tivac.

Interfacing robot hardware with ROS

In this section, we will see how we can interface a robot's embedded controller with ROS. The embedded controller can send speed commands to the motors and obtain speed commands from robot controller nodes. The ROS robot controller nodes receive linear and angular Twist command from the ROS navigation stack. The Twist command will be subscribed to by the robot controller node and converted into equivalent motor velocity, that is Vl and Vr .

The robot controller nodes also receive encoder ticks from the embedded controller and calculate the distance traveled by each wheel. Let's take a look at the robot controller nodes.

The Chefbot robot controller nodes are placed in `chefbot_bringup/scripts`. You can check out each node; they're all written in Python.

- `launchpad_node.py`: This is the ROS driver node for handling Launchpad boards. This node will receive serial data from Launchpad and also send data to the board. After running this node, we will get serial data from the board as topics, and we can send data to the board through topics too.
- `serialDataGateway.py`: This Python module is used to handle serial receive or transmit data in a thread. The `launchpad_node.py` node uses this module to send or receive data to or from the board.
- `Twist_to_motors.py`: This node will subscribe to Twist messages from the ROS navigation stack or teleop node and convert them into wheel target velocities.
- `pid_velocity.py`: This is a node that implements the PID controller, which subscribes to the wheel target velocity and converts it into equivalent motor velocity.
- `diff_tf.py`: This node basically subscribes to the encoder data and calculates the distance traversed by the robot. It then publishes as the odometry and **transformation (TF)** topic.

Here is the graph showing the communication between the nodes:

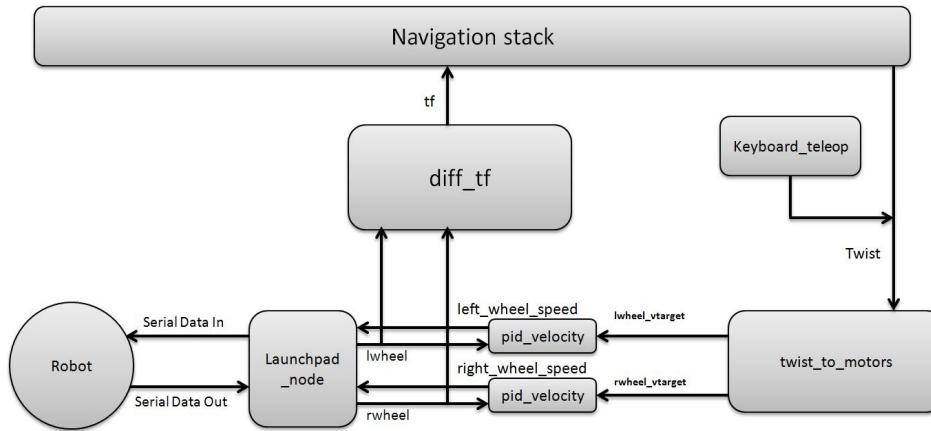


Figure 19: Communication among ROS driver nodes

Here is the list of ROS launch files that we need in order to work with the actual robot. All launch files are placed in the `chefbot_bringup/launch` folder:

- `robot_standalone.launch`: This will launch the ROS driver nodes of Chefbot.
- `model_robot.launch`: This launch file loads the URDF file of Chefbot.
- `view_robot.launch`: This will display the robot model on Rviz.
- `keyboard_teleop.launch`: This will start the keyboard teleop node, which can drive the robot using a keyboard.
- `3dsensor.launch`: This will launch OpenNI to enable depth camera drivers. There may changes to this launch file according to the sensor.
- `gmapping_demo.launch`: This will launch the `gmapping` nodes, which will help us map the robot environment.
- `amcl_demo.launch`: This will launch the AMCL nodes, which help us localize the robot on the map.
- `view_navigation.launch`: This will visualize the map and robot, which helps us command the robot to move to the destination on the map.



Orbbec Astra camera ROS driver: http://wiki.ros.org/astra_camera https://github.com/orbbec/ros_astra_camera

Running Chefbot ROS driver nodes

The following is the block diagram of the connection. Make sure that you are all set with connecting the devices. Make sure you have connected all sensors and the Launchpad board to your PC before running the driver.

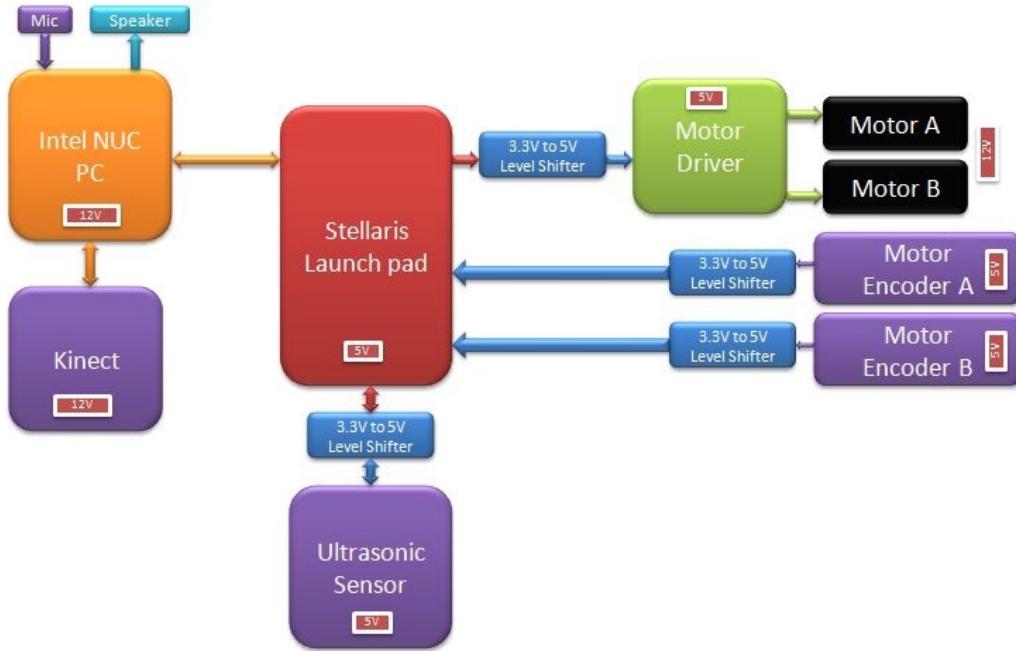


Figure 20: Block diagram of the Chefbot

If we want to launch all driver nodes of the robot, you can simply do it using the following command. Don't forget to change the serial port permission.

```
| $ roslaunch chefbot_bringup robot_standalone.launch
```

If everything working fine, you will get the following ROS topics:

```
robot@robot-pc:~$ 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
/tf_static
/ultrasonic_distance
```

Figure 21: The Chefbot driver topics

You can also visualize the ROS computational graph using `rqt_graph`. Here is the visualization of `rqt_graph`, showing the communication between all nodes:

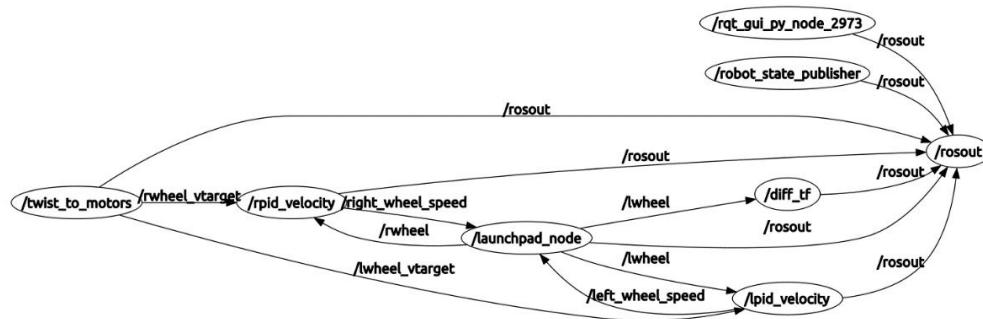


Figure 22: The computation graph view of Chefbot driver nodes

Gmapping and localization in Chefbot

After launching the ROS driver, we can teleop the robot using keyboard teleop. We can use the following command to start keyboard teleoperation:

```
| $ roslaunch chefbot_bringup keyboard_teleop.launch
```

If we want to map the robot environment, we can start the gmapping launch file like we did in the simulation:

```
| $ roslaunch chefbot_bringup gmapping_demo.launch
```

You can visualize the map building in Rviz using the following command:

```
| $ roslaunch chefbot_bringup view_navigation.launch
```

You can build the map by teleoperating the robot around the room. After mapping, save the map as we did in the simulation:

```
| $ rosrun map_server map_saver -f ~/test_map
```

After getting the map, launch AMCL nodes to perform final navigation. You have to restart all the launch files and start again.

Let's look at the commands to launch the AMCL nodes.

First, start the ROS driver nodes using the following command:

```
| $ roslaunch chefbot_bringup robot_standalone.launch
```

Now start the AMCL nodes:

```
| $ roslaunch chefbot_bringup amcl_demo.launch map_file:~/test_map.yaml
```

Then start Rviz to command the robot on the map:

```
| $ roslaunch chefbot_bringup view_navigation.launch
```

You will see Rviz showing something like the following screenshot, in which you can command the robot and the robot can run autonomously:

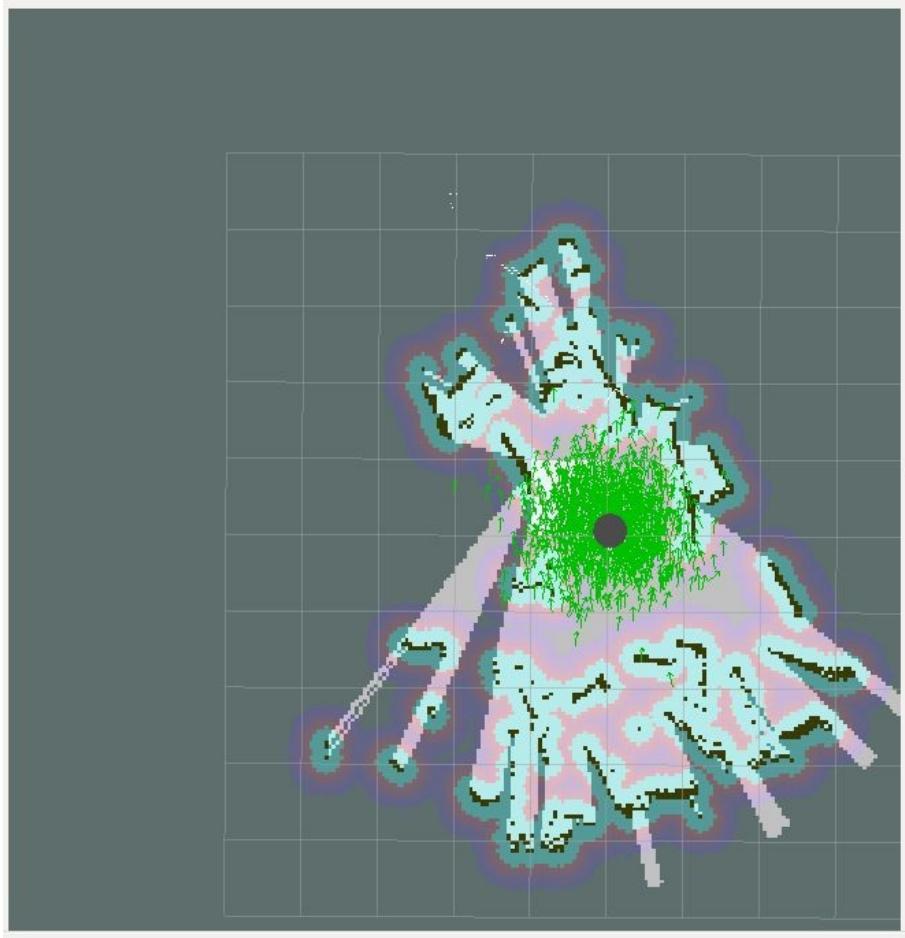


Figure 23: Localization and navigation with Chefbot

The following diagram shows the actual robot hardware. As per our design, we can see circular plate and hollow tubes to add additional layers to the robot. You can also see the **Intel NUC** and **Kinect** camera for robot navigation:

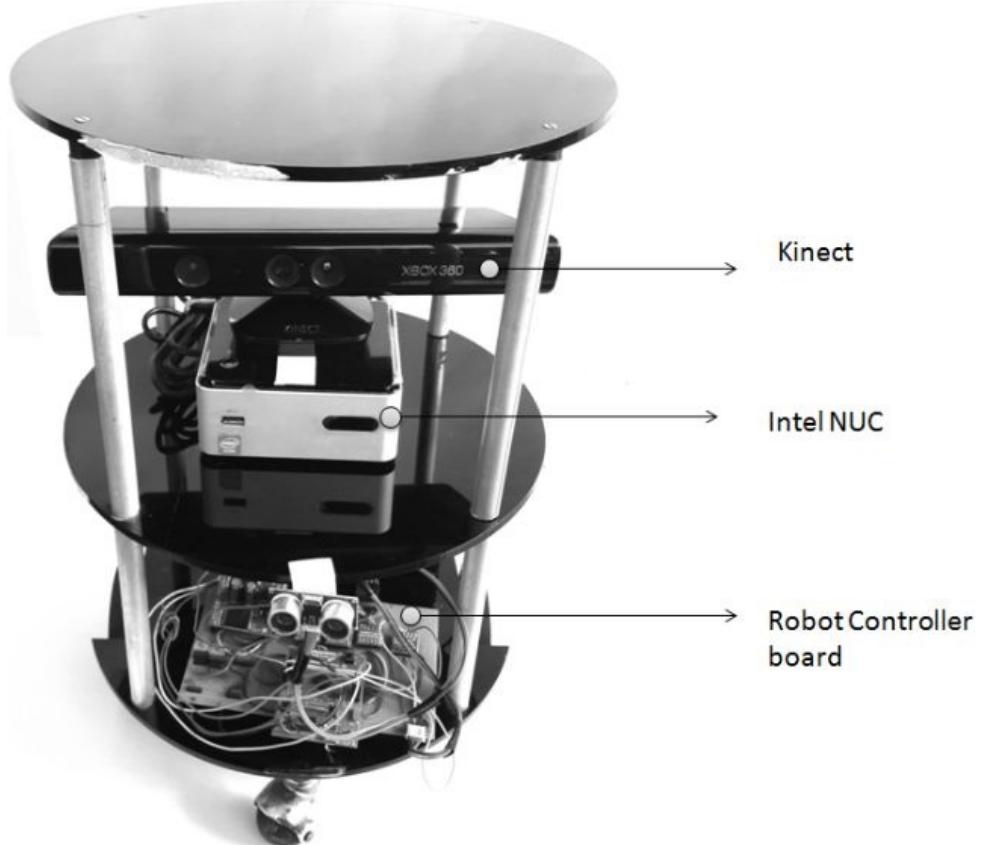


Figure 24: The actual Chefbot prototype



Questions

- How to convert encoder data to estimate the robot's position?
- What is the role of SLAM in robot navigation?
- What is AMCL and why is it used?
- What is the importance of the ROS navigation stack?

Summary

In this chapter, we designed and built an autonomous mobile robot from scratch. The design of the robot started with its specification. From the specification, we designed various parameters of the robot, such as motor torque and speed. After finding out each parameter, we modeled the robot chassis and simulated it using ROS and Gazebo. After simulation, we saw how to create the actual hardware. We selected the components and interconnected the sensors and actuators to the embedded board. We wrote the firmware of the embedded board. The board can communicate with the PC on which the ROS is running. The ROS driver node receives the data from the robot and interfaces with the gmapping and AMCL packages to perform autonomous navigation.

In the next chapter, we will see how to create a self-driving car and interface to Robot Operating System.

Creating a Self-Driving Car Using ROS

In this chapter, we will discuss a big technology that is trending in the robotics industry: driverless cars, or self-driving cars. Many of you may have heard about this technology; those who haven't will get an introduction in the first section of the chapter. In this chapter, you can find the following important topics.

- Getting started with self-driving cars
- Software block diagram of a typical self-driving car
- Simulating and interfacing self-driving car sensors in ROS
- Simulating a self-driving car with sensors in Gazebo
- Interfacing a DBW car into ROS
- Introducing the Udacity open source self-driving car project
- Open source self-driving car simulator from Udacity

Creating a self-driving car from scratch is out of the scope of this section, but this chapter may give you an abstract idea of self-driving car components, and tutorials to simulate it.

Getting started with self-driving cars

Just imagine a car driving by itself without the help of anyone. Self-driving cars are robot cars that can think about and decide how to reach the destination. The passenger only needs to specify the destination, and the robot car will take you to the destination safely. To convert an ordinary car into a robotic car, we should add some robotic sensors to it. We know that for a robot, there should be at least three important capabilities. It should be able to *sense*, *plan*, and *act*. Self-driving cars satisfy all these requirements. We'll discuss all the components we need for building a self-driving car. Before discussing the building of self-driving car, let's go through some milestones in self-driving car development.

History of autonomous vehicles

The concept of automating vehicles started long ago. From 1930, people have been trying to automate cars and aircraft, but the hype of self-driving cars increased between 2004 and 2013. To encourage autonomous vehicle technology, the U.S. Department of Defense's research arm, DARPA, conducted a challenge called the Grand DARPA Grand Challenge in 2004. The aim of the challenge was to autonomously drive for 150 miles through a desert roadway. In this challenge, no team was able to complete the goal, so they again challenged engineers in 2007 (<http://archive.darpa.mil/grandchallenge/>), but this time, the aim was slightly different. Instead of a desert roadway, there was an urban environment spread across 60 miles. In this challenge, four teams were able to finish the goal. The winner of the challenge was Team Taran Racing from Carnegie Mellon University (<http://www.tartanracing.org/>). The second-place team was Stanford Racing from Stanford University (<http://cs.stanford.edu/group/roadrunner/>).

Here is the autonomous car that won the DARPA challenge:



Figure 1: Boss, the Tartan autonomous vehicle

After the DARPA Challenge, car companies started working hard to implement autonomous driving capabilities in their cars. Now, almost all car companies have their own autonomous car prototype. In 2009, Google started to develop their self-driving car project, now known as Waymo (<https://waymo.com/>). This project greatly influenced other car companies, and the project was lead by Sebastian Thrun (<http://robotics.stanford.edu/>), the former director of the Stanford Artificial Intelligence Laboratory (<http://ai.stanford.edu/>).

The car autonomously traveled around 2.7 million kilometers in 2016. Take a look at it:



Source : Google

Figure 2: The Google self-driving car

In 2015, Tesla motors introduced a semi-autonomous autopilot feature in their electric cars. It enables hands-free driving mainly on highways and everything. In 2016, Nvidia introduced their own self-driving car (<http://www.nvidia.com/object/drive-px.html>), built using their AI car computer called **NVIDIA-DGX-1** (<http://www.nvidia.com/object/deep-learning-system.html>). This computer was specially designed for the self-driving car and is the best for developing autonomous training driving models.

Other than self-driving cars, there are self-driving shuttles for campus mobility. A lot of startups are building self-driving shuttles now, and one of these startups is called **Auro robotics** (<http://www.auro.ai/>). Here is the shuttle they're building for campuses:



Source: <http://www.auro.ai/>

Figure 3: Self-driving shuttle from Auro robotics

There is tremendous progress happening in self-driving car technology. Latest reports say that by the end

of 2020, self-driving cars will conquer our roads (<http://www.businessinsider.com/report-10-million-self-driving-cars-will-be-on-the-road-by-2020-2015-5-6?IR=T>). One of the most common terms used when describing autonomous cars is a level of autonomy. Let's go through the different levels of autonomy used when describe an autonomous vehicle.

Levels of autonomy

- **Level 0:** Vehicles having level 0 autonomy are completely manual, with a human driver. Most old cars belong in this category.
- **Level 1:** Vehicles with level 1 autonomy will have a human driver, but they will also have a driver assistance system that can either automatically control the steering system or acceleration/deceleration using information from the environment. All other functions have to be controlled by the driver.
- **Level 2:** In level 2 autonomy, the vehicle can perform both steering and acceleration/deceleration. All other tasks have to be controlled by the driver. We can say that the vehicle is partially automated in this level.
- **Level 3:** In this level, it is expected that all tasks be performed autonomously, but at the same time, it is expected that a human will intervene whenever required. This level is called conditional automation.
- **Level 4:** At this level, there is no need for a driver; everything is handled by an automated system. This kind of autonomous system will work in a particular area under specified weather conditions. This level is called **high automation**
- **Level 5:** This level is called **full automation**. In this level, everything is heavily automated and can work on any road and any weather condition. There is no need for a human driver.

Functional block diagram of a typical self-driving car

The following shows the important components of a self-driving vehicle. The list of parts and their functionalities will be discussed in this section. We'll also look at the exact sensor that was used in the autonomous car for the DARPA Challenge.

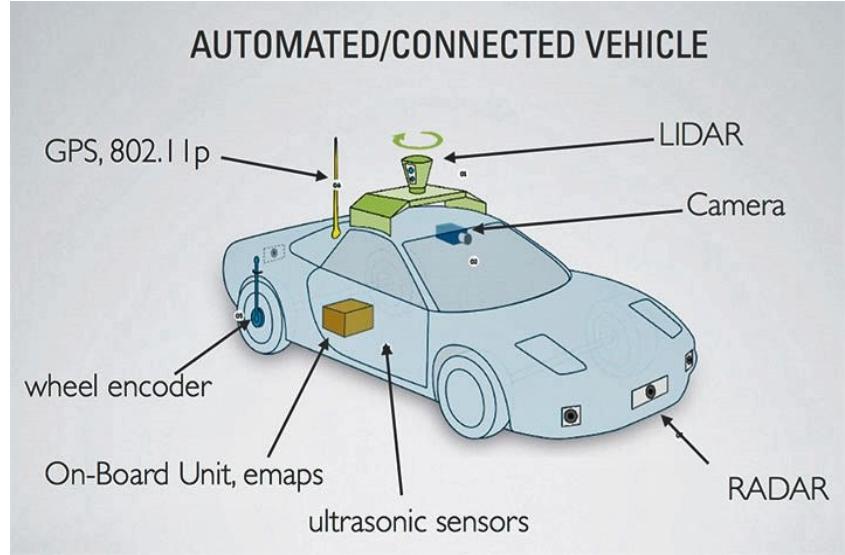


Figure 4: Important components of a self-driving car

GPS, IMU, and wheel encoders

As you know, the **Global Positioning System (GPS)** helps us determine the global position of a vehicle with the help of GPS satellites. The latitude and longitude of the vehicle can be calculated from the GPS data. The accuracy of GPS can vary with the type of sensor; some sensors have an error in the range of meters, and some have less than 1 meter of error. We can find vehicle state by combining GPS, **inertial measurement unit (IMU)** and wheel odometry data, and by using sensor fusion algorithms. This can give better estimate of the vehicle. Let's look at the position estimation modules used for the DARPA Challenge 2007.

POS LV modules from Applanix: This is the module used in the Standford autonomous car, *Junior*. It is a combination of GPS, IMU, and wheel encoders or **distance measurement indicator (DMI)**. You can find it at <http://www.applanix.com/products/poslv.html>.

Here is what the module looks like:



Source: <http://www.applanix.com/>

Figure 5: Applanix module for autonomous navigation

As you can see from the preceding image, there are wheel encoders, an IMU, and a GPS receiver provided with this package.

OxTS module: This is another GPS/IMU combo module from **Oxford Technical Solution (OxTS)** (<http://www.oxts.com/>). This module was extensively used in the DARPA Challenge in 2007. The module is from the RT 3000 v2 family (<http://www.oxts.com/products/rt3000-family/>). The entire range of GPS modules from OxTS can be found at <http://www.oxts.com/industry/automotive-testing/>. Here is the list of the autonomous vehicles that use these modules: <http://www.oxts.com/customer-stories/autonomous-vehicles-2/>. The following image shows the RT-3000 v2 module:



Figure 6: RT-3000 v2 module

Xsens MTi IMU

The Xsens MTi series has independent IMU modules that can be used in autonomous cars. Here is the link to purchase this product:

<http://www.xsens.com/products/mti-10-series/>

Camera

Most autonomous vehicles are deployed with stereo or monocular cameras to detect various things, such as traffic signal status, pedestrians, cyclists, and vehicles. Companies such as MobileEye (<http://www.mobileye.com/>) which has been acquired by Intel built their **advanced driving assistance system (ADAS)** using a sensor fusion of cameras and LIDAR data to predict obstacles and path trajectory.

Other than ADAS, we can also use our own control algorithms by only using camera data. One of the cameras used by the Boss robot car in DARPA 2007 was **Point Grey Firefly (PGF)** (<https://www.ptgrey.com/firefly-mv-usb2-cameras>). These are high dynamic range cameras and have a 45-degree **field of view (FOV)**:



Figure 7: Point Grey Firefly camera

Ultrasonic sensors

In an ADAS system, ultrasonic sensors play an important role in the parking of vehicles, avoiding obstacles in blind spots, and detecting pedestrians. One of the companies providing ultrasound sensors for ADAS systems is Murata (<http://www.murata.com/>). They provide ultrasonic sensors for up to 10 meters, which are optimum for a **parking assistance system (PAS)**. The following diagram shows where ultrasonic sensors are placed on a car:

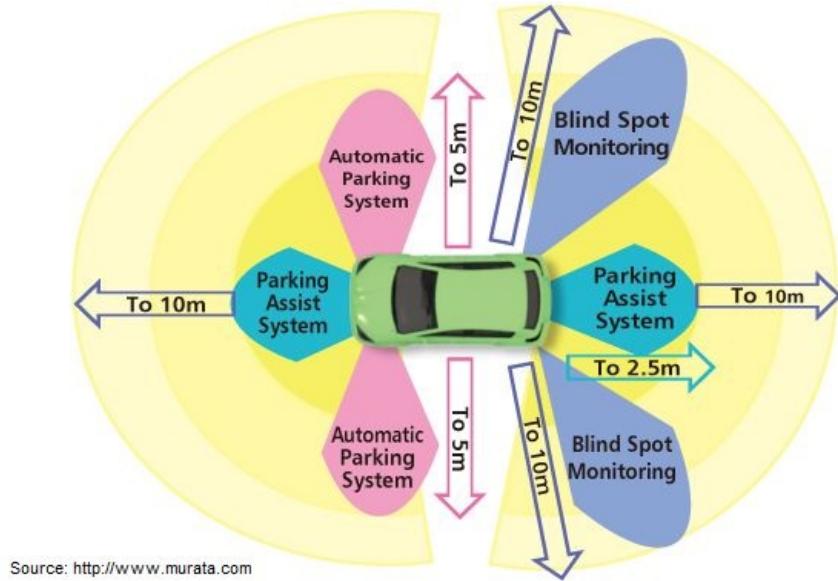


Figure 8: Placement of ultrasonic sensors for PAS

LIDAR and RADAR

The **LIDAR (Light Detection and Ranging)** (<http://oceanservice.noaa.gov/facts/lidar.html>) sensors are the core sensors of a self-driving car. A LIDAR sensor basically measures the distance to an object by sending a laser signal and receiving its reflection. It can provide accurate 3D data of the environment, computed from each received laser signal. The main application of LIDAR in autonomous car is mapping the environment from the 3D data, obstacle avoidance, object detection, and so on. Some of the LIDARs used in the DARPA Challenge are will be discussed here.

Velodyne HDL-64 LIDAR

The Velodyne HDL-64 sensor is designed for obstacle detection, mapping, and navigation for autonomous cars. It can give us 360-degree view laser-point cloud data with a high data rate. The range of this laser scan is 80 to 120 m. This sensor is used for almost all the self-driving cars available today. A list of Velodyne sensors available on the market can be found at <http://velodynelidar.com/products.html>.

Here are a few of them:



Source: <http://velodynelidar.com>

Figure 9: Some Velodyne sensors

SICK LMS 5xx/1xx and Hokuyo LIDAR

The company SICK (<https://www.sick.com>) provides a variety of laser scanners that can be used indoor or outdoor. The **SICK Laser Measurement System (LMS)** 5xx and 1xx models are commonly used in autonomous cars for obstacle detection. It provides a scanning range of 180 degrees and has high-resolution laser data. The list of SICK laser scanners available in the market is at <https://www.sick.com/in/en>. Another company, called **Hokuyo** (<http://www.hokuyo-aut.jp/index.html>), also builds laser scanners for autonomous vehicles. Here is the list of laser scanners provided by Hokuyo: <http://www.hokuyo-aut.jp/02sensor/>.

These are two laser scanners by SICK and Hokuyo:



Sick Laser



Hokuyo Laser

Figure 10: SICK and Hokuyo laser scanners



Some of the other LIDARs used in the DARPA Challenge provided given here: http://www.concept-i-online.com/www/industrial_sensors_de_en/ <https://www.ibeo-as.com/aboutibeo/lidar/>

Continental ARS 300 radar (ARS)

Apart from LIDARs, self-driving cars are also deployed with long-range radars. One of the popular long-range radars is ARS 30X by Continental (http://www.consilonline.com/www/industrial_sensors_de_en/themes/ars_300_en.html). It works using the Doppler principle and can measure up to 200 meters. Bosch also manufactures radars suitable for self-driving cars. The main application of radars is collision avoidance. Commonly, radars are deployed at the front of the vehicles.

Delphi radar

Delphi has a new radar for autonomous cars. Here is the link to view the product:

<http://www.delphi.com/manufacturers/auto/safety/active/electronically-scanning-radar>

On-board computer

The onboard computer is the heart of the self-driving car. It may have high-end processors such as Intel Xenon and GPUs to crunch data from various sensors. All sensors are connected to this computer, and it finally predicts the trajectory and sends control commands, such as steering angle, throttle, and braking for the self-driving car.

Software block diagram of self-driving cars

In this section, we will discuss a basic software block diagram of a self-driving car that was in DARPA Challenge:

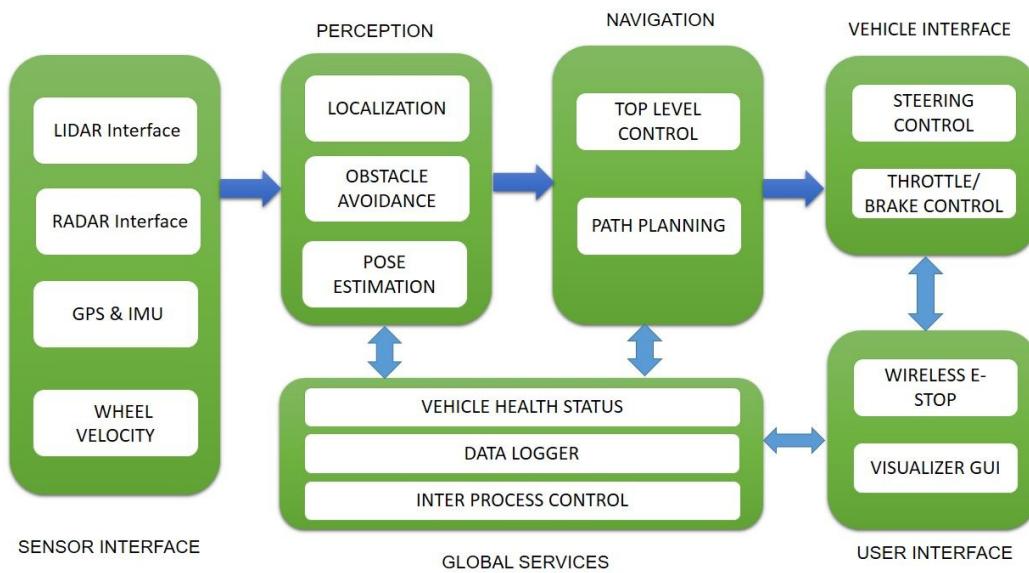


Figure 11: Software block diagram of a self-driving car

Let's learn what each block means. Each block can interact with others using **inter-process communication (IPC)** or shared memory. ROS messaging middleware is a perfect fit in this scenario. In DARPA Challenge, they implemented a publish/subscribe mechanism to do these tasks. One of the IPC library development by MIT for 2006 DARPA challenge was **Lightweight Communications and Marshalling (LCM)**. You may can learn more about LCM from the following link (<https://lcm-proj.github.io/>).

- **Sensor interface modules:** As the name of the module indicates, all the communication between the sensors and the vehicle is done in this block. The block enables us to provide the various kinds of sensor data to all other blocks. The main sensors include LIDAR, camera, radar, GPS, IMU, and wheel encoders.
- **Perception modules:** These modules perform processing on perception data from sensors such as LIDAR, camera, and radar and segment the data to find moving and static objects. They also help localize the self-driving car relative to the digital map of the environment.
- **Navigation modules:** This module determines the behavior of the autonomous car. It has motion planners and finite state machines for different behaviors in the robot.
- **Vehicle interface:** After the path planning, the control commands, such as steering, throttle, and brake control, are sent to the vehicle through a **drive-by-wire (DBW)** interface. DBW basically works through the CAN bus. Only some vehicles support the DBW interface. Examples are the Lincoln MKZ, VW Passat Wagon, and some models from Nissan.
- **User interface:** The user interface section provides controls to the user. It can be a touch screen to view maps and set the destination. Also, there is an emergency stop button for the user.
- **Global services:** This set of modules helps log the data and has time stamping and message-passing

support to keep the software running reliably.

Simulating and interfacing self-driving car sensors in ROS

In the preceding section, we discussed the basic concepts of a self-driving car. That understanding will definitely help in this section too. In this section, we are simulating and interfacing some of the sensors that we are using in self-driving cars. Here is the list of sensors that we are going to simulate and interface with ROS:

- Velodyne LIDAR
- Laser scanner
- Camera
- Stereo camera
- GPS
- IMU
- Ultrasonic sensor

We'll discuss how to set up the simulation using ROS and Gazebo and read the sensor values. This sensor interfacing will be useful when you build your own self-driving car simulation from scratch. So if you know how to simulate and interface these sensors, it can definitely accelerate your self-driving car development.

Simulating the Velodyne LIDAR

The Velodyne LIDAR is becoming an integral part of a self-driving car. Because of high demand, there are enough software modules available for working with this sensor. We are going to simulate two popular models of Velodyne, called **HDL-32E** and **VLP-16**. Let's see how to do it in ROS and Gazebo.

In ROS-Kinetic and Indigo, we can install from a binary package or compile from source code. Here is the command to install Velodyne packages on ROS Kinetic:

```
| $ sudo apt-get install ros-kinetic-velodyne-simulator
```

In ROS Indigo, just replace the ROS distribution name:

```
| $ sudo apt-get install ros-indigo-velodyne-simulator
```

To install it from source code, just clone the source package to the ROS workspace using the following command:

```
| $ git clone https://bitbucket.org/DataspeedInc/velodyne_simulator.git
```

After cloning the package, you can build it using the `catkin_make` command. Here is the ROS wiki page of the Velodyne simulator:

http://wiki.ros.org/velodyne_simulator

So you are installed the packages. Now it's time to start the simulation of the Velodyne sensor. You can start the simulation using the following command:

```
| $ roslaunch velodyne_description example.launch
```

This command will launch the sensor simulation in Gazebo. Note that this simulation will consume a lot of RAM of your system; your system should have at least 8 GB before start the simulation.

You can add some obstacles around the sensor for testing, like this:

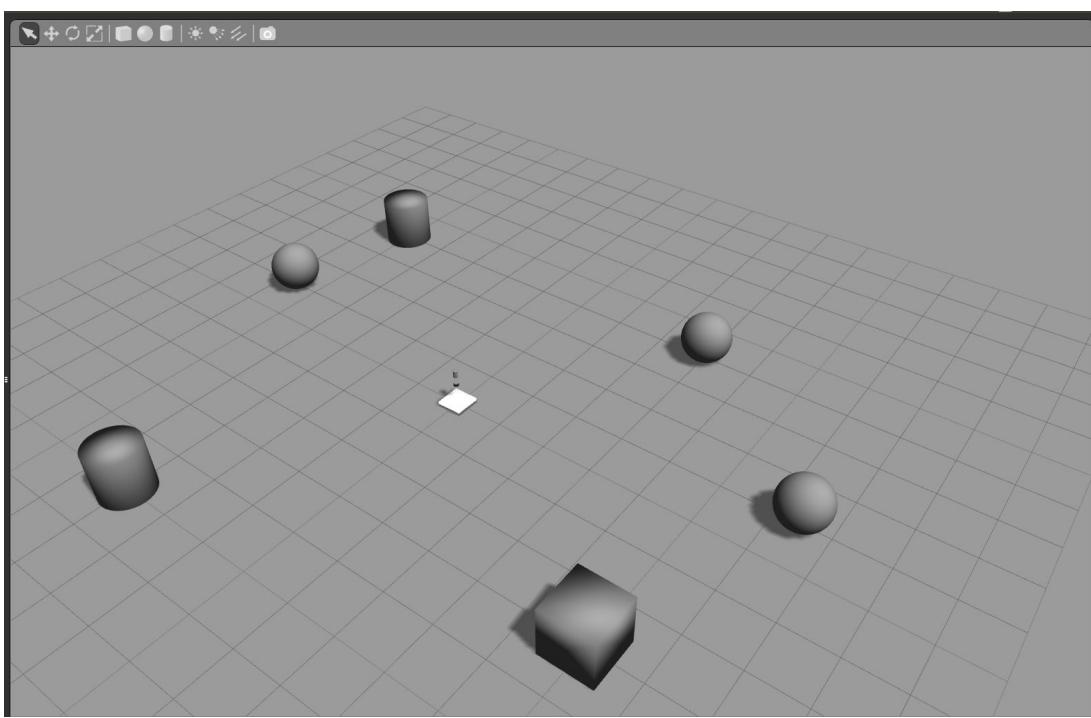


Figure 12: Simulation of Velodyne in Gazebo

You can visualize the sensor data in Rviz by adding display types such as PointCloud2 and Robot Model to visualize sensor data and sensor models. You have to set the Fixed Frame to velodyne. You can clearly see the obstacles around the sensor in the following figure:

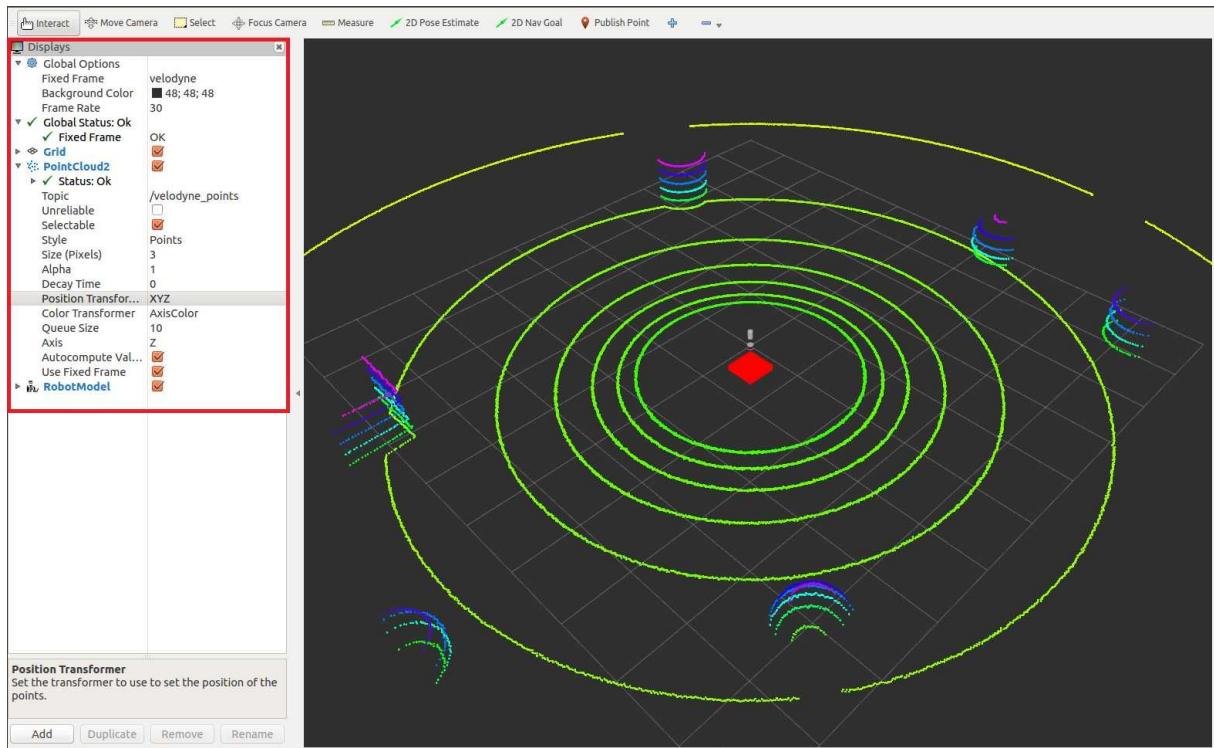


Figure 13: Visualization of a Velodyne sensor in Rviz

Interfacing Velodyne sensors with ROS

We have seen how to simulate a Velodyne sensor; now let's have a look at how we can interface a real Velodyne sensor with ROS.

The following commands are to install the `velodyne` ROS driver package to convert Velodyne data to point cloud data.

ROS Kinetic:

```
| $ sudo apt-get install ros-kinetic-velodyne
```

ROS Indigo:

```
| $ sudo apt-get install ros-indigo-velodyne
```

These commands will install the ROS Velodyne driver and point cloud converter packages.

This driver supports models such as the HDL-64E, HDL-32E, and VLP-16.

Here are the commands to start the driver nodelets:

```
| $ roslaunch velodyne_driver nodelet_manager.launch model:=32E
```

Here, you need to mention the model name along with the launch file to start the driver for a specific model.

The following command will start the converter nodelets to convert Velodyne messages (`velodyne_msgs/VelodyneScan`) to a point cloud (`sensor_msgs/PointCloud2`). Here is the command to perform this conversion:

```
| $ roslaunch velodyne_pointcloud cloud_nodelet.launch calibration:~/calibration_file.yaml
```

This will launch the calibration file for Velodyne, which is necessary for correcting noise from the sensor.

We can write all these commands to a launch file, which is shown in the following code block. If you run this launch file, the driver node and point cloud convertor nodelets will start, and we can work with the sensor data:

```
<launch>
  <!-- start nodelet manager and driver nodelets -->
  <include file="$(find
    velodyne_driver)/launch/nodelet_manager.launch" />

  <!-- start transform nodelet -->
  <include file="$(find
    velodyne_pointcloud)/launch/transform_nodelet.launch">
    <arg name="calibration"
      value="$(find
        velodyne_pointcloud)/params/64e_utexas.yaml"/>
  </include>
</launch>
```

The calibration files for each model are available in the `velodyne_pointcloud` package.



Note: The connection procedure of Velodyne to PC is given here: <http://wiki.ros.org/velodyne/Tutorials/Getting%20Started%20with%20the%20HDL-32E>

Simulating a laser scanner

In this section, we will see how to simulate a laser scanner in Gazebo. We can simulate it by providing custom parameters according to our application. When you install ROS, you also automatically install several default Gazebo plugins, which include Gazebo laser scanner plugin.

We can simply use this plugin and apply our custom parameters. For demonstration, you can use a tutorial package inside `chapter_10_codes` called `sensor_sim_gazebo`. You can simply copy the package to the workspace and build it using the `catkin_make` command. This package contains a basic simulation of the laser scanner, camera, IMU, ultrasonic sensor, and GPS.

Before starting with this package, you should install a package called `hector-gazebo-plugins` using the following command. This package contains Gazebo plugins of several sensors that can be used in self-driving car simulations.

```
| $ sudo apt-get install ros-kinetic-hector-gazebo-plugins
```

To start the laser scanner simulation, just use the following command:

```
| $ roslaunch sensor_sim_gazebo laser.launch
```

We'll first look at the output of the laser scanner and then dig into the code.

When you launch the preceding command, you will see an empty world with an orange box. The orange box is our laser scanner. You can use any mesh file to replace this shape according to your application. To show laser scanner data, we can place some objects in Gazebo, as shown here. You can add models from Gazebo's top panel.

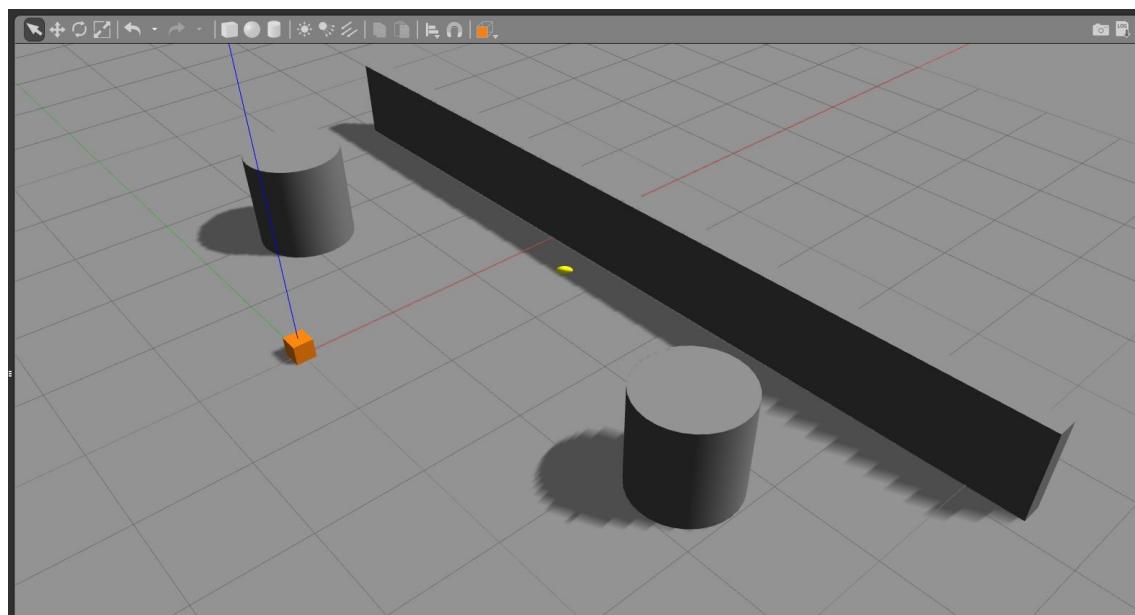


Figure 14: Simulation of a laser scanner in Gazebo

You can visualize the laser data in Rviz, as shown in the next screenshot. The topic to which the laser data

is coming is `/laser/scan`. You can add a LaserScan display type to view this data:

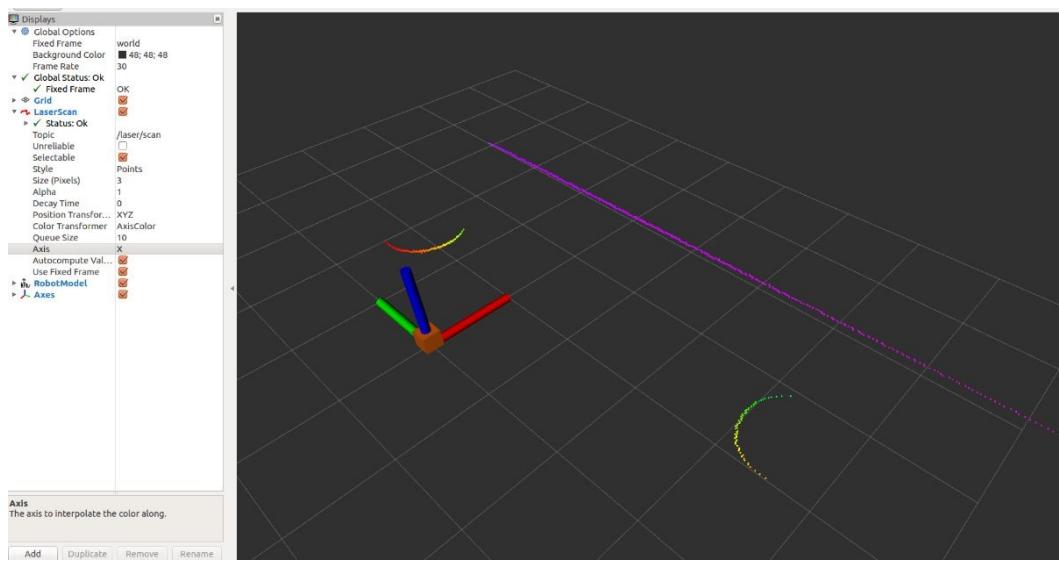


Figure 15: Visualization of laser scanner data in Rviz

You have to set the Fixed Frame to a world frame and enable the RobotModel and Axes display types in Rviz.

The following is the list of topics generated while simulating this sensor. You can see the `/laser/scan` topic.

```
robot@robot-pc:~$ rostopic list
/clicked_point
/clock
/gazebo/link_states
/gazebo/model_states
/gazebo/parameter_descriptions
/gazebo/parameter_updates
/gazebo/set_link_state
/gazebo/set_model_state
/initialpose
/joint_states
/laser/scan
/move_base_simple/goal
/rosout
/rosout_agg
/tf
/tf_static
```

Figure 16: List of topics from the laser scanner simulation

Explaining the simulation code

The `sensor_sim_gazebo` package has the following list of files for simulating all self-driving car sensors. Here is the directory structure of this package:

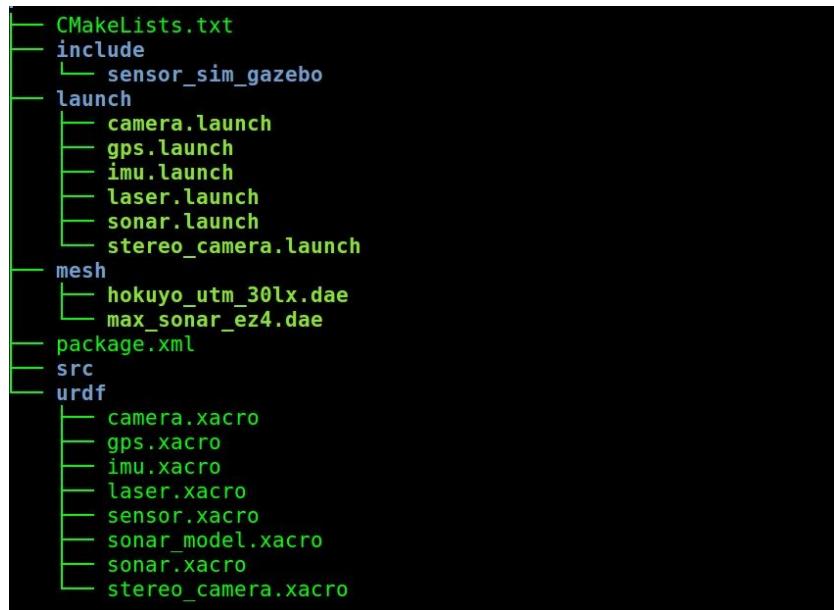


Figure 17: List of files in `sensor_sim_gazebo`

To simulate a laser, launch the `laser.launch` file; similarly, to start simulating the IMU, GPS, and camera, launch the corresponding launch files. Inside URDF, you can see the Gazebo plugin definition for each sensor. The `sensor.xacro` file is the orange box definition that you saw in the preceding simulation. It is just a box for visualizing a sensor model. We are using this model for representing all the sensors inside this package. You can use your own model instead of this, too.

The `laser.xacro` file has the Gazebo plugin definition of the laser, as shown here:

```
<gazebo reference="sensor">
  <sensor type="ray" name="head_hokuyo_sensor">
    <pose>0 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>0.8</min>
        <max>30.0</max>
        <resolution>0.01</resolution>
      </range>
      <noise>
        <type>gaussian</type>
        <!-- Noise parameters based on published spec for Hokuyo
laser
of 0.0m and
achieving "+-30mm" accuracy at range < 10m. A mean
stddev of 0.01m will put 99.7% of samples within
```

```

0.03m of the true
    reading. -->
    <mean>0.0</mean>
    <stddev>0.01</stddev>
  </noise>
</ray>
<plugin name="gazebo_ros_head_hokuyo_controller"
filename="libgazebo_ros_laser.so">
  <topicName>/laser/scan</topicName>
  <frameName>world</frameName>
</plugin>
</sensor>
</gazebo>

```

Here, you can see various parameters of the laser scanner plugin. We can fine-tune these parameters for our custom applications. The plugin we've used here is `libgazebo_ros_laser.so`, and all the parameters are passed to this plugin.

In the `laser.launch` file, we are creating an empty world and spawning the `laser.xacro` file. Here is the code snippet to spawn the model into Gazebo and start a joint-state publisher to start publishing TF data:

```

<param name="robot_description" command="$(find xacro)/xacro --
inorder '$(find sensor_sim_gazebo)/urdf/laser.xacro'" />

<node pkg="gazebo_ros" type="spawn_model" name="spawn_model"
args="-urdf -param /robot_description -model example"/>

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

```

Interfacing laser scanners with ROS

Now that we've discussed the simulation of the laser scanner, let's see how to interface real sensors with ROS.

Here are some links to guide you with setting up Hokuyo and SICK laser scanners in ROS. The complete installation instructions is available.

Hokuyo sensors: http://wiki.ros.org/hokuyo_node

SICK lasers: http://wiki.ros.org/sick_tim

You can install Hokuyo drivers from binary packages using the following commands:

Hokuyo laser scanners.

```
| $ sudo apt-get install ros-kinetic-hokuyo3d
```

SICK laser scanners.

```
| $ sudo apt-get install ros-kinetic-sick-tim ros-kinetic-lms1xx
```

Simulating stereo and mono cameras in Gazebo

In the previous section, we discussed laser scanner simulation. In this section, we will see how to simulate a camera. A camera is an important sensor for all kinds of robots. We will see how to launch both mono and stereo camera simulations. You can use the following commands to launch the simulations.

Mono camera:

```
| $ roslaunch sensor_sim_gazebo camera.launch
```

Stereo camera:

```
| $ roslaunch sensor_sim_gazebo stereo_camera.launch
```

You can view the image from the camera either using Rviz or using a tool called `image_view`.

You can look at the mono camera view using the following command:

```
| $ rosrun image_view image_view image:=/sensor/camera1/image_raw
```

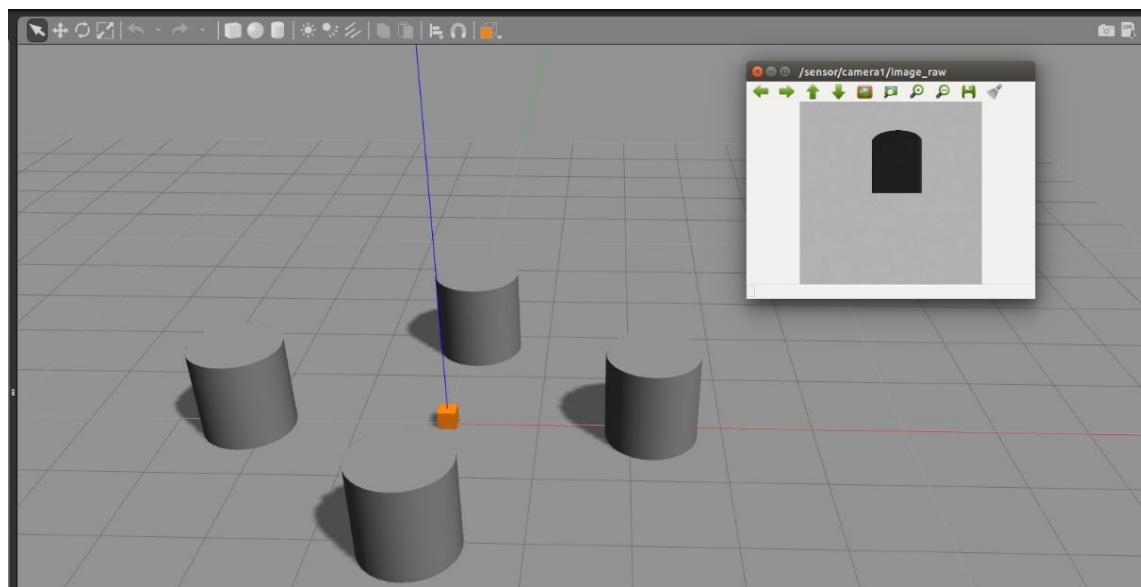


Figure 18: Image from simulated camera

To view images from a simulated stereo camera, use the following commands:

```
| $ rosrun image_view image_view image:=/stereo/camera/right/image_raw  
$ rosrun image_view image_view image:=/stereo/camera/left/image_raw
```

This commands will display two image windows from each camera of the stereo camera, which is shown here:

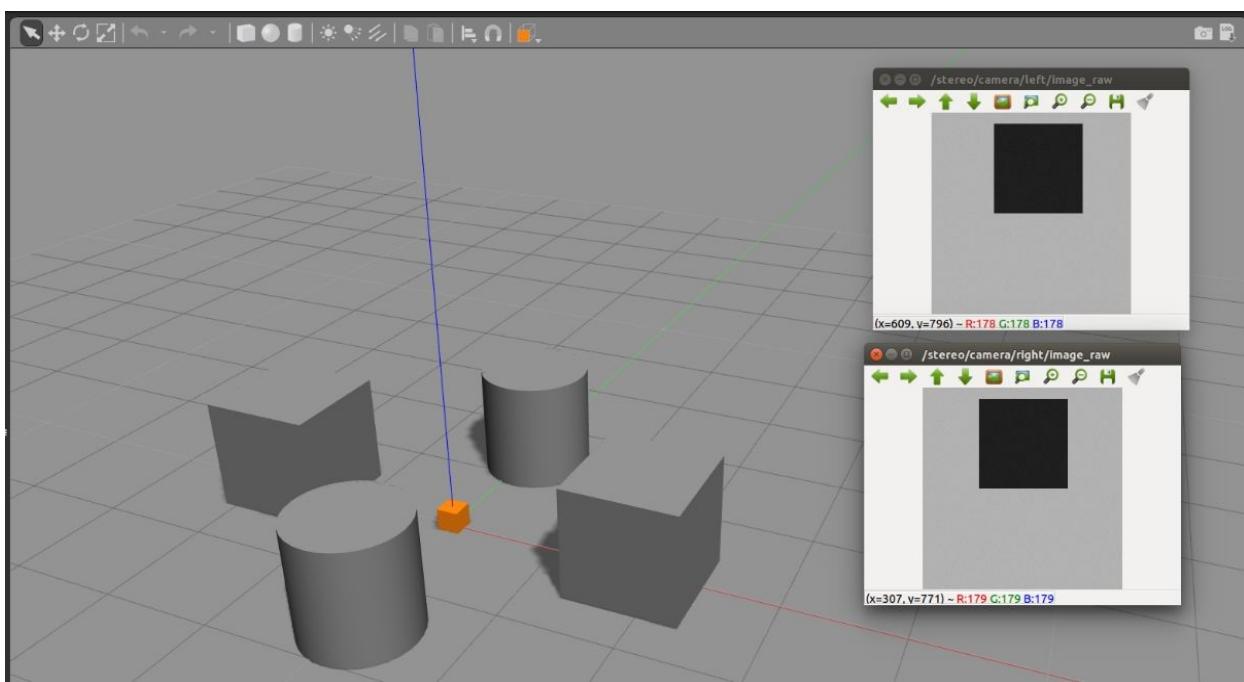


Figure 19: Image from simulated stereo camera

Similar to the laser scanner plugin, we are using a separate plugin for mono and stereo cameras. You can see the Gazebo plugin definition in `sensor_sim_gazebo/urdf/camera.xacro` and `stereo_camera.xacro`.

The `libgazebo_ros_camera.so` plugin is used to simulate a mono camera, and `libgazebo_ros_multicamera.so` for a stereo camera.

Interfacing cameras with ROS

In this section, we will see how to interface an actual camera with ROS. There are a lot of cameras available in the market. We'll look at some of the commonly used cameras and how to interface with them.

There are some links to guide you with setting up each driver in ROS.

For the **Point Grey** camera, you can refer to the following link: http://wiki.ros.org/pointgrey_camera_driver

If you are working with a Mobileye sensor, you may get ROS drivers by contacting the company. All details of the driver and its SDK are available at the following link:

<https://autonomousstuff.com/product/mobileye-camera-dev-kit>

If you are working on IEEE 1394 digital cameras, the following drivers can be used to interface with ROS: <http://wiki.ros.org/camera1394>

One of the latest stereo cameras available is the ZED camera (<https://www.stereolabs.com/>). The ROS drivers of this camera are available at the following link:

<http://wiki.ros.org/zed-ros-wrapper>

If you are working with some normal USB web camera, the `usb_cam` driver package will be best for interfacing with ROS: http://wiki.ros.org/usb_cam

Simulating GPS in Gazebo

In this section, we will see how to simulate a GPS sensor in Gazebo. As you know, GPS is one of the essential sensors in a self-driving car. You can start a GPS simulation using the following command:

```
| $ roslaunch sensor_sim_gazebo gps.launch
```

Now, you can list out the topic and find the GPS topics published from the Gazebo plugin. Here is a list of topics from the GPS plugin:

```
robot@robot-pc:~$ rostopic list
/clock
/gazebo/link_states
/gazebo/model_states
/gazebo/parameter_descriptions
/gazebo/parameter_updates
/gazebo/set_link_state
/gazebo/set_model_state
/gps/fix
/gps/fix/position/parameter_descriptions
/gps/fix/position/parameter_updates
/gps/fix/status/parameter_descriptions
/gps/fix/status/parameter_updates
/gps/fix/velocity/parameter_descriptions
/gps/fix/velocity/parameter_updates
/gps/fix_velocity
/joint_states
/rosout
/rosout_agg
/tf
/tf_static
```

Figure 20: List of topics from the Gazebo GPS plugin

You can echo the `/gps/fix` topic to confirm that the plugin is publishing the values correctly.

You can use the following command to echo this topic:

```
| $ rostopic echo /gps/fix
```

```
robot@robot-pc:~$ rostopic echo /gps/fix
header:
  seq: 161
  stamp:
    secs: 40
    nsecs: 500000000
  frame_id: sensor
status:
  status: 0
  service: 0
latitude: -30.0602249716
longitude: -51.17391374
altitude: 9.960587315
position_covariance: [0.002501000000000006, 0.0, 0.0, 0.0, 0.00250100000006, 0.0, 0.0, 0.0, 0.002501000000000006]
position_covariance_type: 2
---
```

Figure 21: Values published to the `/gps/fix` topic

If you look at the code in `sensor_sim_gazebo/urdf/gps.xacro`, you will find `<plugin name="gazebo_ros_gps" filename="libhector_gazebo_ros_gps.so">`; these plugins belong to the `hector_gazebo_ros_plugins` package, which we installed at the beginning of the sensor interfacing. We can set all parameters related to GPS in this plugin

description, and you can see the test parameters values in the `gps.xacro` file. The GPS model is visualized as a box, and you can test the sensor values by moving this box in Gazebo.

Interfacing GPS with ROS

In this section, we will see how to interface some popular GPS modules with ROS. One of the popular GPS modules we discussed earlier was **Oxford Technical Solutions (OxTS)**. You can find GPS/IMU modules at <http://www.oxts.com/products/>. The ROS interface of this module can be found at http://wiki.ros.org/oxford_gps_eth. The Applanix GPS/IMU ROS module driver can be found at the following links:

http://wiki.ros.org/applanix_driver

<http://wiki.ros.org/applanix>

Simulating IMU on Gazebo

Similar to GPS, we can start the IMU simulation using the following command:

```
| $ roslaunch sensor_sim_gazebo imu.launch
```

You will get orientation values, linear acceleration, and angular velocity from this plugin. After launching this file, you can list out the topics published by the `imu` plugin. Here is the list of topics published by this plugin:

```
robot@robot-pc:~$ rostopic list
/clock
/gazebo/link_states
/gazebo/model_states
/gazebo/parameter_descriptions
/gazebo/parameter_updates
/gazebo/set_link_state
/gazebo/set_model_state
 imu
/joint_states
/rosout
/rosout_agg
/tf
/tf_static
```

Figure 22: List of topics published from the `imu` ROS plugin

We can check out the `/imu` topic by echoing the topic. You can find orientation, linear acceleration, and angular velocity data from this topic. The values are shown here:

```
robot@robot-pc:~$ rostopic echo /imu
header:
  seq: 0
  stamp:
    secs: 24
    nsecs: 95000000
  frame_id: sensor
orientation:
  x: -9.88131291682e-324
  y: -9.88131291682e-324
  z: 8.87671670196e-17
  w: 1.0
orientation_covariance: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
angular_velocity:
  x: 3.95252516673e-321
  y: 3.95252516673e-321
  z: 0.0
angular_velocity_covariance: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
linear_acceleration:
  x: -1.95719626798e-20
  y: 8.93613280022e-20
  z: 7.28456264068e-12
```

Figure 23: Data from the `/imu` topic

If you look at the IMU plugin definition code from `sensor_sim_gazebo/urdf imu.xacro`, you can find the name of the plugin and its parameters.

The name of the plugin is mentioned in the following code snippet:

```
<gazebo>
  <plugin name="imu_plugin" filename="libgazebo_ros_imu.so">
    <alwaysOn>true</alwaysOn>
```

```

<bodyName>sensor</bodyName>
<topicName>imu</topicName>
<serviceName>imu_service</serviceName>
<gaussianNoise>0.0</gaussianNoise>
<updateRate>20.0</updateRate>
</plugin>
</gazebo>

```

The plugin's name is `libgazebo_ros_imu.so`, and it is installed along with a standard ROS installation.

You can also visualize IMU data in Rviz. Choose the Imu display type to view it. The IMU is visualized as a box itself, so if you move the box in Gazebo, you can see an arrow moving in the direction of movement. The Gazebo and Rviz visualizations are shown here:

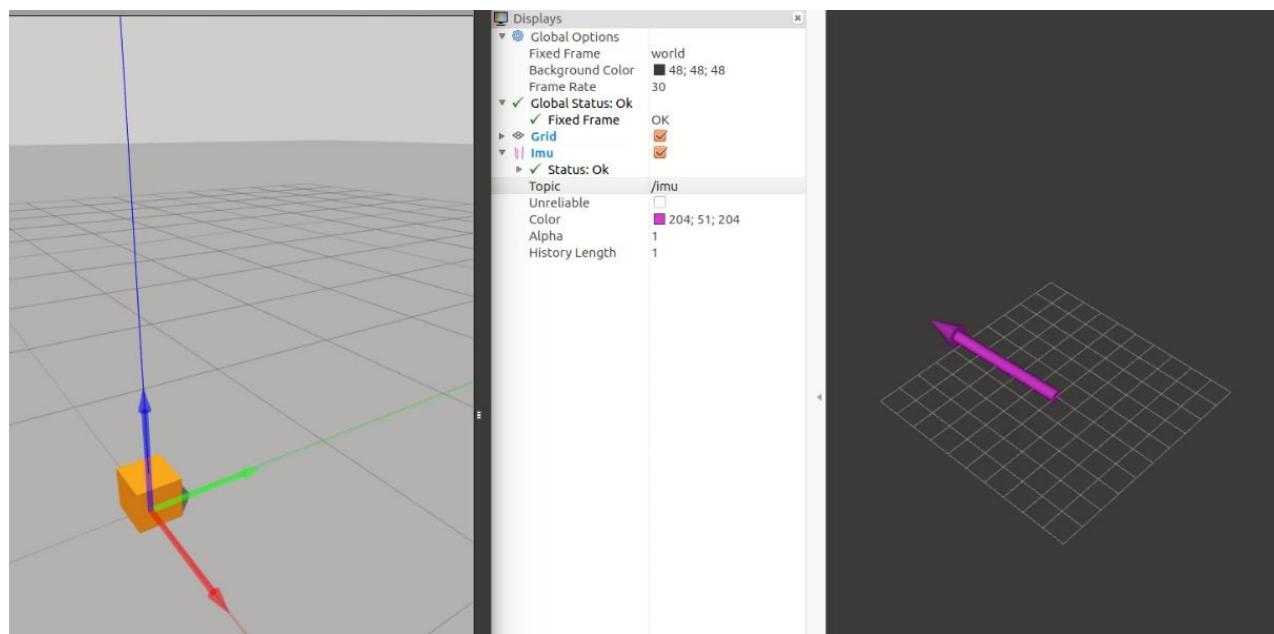


Figure 24: Visualization of the `/imu` topic

Interfacing IMUs with ROS

Most self-driving cars use integrated modules for GPS, IMU, and wheel encoders for accurate position prediction. In this section, we will look at some popular IMU modules that you can use if you want to use IMU alone.

I'll point you to a few links for ROS drivers used to interface with it. One of the popular IMUs is the MicroStrain 3DM-GX2 (<http://www.microstrain.com/inertial/3dm-gx2>):



Source: <http://www.microstrain.com>

Figure 25: Microstrain-3DM-GX2 IMU

Here are the ROS drivers for this IMU series:

http://wiki.ros.org/microstrain_3dmgx2_imu

http://wiki.ros.org/microstrain_3dm_gx3_45

Other than that, there are IMUs from Phidget (http://wiki.ros.org/phidgets_imu) and popular IMUs such as InvenSense MPU 9250, 9150, and 6050 models (https://github.com/jeskesen/i2c_imu). Another IMU sensor series called MTi from Xsens and its drivers can be found at http://wiki.ros.org/xsens_driver.

Simulating an ultrasonic sensor in Gazebo

Ultrasonic sensors also play a key role in self-driving cars. We've already seen that range sensors are widely used in parking assistant systems. In this section, we are going to see how to simulate a range sensor in Gazebo. The range sensor Gazebo plugin is already available in the `hector` Gazebo ROS plugin, so we can just use it in our code.

Like we did in earlier demos, we will first see how to run the simulation and watch the output.

The following command will launch the range sensor simulation in Gazebo:

```
| $ roslaunch sensor_sim_gazebo sonar.launch
```

In this simulation, we are taking the actual 3D model of the sonar, and it's very small. You may need to zoom in Gazebo to view the model. We can test the sensor by putting an obstacle in front of it. We can start Rviz and can view the distance using the Range display type. The topic name is `/distance` and the Fixed Frame is world.

Here is the range sensor value when the obstacle is far away:

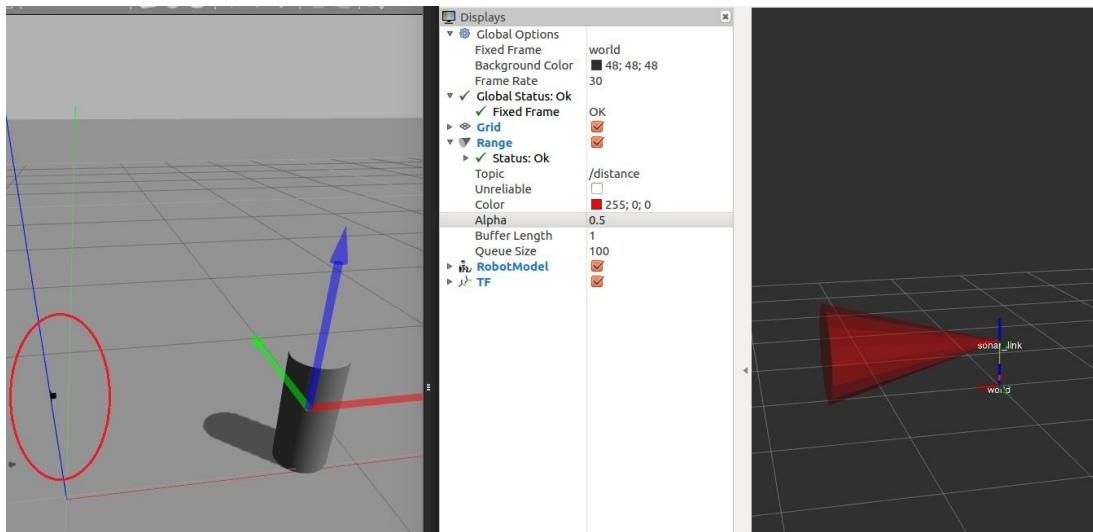


Figure 26: Range sensor value when the obstacle is far away

You can see that the marked point is the ultrasonic sound sensor, and on the right, you can view the Rviz range data as a cone-shaped structure. If we move the obstacle near the sensor, we can see what happens to the range sensor data:

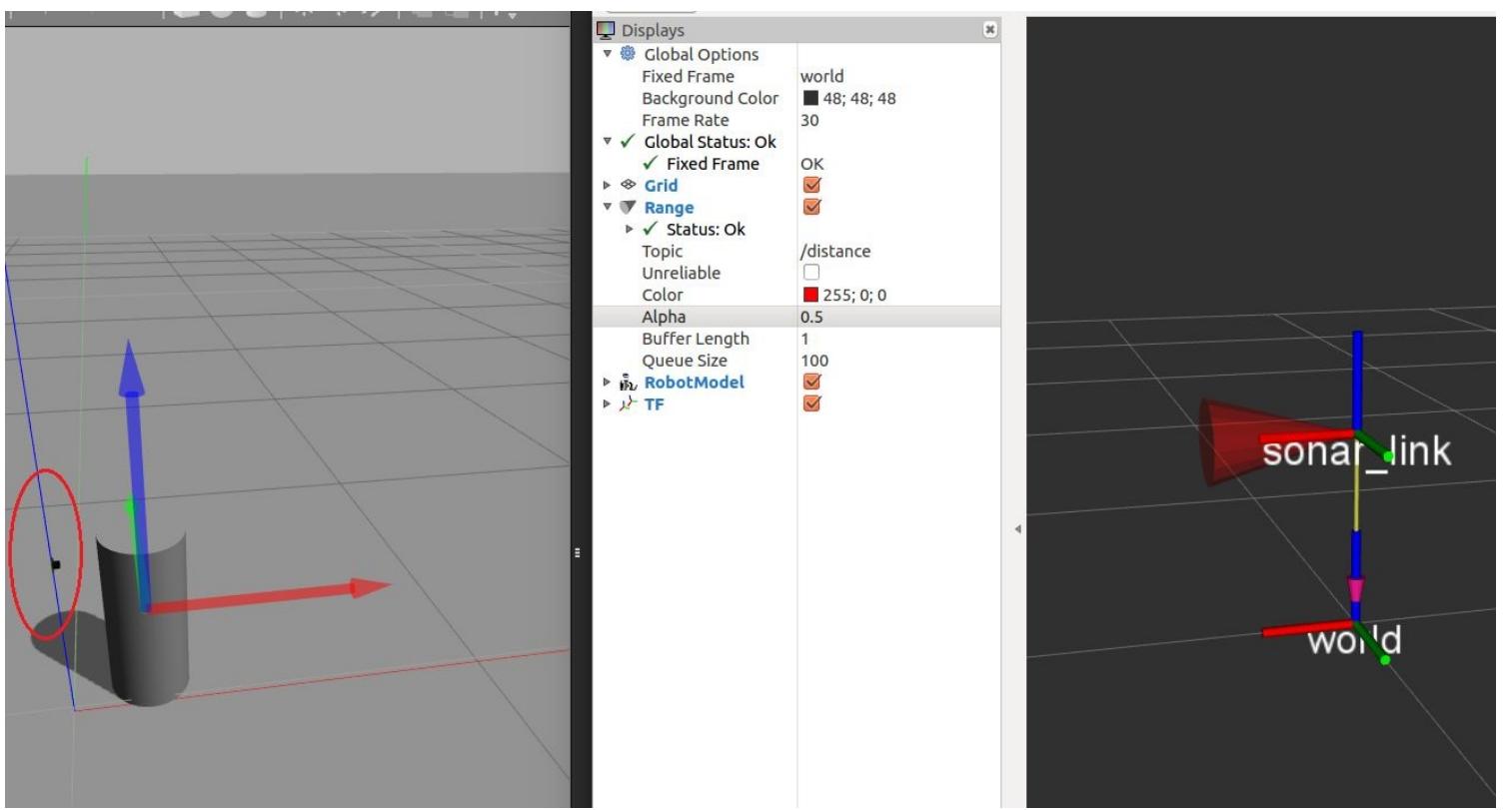


Figure 27: Range sensor value when the obstacle is near

When the obstacle is too near the sensor, the cone size is reduced, which means the distance to the obstacle is very low.

Open the Gazebo sonar plugin definition from `sensor_sim_gazebo/urdf/ sonar.xacro`. This file includes a reference to another file called `sonar_model.xacro`, which has the complete sonar plugin definition.

We are using the `libhector_gazebo_ros_sonar` plugin to run this simulation, which is given in the following code snippet from `sonar_mode.xacro`:

```
<plugin name="gazebo_ros_sonar_controller"
filename="libhector_gazebo_ros_sonar.so">
```

Low-cost LIDAR sensors

This is an add-on section for hobbyists. If you are planning to build a miniature model of a self-driving car, you can use the following LIDAR sensors.

Sweep LIDAR

The Sweep 360-degree rotating LIDAR (<http://scansense.io/>) has a range of 40 meters. Compared to high-end LIDARs such as Velodyne, it is very cheap and good for research and hobby projects:



Figure 28: Sweep LIDAR

There is a good ROS interface available for this sensor. Here's the link to the Sweep sensor ROS package: <https://github.com/scansense/sweep-ros>. Before building the package, you need to install some dependencies:

```
| $ sudo apt-get install ros-kinetic-pcl-conversions ros-kinetic-pointcloud-to-laserscan
```

Now you can simply copy the `sweep-ros` package to your Catkin workspace and build it using the `catkin_make` command.

After building the package, you can plug the LIDAR to your PC through a serial-to-USB converter. If you plug this converter into a PC, Ubuntu will assign a device called `/dev/ttyUSB0`. First, you need to change the permission of the device using the following command:

```
| $ sudo chmod 777 /dev/ttyUSB0
```

After changing the permission, we can start launching any of the launch files to view the laser's `/scan` point cloud data from the sensor.

The launch file will display the laser scan in Rviz:

```
| $ roslaunch sweep_ros view_sweep_laser_scan.launch
```

The launch file will display the point cloud in Rviz:

```
| $ roslaunch sweep_ros view_sweep_pc2.launch
```

Here is the visualization of the Sweep LIDAR:

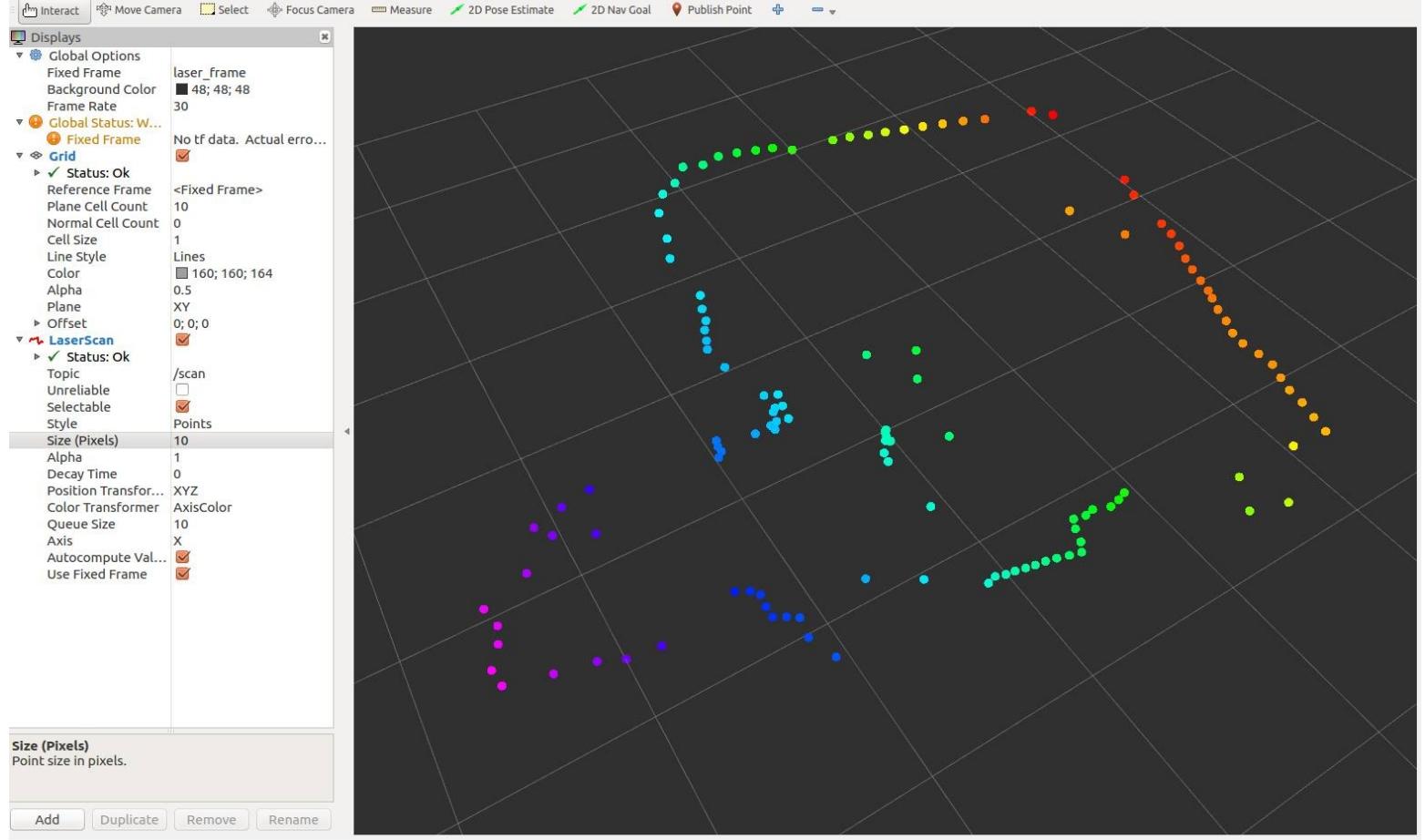


Figure 29: Sweep LIDAR visualization in Rviz

RPLIDAR

Similar to the Sweep LIDAR, RPLIDAR (<http://www.slamtec.com/en/lidar>) is another low-cost LIDAR for hobby projects. RPLIDAR and Sweep have the same applications: SLAM and autonomous navigation:



Figure 30: RPLIDAR

There is a ROS driver for interfacing the RPLIDAR with ROS. The ROS package is at <http://wiki.ros.org/rplidar>. The GitHub link of the package is https://github.com/robopeak/rplidar_ros.

Simulating a self-driving car with sensors in Gazebo

In this section, we are going to discuss an open-source self-driving car project done in Gazebo. In this project, we will learn how to implement a robot car model in Gazebo and how to integrate all sensors into it. Also, we will move the robot around the environment using a keyboard, and finally, we will build a map of the environment using SLAM.

Installing prerequisites

This project is fully compatible with ROS Indigo, but some packages are yet to be released in ROS Kinetic. Let's take a look at the prerequisites for setting up packages in ROS Indigo.

The commands given here will install the ROS Gazebo controller manager:

```
| $ sudo apt-get install ros-indigo-controller-manager  
| $ sudo apt-get install ros-indigo-ros-control ros-indigo-ros-controllers  
| $ sudo apt-get install ros-indigo-gazebo-ros-control
```

After installing this, we can install the Velodyne simulator packages in Indigo using the following command:

```
| $ sudo apt-get install ros-indigo-velodyne
```

This project uses SICK laser scanners, so we have to install the SICK ROS toolbox packages:

```
| $ sudo apt-get install ros-indigo-sicktoolbox ros-indigo-sicktoolbox-wrapper
```

After installing all these dependencies, we can clone the project files into a new ROS workspace. Use these commands:

```
| $ cd ~  
| $ mkdir -p catvehicle_ws/src  
| $ cd catvehicle_ws/src  
| $ catkin_init_workspace
```

We have created a new ROS workspace, and now it's time to clone the project files to the workspace. The following commands will do this:

```
| $ cd ~/catvehicle_ws/src  
| $ git clone https://github.com/sprinkjm/catvehicle.git  
| $ git clone https://github.com/sprinkjm/obstaclestopper.git  
| $ cd ../  
| $ catkin_make
```

If all packages have compiled successfully, you can add the following line to the `.bashrc` file:

```
| $ source ~/catvehicle_ws/devel/setup.bash
```

You can launch the vehicle simulation using the following command:

```
| $ roslaunch catvehicle catvehicle_skidpan.launch
```

This command will only start simulation in the command line.

In another Terminal window, run the following command:

```
| $ gzclient
```

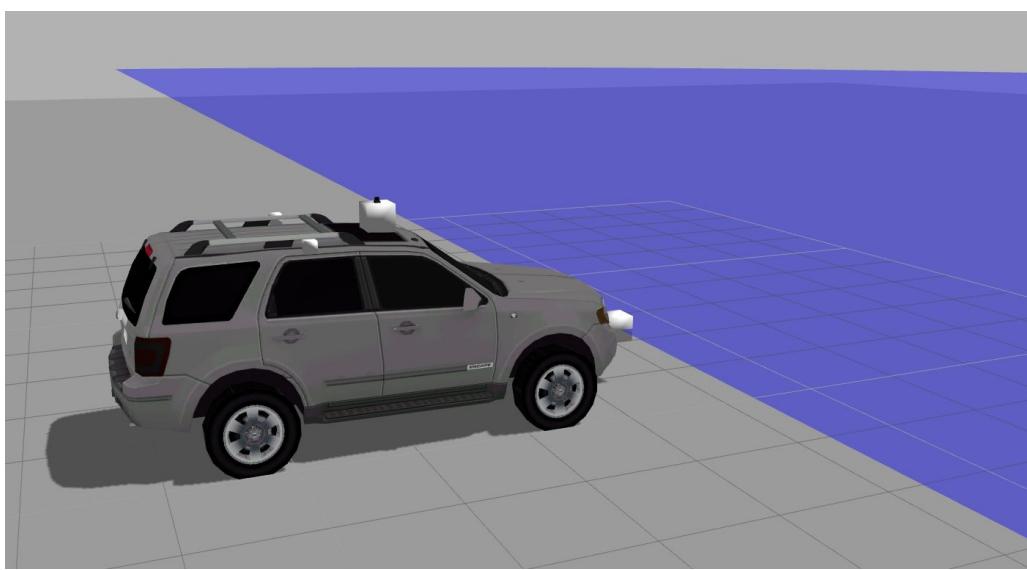


Figure 31: Robot car simulation in Gazebo

You can see the Velodyne scan in front of the vehicle. We can list out all ROS topics from the simulation using the `rostopic` command. Here are the main topics generated in the simulation:

```
/catvehicle/cmd_vel
/catvehicle/cmd_vel_safe
/catvehicle/distanceEstimator/angle
/catvehicle/distanceEstimator/dist
/catvehicle/front_laser_points
/catvehicle/front_left_steering_position_controller/command
/catvehicle/front_right_steering_position_controller/command
/catvehicle/joint1_velocity_controller/command
/catvehicle/joint2_velocity_controller/command
/catvehicle/joint_states
/catvehicle/lidar_points
/catvehicle/odom
/catvehicle/path
/catvehicle/steering
/catvehicle/vel
/clock
/gazebo/link_states
/gazebo/model_states
/gazebo/parameter_descriptions
/gazebo/parameter_updates
/gazebo/set_link_state
/gazebo/set_model_state
/rosout
/rosout_agg
/tf
/tf_static
```

Figure 32: Main topics generated by robotic car simulation

Visualizing robotic car sensor data

We can view each type of sensor data from the robotic car in Rviz. Just run Rviz and open the `catvehicle.rviz` configuration from `chapter_10_codes`. You can see the Velodyne points and robot car model from Rviz, as shown here:

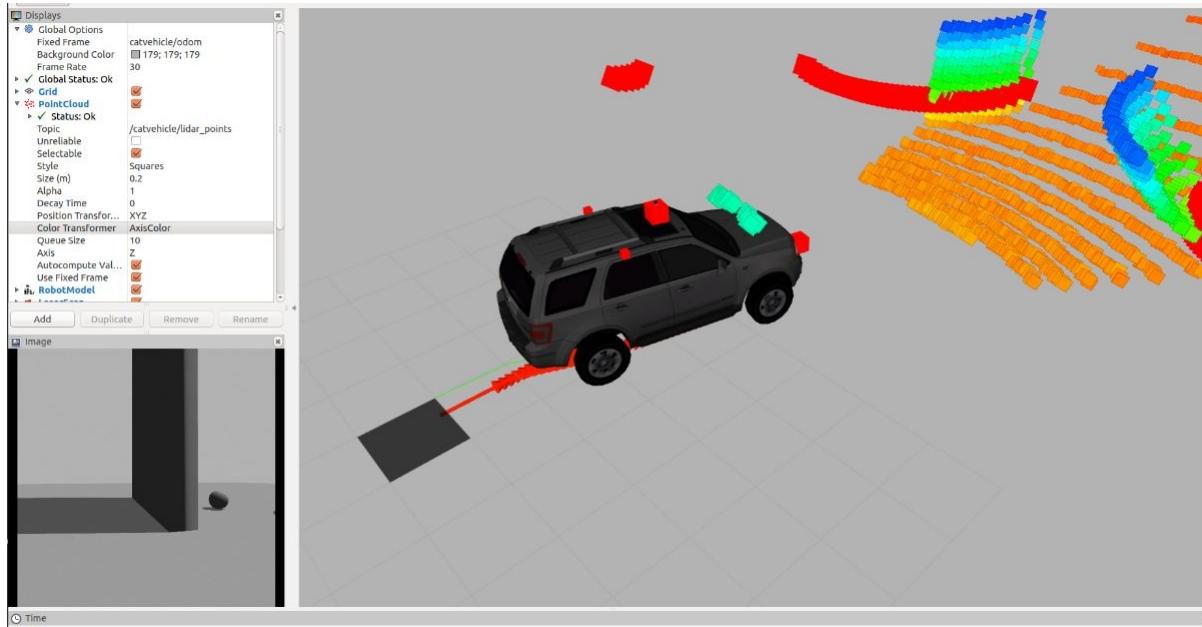


Figure 33: Complete robot car simulation in Rviz

You can also add a camera view in Rviz. There are two cameras, on the left and right side of the vehicle. We have added some obstacles in Gazebo to check whether the sensor is detecting obstacles. You can add more sensors, such as SICK laser and IMU, to Rviz.

Moving a self-driving car in Gazebo

Okay, so we are done with simulating a complete robotic car in Gazebo; now, let's move the robot around the environment. We can do this using a keyboard teleop node.

We can launch an existing TurtleBot teleop node using the following command:

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

The TurtleBot teleop node is publishing Twist messages to `/cmd_vel_mux/input/teleop`, and we need to convert them into `/catvehicle/cmd_vel`.

The following command can do this conversion:

```
| $ rosrun topic_tools relay /cmd_vel_mux/input/teleop /catvehicle/cmd_vel
```

Now, you can move the car around the environment using the keyboard. This will be useful while we perform SLAM.

Running hector SLAM using a robotic car

After moving the robot around the world, let's do some mapping of the world. There are launch files present to start a new world in Gazebo and start mapping. Here is the command to start a new world in Gazebo:

```
| $ roslaunch catvehicle catvehicle_canyonview.launch
```

This will launch the Gazebo simulation in a new world. You can enter the following command to view Gazebo:

```
| $ gzclient
```

The Gazebo simulator with a new world is shown here:

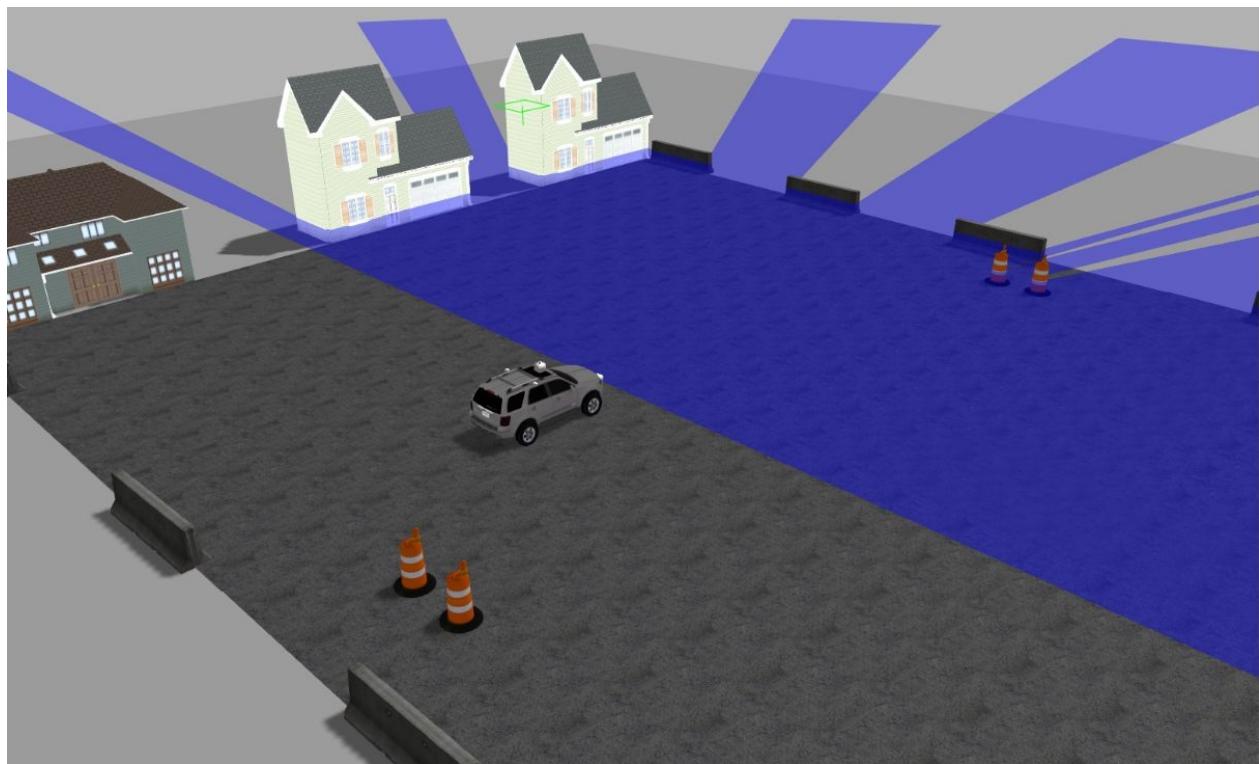


Figure 34: Visualization of a robotic car in an urban environment

You can start the teleoperation node to move the robot, and the following command will start the hector SLAM:

```
| $ roslaunch catvehicle hectorslam.launch
```

To visualize the map generated, you can start Rviz and open the configuration file called `catvehicle.rviz`.

You will get the following kind of visualization in Rviz:

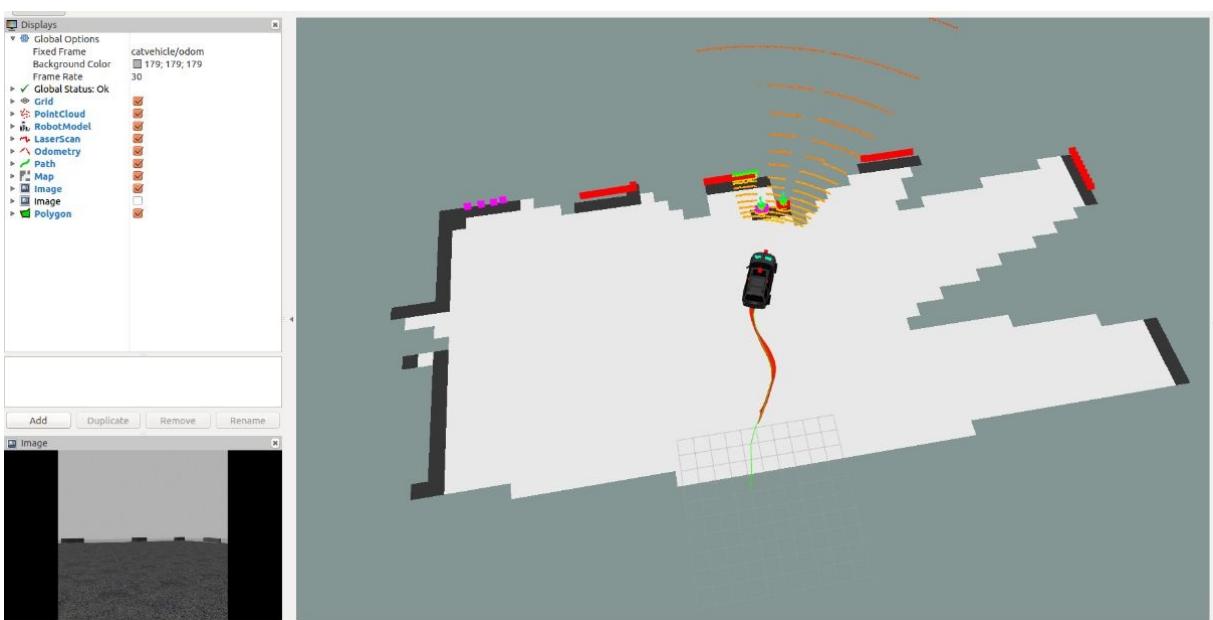


Figure 35: Visualization of a map in Rviz using a robotic car

After completing the mapping process, we can save the map using the following command:

```
| $ rosrun map_server map_saver -f map_name
```

The preceding command will save the current map as two files, called `map_name.pgm` and `map_name.yaml`.



For more details of this project, you can check the following link: <http://cps-vo.org/group/CATVehicleTestbed>

Interfacing a DBW car with ROS

In this section, we will see how to interface a real car with ROS and make it autonomous. As we discussed earlier, the DBW interface enables us to control a vehicle's throttle, brake, and steering using the CAN protocol.

There's an existing open source project that is doing this job. The project is owned by a company called Dataspeed Inc. (<http://dataspeedinc.com/>). Here is the list of projects related to self-driving cars from Dataspeed:

<https://bitbucket.org/DataspeedInc/>

We are going to discuss Dataspeed's ADAS vehicle development project.

First, we will see how to install the ROS packages of this project and look at the functionality of each package and node.

Installing packages

Here are the complete instructions to install these packages. We only need a single command to install all these packages.

We can install this on ROS Indigo and ROS Kinetic using the following command:

```
bash <(wget -q -O -  
https://bitbucket.org/DataspeedInc/dbw_mkz_ros/raw/default/dbw_mkz/scripts/ros_install.bash)
```

You will get other methods of installation from the following link:

http://wiki.ros.org/dbw_mkz

Visualizing the self-driving car and sensor data

The previous packages help you interface a DBW car with ROS. If we don't have a real car, we can work with ROS bag files, visualize data, and process it offline.

The following command helps you visualize the URDF model of a self-driving car:

```
| $ roslaunch dbw_mkz_description rviz.launch
```

You will get following model when you execute it:

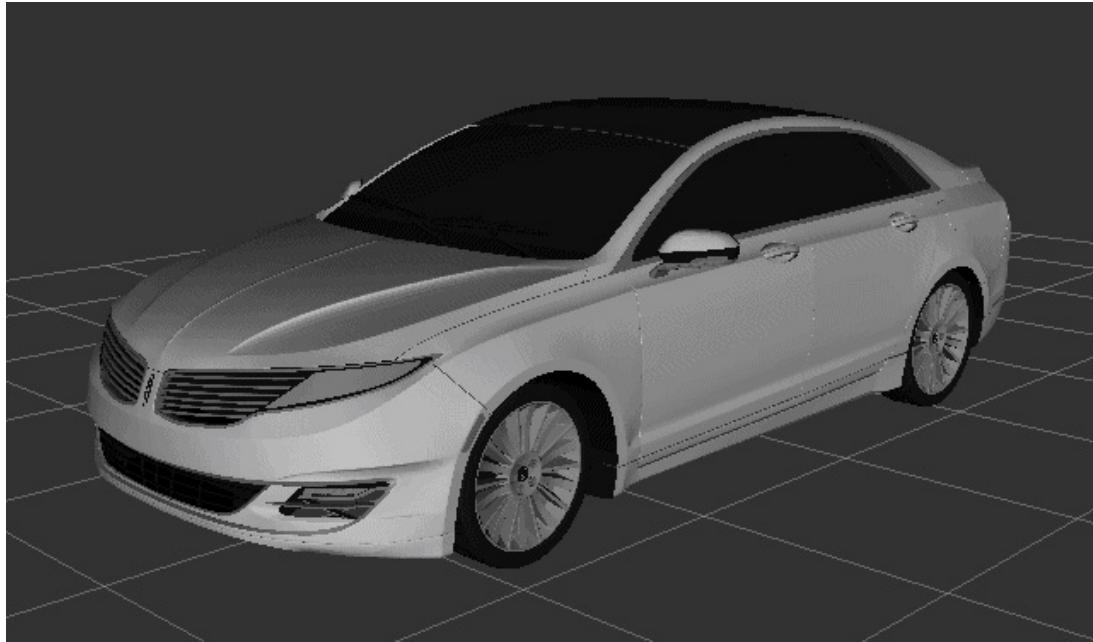


Figure 36: Visualization of a self-driving car

If we want to visualize the Velodyne sensor data, other sensors such as GPS and IMU, and control signal such as steering commands, brake, and acceleration, you can use the following commands:

Use this command to download the ROS bag file:

```
| $ wget https://bitbucket.org/DataspeedInc/dbw_mkz_ros/downloads/mkz_20151207_extra.bag.tar.gz
```

You will get a compressed file from the preceding command; extract it to your home folder.

Now you can run the following command to read data from the bag file:

```
| $ roslaunch dbw_mkz_can offline.launch
```

The following command will visualize the car model:

```
| $ roslaunch dbw_mkz_description rviz.launch
```

And finally, we have to run the bag file:

```
| $ rosbag play mkz_20151207.bag -clock
```

To view the sensor data in Rviz, we have to publish a static transform:

```
| $ rosrun tf static_transform_publisher 0.94 0 1.5 0.07 -0.02 0 base_footprint velodyne 50
```

This is the result:

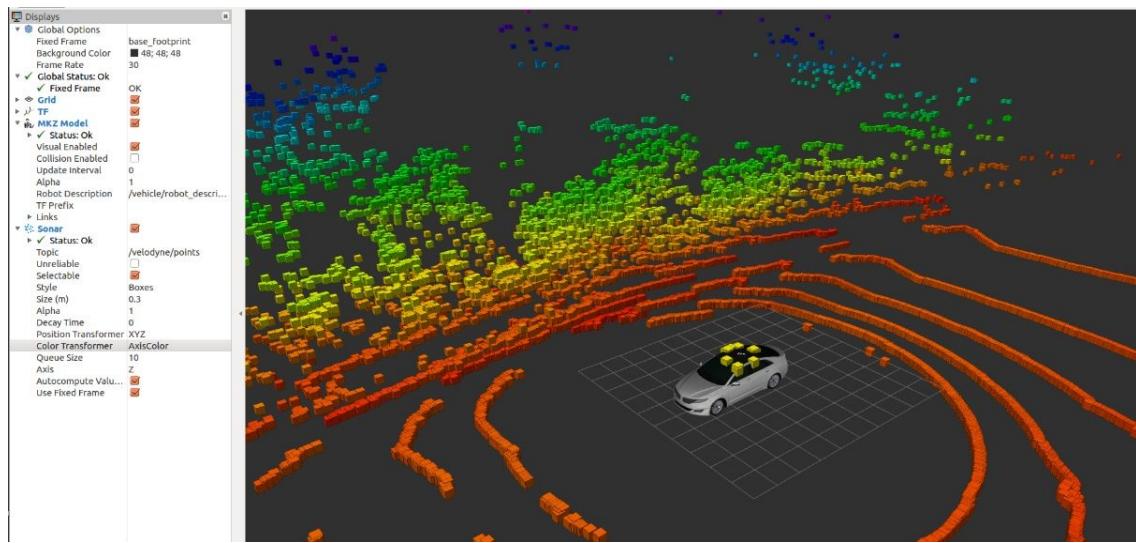


Figure 37: Visualization of a self-driving car

You can set Fixed Frame as the `base_footprint` and view the car model and Velodyne data.

The following commands can help to communicate using ROS with DBW-based cars.

This is the command to do so:

```
| $ roslaunch dbw_mkz_can dbw.launch
```

Now you can test the car using a joystick. Here is the command to launch its nodes:

```
| $ roslaunch dbw_mkz_joystick_demo joystick_demo.launch sys:=true
```

Data provided by *Dataspeed Inc*, located in *Rochester Hills, Michigan*. For more information please visit <http://dataspeedinc.com>.

Communicating with DBW from ROS

In this section, we will see how we can communicate from ROS with DBW-based cars.

This is the command to do so:

```
| $ roslaunch dbw_mkz_can dbw.launch
```

Now you can test the car using a joystick. Here is the command to launch its nodes:

```
| $ roslaunch dbw_mkz_joystick_demo joystick_demo.launch sys:=true
```

Introducing the Udacity open source self-driving car project

There is another open source self-driving car project by Udacity (<https://github.com/udacity/self-driving-car>) that was created for teaching their Nanodegree self-driving car program. The aim of this project is to create a complete autonomous self-driving car using deep learning and using ROS as middleware for communication. The project is split into a series of challenges, and anyone can contribute to the project and win a prize. The project is trying to train a **convolution neural network (CNN)** from a vehicle camera dataset to predict steering angles. This approach is a replication of end-to-end deep learning from NVIDIA (<https://devblogs.nvidia.com/parallelforall/deep-learning-self-driving-cars/>), used in their self-driving car project called DAVE-2.

The following is the block diagram of DAVE-2. DAVE-2 stands for DARPA Autonomous Vehicle-2, which is inspired by the DAVE project by DARPA.

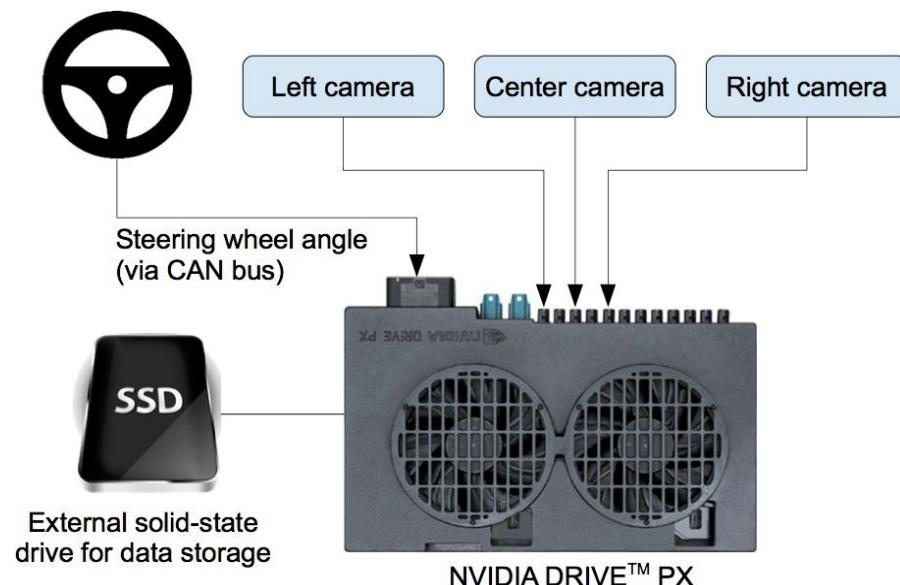


Figure 38: DAVE-2 block diagram

This system basically consists of three cameras and an NVIDIA supercomputer called NVIDIA PX. This computer can train images from this camera and predict the steering angle of the car. The steering angle is fed to the CAN bus and controls the car.

The following are the sensors and components used in the Udacity self-driving car:

- **2016 Lincoln MKZ:** This is the car that is going to be made autonomous. In the previous section, we saw the ROS interfacing of this car. We are using that project here too.
- **Two Velodyne VLP-16 LiDARs**
- **Delphi radar**
- **Point Grey Blackfly cameras**
- **Xsens IMU**

- **Engine control unit (ECU)**

This project uses the `dbw_mkz_ros` package to communicate from ROS to the Lincoln MKZ. In the previous section, we set up and worked with the `dbw_mkz_ros` package. Here is the link to obtain a dataset for training the steering model: <https://github.com/udacity/self-driving-car/tree/master/datasets>. You will get a ROS launch file from this link to play with these bag files too.

Here is the link to get an already trained model that can only be used for research purposes: <https://github.com/udacity/self-driving-car/tree/master/steering-models>. There is a ROS node for sending steering commands from the trained model to the Lincoln MKZ. Here, `dbw_mkz_ros` packages act as an intermediate layer between the trained model commands and the actual car.

Open source self-driving car simulator from Udacity

Udacity also provides an open source simulator for training and testing self-driving deep-learning algorithms. The simulator project is available at <https://github.com/udacity/self-driving-car-sim>. You can also download the precompiled version of a simulator for Linux, Windows, and Mac from the same link.

Here are the screenshots of this simulator. We can discuss the working of the simulator along with the screenshots.



Figure 39: Udacity self-driving car simulator

You can see two options in the simulator; the first is for training and the second is for testing autonomous algorithms. We can also select the Track in which we have to drive the vehicle. When you click on the Training Mode button, you will get a racing car on the selected track. You can move the car using the WASD key combination, like a game. Here is a screenshot of the training mode.



Figure 40: Udacity self-driving car simulator in training mode

You can see a RECORD button in the top-right corner, which is used to capture the front camera images of the car. We can browse to a location, and those captured images will be stored in that location.

After capturing the images, we have to train the car using deep-learning algorithms to predict steering angle, acceleration, and braking. We are not discussing the code, but I'll provide a reference for you to write it. The complete code reference to implement the driving model using deep learning and the entire explanation for it are at <https://github.com/thomasantony/sdc-live-trainer>. The `live_trainer.py` code helps us train the model from captured images.

After training the model, we can run `hybrid_driver.py` for autonomous driving. For this mode, we need to select autonomous mode in the simulator and execute the `hybrid_driver.py` code.



Figure 41: Udacity self-driving car simulator in autonomous mode

You can see the car moving autonomously and manually override the steering control at any time.

This simulator can be used to test the accuracy of the deep learning algorithm we are going to use in a real self-driving car.

MATLAB ADAS toolbox

MATLAB also providing toolbox for working with ADAS and autonomous system. You can design, simulate, and test ADAS and autonomous driving systems using this toolbox. Here is the link to check the new toolbox.

<https://in.mathworks.com/products/automated-driving.html>

Questions

- What is the level of autonomy of a self-driving car?
- What are the different levels of autonomy?
- What are the important block diagrams of a self-driving car?
- List down five important sensors used in a self-driving car.

Summary

This chapter was a deep discussion of self-driving cars and their implementation. The chapter started by discussing the basics of self-driving car technology and its history. Afterward, we discussed the core blocks of a typical self-driving car. We also discussed the concept of autonomy levels in self-driving cars. Then, we took a look at different sensors and components commonly used in a self-driving car. We discussed how to simulate such a car in Gazebo and interfacing it with ROS. After discussing all sensors, we saw an open-source self-driving car project that incorporates all sensors and simulated the car model itself in Gazebo. We visualized its sensor data and moved the robot using a teleoperation node. We also mapped the environment using hector SLAM. The next project was from Dataspeed Inc., in which we saw how to interface a real DBW-compatible vehicle with ROS. We visualized the offline data of the vehicle using Rviz. Finally, we took a look at the Udacity self-driving car project and its simulator.

In the next chapter, we will see how to teleoperate a robot using the VR headset and Leap motion.

Teleoperating a Robot Using a VR Headset and Leap Motion

The term **virtual reality** is gaining popularity nowadays, even though it started long ago. The concept of virtual reality began in the 1950s as science fiction, but it took 60 years to become more popular and acceptable. Why is it more popular now? The answer is the availability of cheap computing. Before, a virtual reality headset was very expensive. Now, we can build one for \$5. You may have heard about Google Cardboard, which is the cheapest virtual reality headset available currently, and there are many upcoming models based on it. Now we only need a good smartphone and cheap **virtual reality (VR)** headset to get the virtual reality experience. There are also high-end VR headsets such as the Oculus Rift and HTC Vive that have a high frame rate and response.

In this chapter, we will discuss a ROS project in which we can control a robot using a Leap Motion sensor and experience the robot environment using a virtual reality headset. We will demonstrate this project using a TurtleBot simulation in Gazebo and control the robot using Leap Motion. To visualize the robot environment, we will use a cheap VR headset along with an Android smartphone.

Here are the main topics we will discuss in this chapter:

- Getting started with a VR headset and Leap Motion
- Project prerequisites
- Design and working of the project
- Installing the Leap Motion SDK on Ubuntu
- Playing with the Leap Motion visualizer tool
- Installing ROS packages for Leap Motion
- Visualizing Leap Motion data in Rviz
- Creating a teleoperation node for Leap Motion
- Building and installing the ROS-VR Android application
- Working with the ROS-VR application and interfacing with Gazebo
- Working with the TurtleBot simulation in VR
- Troubleshooting the ROS-VR application
- Integrating the ROS-VR application and Leap Motion teleoperation

Getting started with a VR headset and Leap Motion

This section is for beginners who haven't worked with VR headsets and Leap Motion yet. A (**VR**) headset is a head-mounted display in which we can either put a smartphone or that has an inbuilt display that can be connected to HDMI or some other display port. A VR headset can create a virtual 3D environment by mimicking human vision, that is, stereo vision. Human vision works like this: we have two eyes and get two separate and slightly different images in each eye. The brain then combines these two images and generates a 3D image of the surroundings. Similarly, VR headsets have two lenses and a display. The display can be inbuilt or a smartphone. This screen will show a separate view of the left and right image, and when we put the smartphone or inbuilt display into the headset, it will focus and reshape using two lenses and will simulate 3D stereoscopic vision. In effect, we can explore a 3D world inside this headset. Rather than just visualizing the world, we can also control the event in the 3D world and hear sound too. Cool, right?

Here is the internal structure of a Google Cardboard VR headset:

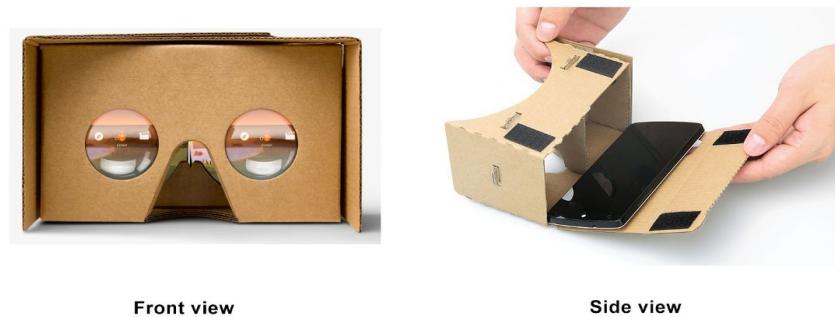


Figure 1: Google Cardboard VR headset

There is a variety of models of VR headsets available in addition to the high-end models such as Oculus Rift, HTC Vive, and so on. The following is one of the VR headsets, which we will use in this chapter. It works based on the same principle of Google Cardboard, but instead of cardboard, it uses a plastic body:



Figure 2: VR-SHINECON headset

You can test the VR feature by downloading Android VR applications from Google Play Store.



You can search for `cardboard` in Google Play Store to get the Google VR application. You can use it for testing VR on your smartphone.

The next device we are using in this project is the Leap Motion controller (<https://www.leapmotion.com>). The Leap Motion controller is basically an input device like a PC mouse in which we can control everything using hand gestures. The Leap can accurately track the hands of a user and map the position and orientation of each finger joint accurately. It has two IR cameras and several IR projectors facing upward. The user can position their hand above the device and move their hand. The position and orientation of hands and fingers can be accurately retrieved from their SDK.

Here is the Leap Motion controller and how we can interact with it:



Figure 3: Interacting with the Leap Motion controller

Project prerequisites

So let's start discussing the project. The following are the software and hardware prerequisites of this project:

| No | Component/software | Link |
|----|------------------------|---|
| 1 | Low-cost VR headset | https://vr.google.com/cardboard/get-cardboard/ |
| 2 | Leap Motion controller | https://www.leapmotion.com/ |
| 3 | Wi-Fi router | Any router can connect to a PC or Android phone |
| 4 | Ubuntu 14.04.5 LTS | http://releases.ubuntu.com/14.04/ |
| 5 | ROS Indigo | http://wiki.ros.org/indigo/Installation/Ubuntu |
| 6 | Leap Motion SDK | https://www.leapmotion.com/setup/linux |

This project has been tested on ROS Indigo, and the code is compatible with ROS Kinetic too, but the Leap Motion SDK is still in development for Ubuntu 16.04 LTS. So here the code is tested using Ubuntu 14.04.5 and ROS Indigo. If you are ready with the components, let's look at the design of the project and how it works.

Design and working of the project

This project can be divided into two sections: teleoperation using **Leap Motion** and streaming images to an Android phone to get a VR experience inside a VR headset. Before going to discuss each design aspect, let's see how we have to interconnect these devices.

The following figure shows how the components are interconnected for this project:

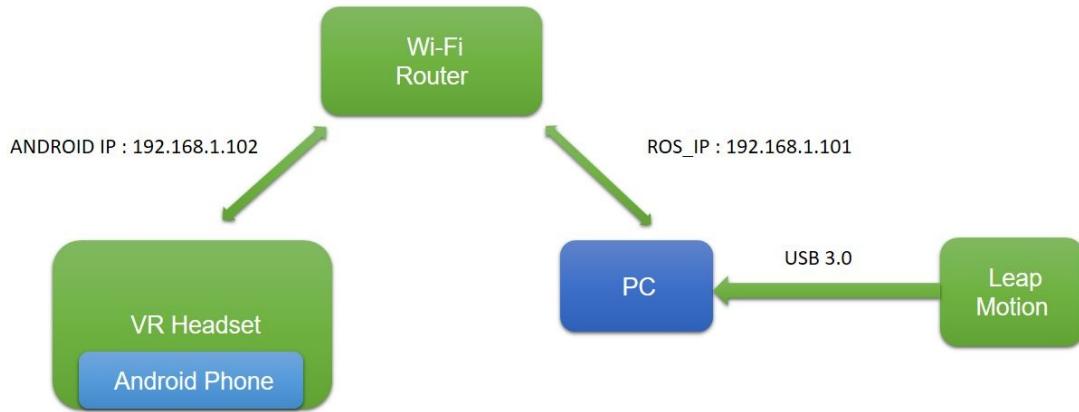


Figure 4: Hardware components and connection

You can see that each device (that is, PC and Android phone) is connected to a Wi-Fi router, and the router has assigned an IP to each device. Each device communicates using these IP addresses. You will see the importance of these addresses in the upcoming sections.

Next, we will see how we can teleoperate a robot in ROS using Leap Motion. We will be controlling it while wearing the VR headset. So, we don't need to press any buttons to move the robot; rather, we can just move it with our hands.

The basic operation involved here is converting the Leap Motion data into ROS Twist messages. Here, we are only interested in reading the orientation of the hand. We are taking roll, pitch, and yaw and mapping them into ROS Twist messages. Here is how:

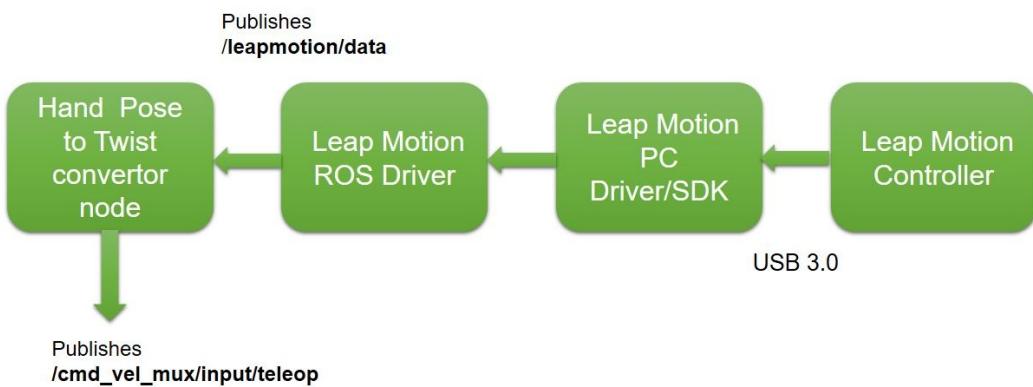


Figure 5: Leap Motion data to ROS command velocity

The preceding figure shows how Leap Motion data is manipulated into ROS Twist messages. The **Leap Motion PC Driver/SDK** interfaces the controller with Ubuntu, and the **Leap Motion ROS Driver**, which works on top of this driver/SDK, fetches the hand and finger position and publishes it as ROS topics. The node we are going to write can convert the hand position to Twist data, which will subscribe to the Leap Motion data topic called `/leapmotion/data`, convert it into corresponding command velocities, and publish to the topic called `/cmd_vel_mux/input/teleop`. The conversion algorithm is based on comparing the hand orientation value. If the value is in a particular range, we will publish a particular Twist value.

Here is the simple algorithm that converts Leap Motion orientation data into Twist messages:

1. Take the orientation values of hand, such as yaw, pitch, and roll, from the Leap Motion ROS driver.
2. The roll movement of the hand corresponds to robot rotation. If the hand rotates anticlockwise, then the robot will be triggered to rotate anticlockwise by sending a command velocity. This will be the opposite case of a roll of the hand in the clockwise direction.
3. If the hand is pitched down, the robot will move forward, and if the hand is pitched up, the robot will move backward.
4. If there is no hand movement, the robot will stop.

This is a simple algorithm to move a robot using Leap Motion. Okay, let's start with setting up a Leap Motion controller in Ubuntu and working with its ROS interface.

Installing the Leap Motion SDK on Ubuntu 14.04.5

In this project, we have chosen Ubuntu 14.04.5 LTS and ROS Indigo because the Leap Motion SDK will smoothly work with this combination. The Leap Motion SDK is not fully supported by Ubuntu 16.04 LTS; if there are any further fixes from the company, this code will work on Ubuntu 16.04 LTS with ROS Kinetic.

The Leap Motion SDK is the core of the Leap Motion controller. The Leap Motion controller has two IR cameras facing upwards and also has several IR projectors. This is interfaced with a PC, and the Leap SDK runs on the PC, which has drivers for the controller. It also has algorithms to process the hand image to produce the joint values of each finger joint.

Here is the procedure to install the Leap Motion SDK in Ubuntu:

1. Download the SDK from <https://www.leapmotion.com/setup/linux>; you can extract this package and you will find two DEB files that can be installed on Ubuntu.
2. Open Terminal on the extracted location and install the DEB file using the following command (for 64-bit PCs):

```
| $ sudo dpkg -install Leap-*-x64.deb
```

If you are installing it on a 32-bit PC, you can use the following command:

```
| $ sudo dpkg -install Leap-*-x86.deb
```

3. If you can install this package without any errors, then you are done with installing the Leap Motion SDK and driver.



There are more detailed installation and debugging tips are given on the following website: <https://support.leapmotion.com/hc/en-us/articles/223782608-Linux-Installation>

Visualizing Leap Motion controller data

If you successfully installed the Leap Motion driver/SDK, we can start the device by following these steps:

1. Plug the Leap Motion controller into a USB port; you can plug it into USB 3.0, but 2.0 is fine too.
2. Open Terminal and execute the `dmesg` command to verify that the device is properly detected on Ubuntu:

```
| $ dmesg
```

3. It may give you the following result if it's detected properly.

```
[10010.420978] usb 2-1.2: new high-speed USB device number 8 using ehci-pci
[10010.513671] usb 2-1.2: New USB device found, idVendor=f182, idProduct=0003
[10010.513682] usb 2-1.2: New USB device strings: Mfr=1, Product=2, SerialNumber=0
[10010.513688] usb 2-1.2: Product: Leap Dev Kit
[10010.513692] usb 2-1.2: Manufacturer: Leap Motion
[10010.514270] uvcvideo: Found UVC 1.00 device Leap Dev Kit (f182:0003)
lentin@lentin-Aspire-4755:~$ █
```

Figure 6: Kernel message when plugging in Leap Motion

If you are getting this message, you're ready to start the Leap Motion controller manager.

Playing with the Leap Motion visualizer tool

You can invoke the Leap Motion controller manager by executing the following command:

```
| $ sudo LeapControlPanel
```

If you want to start just the driver, you can use the following command:

```
| $ sudo leapd
```

Use this command to restart the driver:

```
| $ sudo service leapd stop
```

If you are running the Leap control panel, you can see an additional menu on the left-hand side of the screen. Select the Diagnostic Visualizer to view the data from Leap Motion:

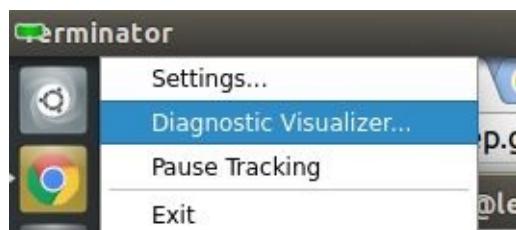


Figure 7: Leap Motion control panel

When you click on this option, a window will pop up in which you can see your hand, and figures get tracked when you put your hand over the device. You can also see the two IR camera views from the device. Here is the screenshot of the Visualizer application.

You can quit the driver from the same drop-down menu, too:

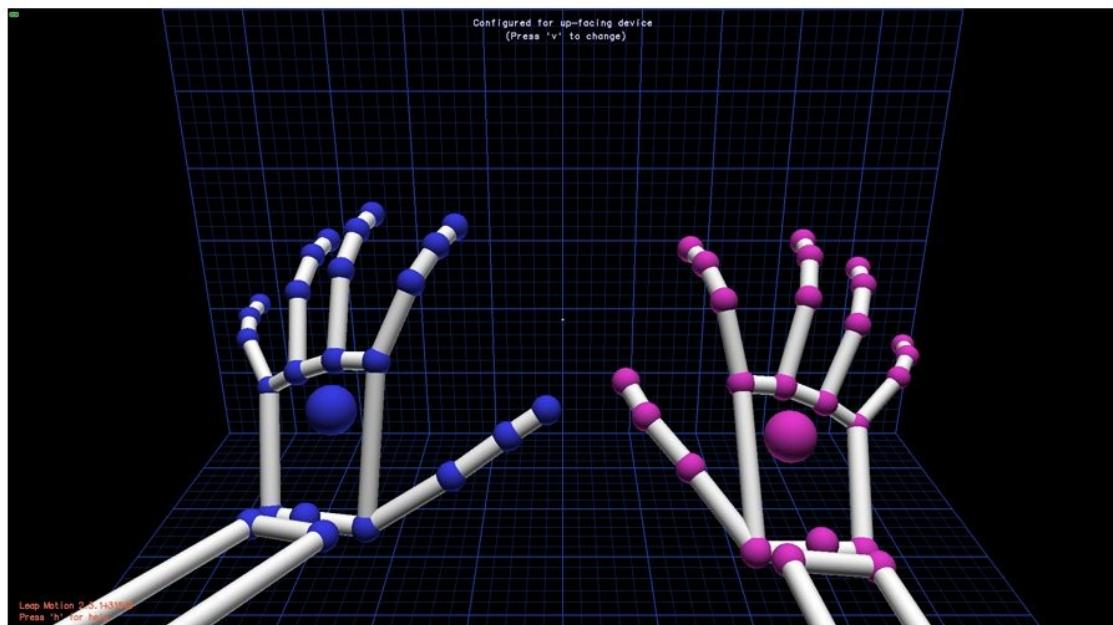


Figure 8: Leap Motion controller Visualizer application

You can interact with the device and visualize the data here. If everything is working well, we can proceed to the next stage: installing ROS driver for the Leap Motion.



You can get more shortcuts to Visualizer from the following link: https://developer.leapmotion.com/documentation/cpp/supplements/Leap_Visualizer.html

Installing the ROS driver for the Leap Motion controller

To interface the Leap Motion with ROS, we will need the ROS driver for it. Here is the link to get the ROS driver for Leap Motion; you can clone it using the command:

```
| $ git clone https://github.com/ros-drivers/leap_motion
```

Before installing the `leap_motion` driver package, we have to do a few things to have it properly compiled.

The first step is to set the path of the Leap Motion SDK in the `.bashrc` file. Assuming that the Leap SDK is in the user's `home` folder with the name `LeapSDK`, we have to set the path variable in `.bashrc` as follows.

```
| $ export LEAP_SDK=$LEAP_SDK:$HOME/LeapSDK
```

This environment variable is needed for compiling the code of the ROS driver, which has Leap SDK APIs.

We also have to add the path of the Python extension of the Leap Motion SDK to `.bashrc`. Here is the command used to do it:

```
| export PYTHONPATH=$PYTHONPATH:$HOME/LeapSDK/lib:$HOME/LeapSDK/lib/x64
```

This will enable Leap Motion SDK APIs in Python. After going through the preceding steps, you can save `.bashrc` and take a new Terminal, so that we will get the preceding variables in the new Terminal.

The final step is to copy the `libLeap.so` file to `/usr/local/lib`. Here is how we do it:

```
| $ sudo cp $LEAP_SDK/lib/x64/libLeap.so /usr/local/lib
```

After copying, execute `ldconfig`:

```
| $ sudo ldconfig
```

Okay, you are finished with setting the environment variables. Now you can compile the `leap_motion` ROS driver package. You can create a ROS workspace or copy the `leap_motion` package to an existing ROS workspace and use `catkin_make`.

You can use the following command to install the `leap_motion` package:

```
| $ catkin_make install --pkg leap_motion
```

This will install the `leap_motion` driver; check whether the ROS workspace path is properly set.

Testing the Leap Motion ROS driver

If everything has been installed properly, we can test it using a few commands.

First, launch the Leap Motion driver or control panel using the following command:

```
| $ sudo LeapControlPanel
```

After launching the command, you can verify that the device is working by opening the Visualizer application. If it's working well, you can launch the ROS driver using the following command:

```
| $ roslaunch leap_motion sensor_sender.launch
```

If it's working properly, you will get topics with this:

```
| $ rostopic list
```

```
lentin@lentin-Aspire-4755:~$ rostopic list
/leapmotion/data
/rosout
/rosout_agg
```

Figure 9: Leap ROS driver topics

If you can see `rostopic/leapmotion/data` in the list, you can confirm that the driver is working. You can just echo the topic and see that the hand and finger values are coming in, as shown in the following screenshot:

```
header:
  seq: 847
  stamp:
    secs: 0
    nsecs: 0
  frame_id: ''
direction:
  x: 0.24784040451
  y: 0.227308988571
  z: -0.941756725311
normal:
  x: 0.0999223664403
  y: -0.972898304462
  z: -0.208529144526
palmpos:
  x: -52.5600471497
  y: 173.553512573
  z: 66.0648040771
ypr:
  x: 25.602668997
  y: 13.5697675013
  z: 132.525765862
```

Figure 10: Data from the Leap ROS driver topic

Visualizing Leap Motion data in Rviz

We can visualize Leap Motion data in Rviz too. There is a ROS package called `leap_client` (https://github.com/qboticslabs/leap_client). You can install this package by setting the following environment variable in `.bashrc`:

```
| export LEAPSDK=$LEAPSDK:$HOME/LeapSDK
```

Note that, when we add new variables in `.bashrc`, you may need to open new Terminal or type `bash` in the existing Terminal.

Now, we can clone the code in a ROS workspace and build the package using `catkin_make`.

Let's play around with this package. To launch nodes, we have to start `LeapControlPanel`:

```
| $ sudo LeapControlPanel
```

Then start the ROS Leap driver launch file:

```
| $ roslaunch leap_motion sensor_sender.launch
```

Now launch the `leap_client` launch file to start the visualization nodes. This node will subscribe to the `leap_motion` driver and convert it into visualization markers in Rviz.

```
| $ roslaunch leap_client leap_client.launch
```

Now, you can open Rviz using the following command and select the `leap_client/launch/leap_client.rviz` configuration file to visualize the markers properly:

```
| $ rosrun rviz rviz
```

If you load the `leap_client.rviz` configuration, you may get hand data like the following (you have to put your hand over the Leap):



Figure 11: Data from the Leap ROS driver topic

Creating a teleoperation node using the Leap Motion controller

In this section, we can see how to create a teleoperation node for a robot using Leap Motion data. The procedure is very simple. We have to create a ROS package for this node. The following is the command to create a new package. You can also find this package from `chapter_11_codes/vr_leap_teleop`.

```
$ catkin_create_pkg vr_leap_teleop roscpp rospy std_msgs visualization_msgs geometry_msgs message_generation visualization_msgs
```

After creation, you can use `catkin_make`. Now, let's create the node to convert Leap Motion data to Twist. You can create a folder called `scripts` inside the `vr_leap_teleop` package. Now you can copy the node called `vr_leap_teleop.py` from the existing package. Let's see how this code works.

We need the following Python modules in this node. Here, we require message definitions from the `leap_motion` package, which is the driver package.

```
import rospy
from leap_motion.msg import leap
from leap_motion.msg import leapros
from geometry_msgs.msg import Twist
```

Now we have to set some range values, in which we have to check whether the current hand value is within range. We are also defining the teleop topic name here.

```
teleop_topic = '/cmd_vel_mux/input/teleop'

low_speed = -0.5
stop_speed = 0
high_speed = 0.5

low_turn = -0.5
stop_turn = 0
high_turn = 0.5

pitch_low_range = -30
pitch_high_range = 30

roll_low_range = -150
roll_high_range = 150
```

Here is the main code of this node. In this code, you can see that the topic from the Leap Motion driver is being subscribed to here. When a topic is received, it will call the `callback_ros()` function:

```
def listener():
    global pub
    rospy.init_node('leap_sub', anonymous=True)
    rospy.Subscriber("leapmotion/data", leapros, callback_ros)
    pub = rospy.Publisher(teleop_topic, Twist, queue_size=1)

    rospy.spin()

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

The following is the definition of the `callback_ros()` function. What it basically does is that it will receive the Leap Motion data and extract the orientation components of the palm only. So we will get yaw, pitch, and roll from this function. We are also creating a `Twist()` message to send the velocity values to the robot.

```
def callback_ros(data):
    global pub

    msg = leapros()
    msg = data

    yaw = msg.ypr.x
    pitch = msg.ypr.y
    roll = msg.ypr.z

    twist = Twist()

    twist.linear.x = 0; twist.linear.y = 0; twist.linear.z = 0
    twist.angular.x = 0; twist.angular.y = 0; twist.angular.z = 0
```

We are performing a basic comparison with the current roll and pitch values again within the following ranges. Here are actions we've assigned for each movement of the robot:

| Hand gesture | Robot movement |
|-------------------------|-----------------------|
| Hand pitch low | Move forward |
| Hand pitch high | Move backward |
| Hand roll anticlockwise | Rotate anticlockwise |
| Hand roll clockwise | Rotate clockwise |

Here is a code snippet taking care of one condition. So in this case, if the pitch is low, then we are providing a high value for linear velocity in the `x` direction for moving forward.

```
if(pitch > pitch_low_range and pitch < pitch_low_range + 30):
    twist.linear.x = high_speed; twist.linear.y = 0;
    twist.linear.z = 0
    twist.angular.x = 0; twist.angular.y = 0; twist.angular.z = 0
```

Okay, so we have built the node, and we can test it at the end of the project. In the next section, we will see how to implement VR in ROS.

Building a ROS-VR Android application

In this section, we will see how to create a virtual reality experience in ROS, especially in robotics simulators such as Gazebo. Luckily, we have an open source Android project called ROS Cardboard (https://github.com/cloudspace/ros_cardboard). This project is exactly what we want for this application. This application is based on ROS-Android APIs, which help us visualize compressed images from a ROS PC. It also does the splitting of the view for the left and right eye, and when we put this on a VR headset, it will feel like 3D.

Here is a figure that shows how this application works:

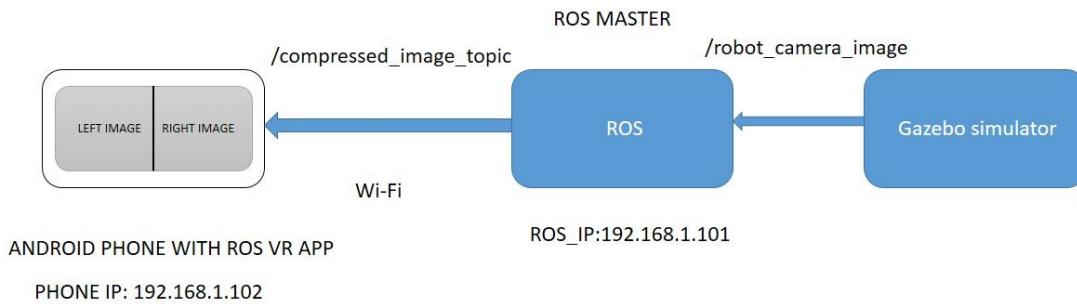


Figure 12: Communication between a ROS PC and Android phone

From the preceding figure, you can see that the image topic from Gazebo can be accessed from a ROS environment, and the compressed version of that image is sent to the ROS-VR app, which will split the view into left and right to provide 3D vision. Setting the `ROS_IP` variable on PC is important for the proper working of the VR application. The communication between PC and phone happens over Wi-Fi, both on same network.

Building this application is not very tough; first, you can clone this app into some folder. You need to have all of the Android development environment and SDK installed. To do so, you can refer to *Chapter 26, ROS on MATLAB and Android*. Just clone it and you can simply build it using the following instructions:

Plug your Android device into Ubuntu and execute the following command to check whether the device is detected on your PC:

```
| $ adb devices
```

The **adb** command, which stands for **Android Debug Bridge**, will help you communicate with an Android device and emulator. If this command lists out the devices, then you are done; otherwise, do a Google search to find out how to make it work. It won't be too difficult.

After getting the device list, clone the ROS Cardboard project using the following command. You can clone into `home` or `desktop`.

```
| $ git clone https://github.com/cloudspace/ros_cardboard.git
```

After cloning, enter the folder and execute the following command to build the entire package and install it on the device:

```
| $ ./gradlew installDebug
```

You may get an error saying the required Android platform is not available; what you need is to simply install it using the Android SDK GUI. If everything works fine, you can able install the APK on an Android device. If you are unable to build the APK, you can also find it in `chapter_11_codes/ros_cardboard`. If installing the APK to the device directly failed, you can find the generated APK from `ros_cardboard/ros_cardboard_module/build/outputs/apk`. You can copy this APK to the device and try to install it. If you have any difficulty installing it, you can use the APK editor app, mentioned in *Chapter 26, ROS on MATLAB and Android*.

Working with the ROS-VR application and interfacing with Gazebo

The new APK will be installed with a name such as `rosserial`; before starting this app, we need to set a few things up on the ROS PC.

The next step is to set the `ROS_IP` variable in the `.bashrc` file. Execute the `ifconfig` command and retrieve the Wi-Fi IP address of the PC, as shown here:



```
wlan0      Link encap:Ethernet HWaddr 94:39:e5:4d:7d:da
          inet addr:192.168.1.101 Bcast:192.168.1.255 Mask:255.255.255.0
            inet6 addr: fe80::9639:e5ff:fe4d:7dda/64 Scope:Link
              UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
              RX packets:1303 errors:0 dropped:0 overruns:0 frame:0
              TX packets:1127 errors:0 dropped:0 overruns:0 carrier:0
              collisions:0 txqueuelen:1000
              RX bytes:1136655 (1.1 MB) TX bytes:243000 (243.0 KB)
```

Figure 13: PC Wi-Fi adapter IP address

For this project, the IP address was `192.168.1.101`, so we have to set the `ROS_IP` variable as the current IP in `.bashrc`. You can simply copy the following line to the `.bashrc` file:

```
| $ export ROS_IP=192.168.1.101
```

We need to set this; only then will the Android VR app work.

Now start the `roscore` command on the ROS PC:

```
| $ roscore
```

The next step is to open the Android app, and you will get a window like the following. Enter `ROS_IP` in the edit box and click on the CONNECT button.

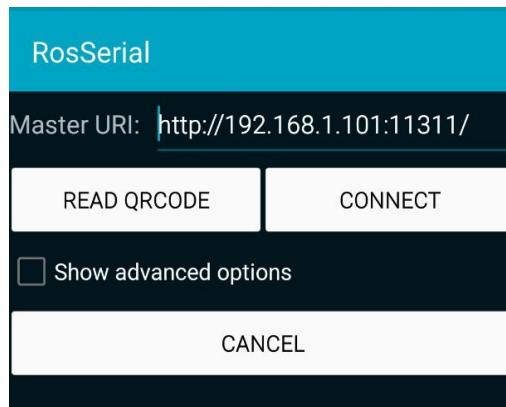


Figure 14: ROS-VR application

If the app is connected to the ROS master on the PC, it will show up as connected and show a blank screen with a split view. Now list out the topics on the ROS PC:

```
lentin@lentin-Aspire-4755:~$ rostopic list
/rosout
/rosout_agg
/usb_cam/image_raw/compressed
lentin@lentin-Aspire-4755:~$
```

Figure 15: Listing ROS-VR topics on PC

You can see topics such as `/usb_cam/image_raw/compressed` or `/camera/image/compressed` in the list, and what we want to do is feed a compressed image to whatever image topic the app is going to subscribe to. If you've installed the `usb_cam` (https://github.com/bosch-ros-pkg/usb_cam) ROS package already, you can launch the webcam driver using the following command:

```
| $ rosrun usb_cam usb_cam-test.launch
```

This driver will publish the camera image in compressed form to the `/usb_cam/image_raw/compressed` topic, and when there is a publisher for this topic, it will display it on the app also. If you are getting some other topics from the app, say, `/camera/image/compressed`, you can use `topic_tools` (http://wiki.ros.org/topic_tools) for remapping the topic to the app topic. You can use the following command:

```
| $ rosrun topic_tools relay /usb_cam/image_raw/compressed /camera/image/compressed
```

Now, you can see the camera view in the VR app like this:



Figure 16: ROS-VR app

This is the split view that we are getting in the application. We can also display images from Gazebo in the similar manner. Simple, right? Just remap the robot camera compressed image to the app topic. In the next section, we will learn how to view Gazebo images in the VR app.

Working with TurtleBot simulation in VR

We can start a TurtleBot simulation using the following command:

```
| $ roslaunch turtlebot_gazebo turtlebot_playground.launch
```

You will get the TurtleBot simulation in Gazebo like this:

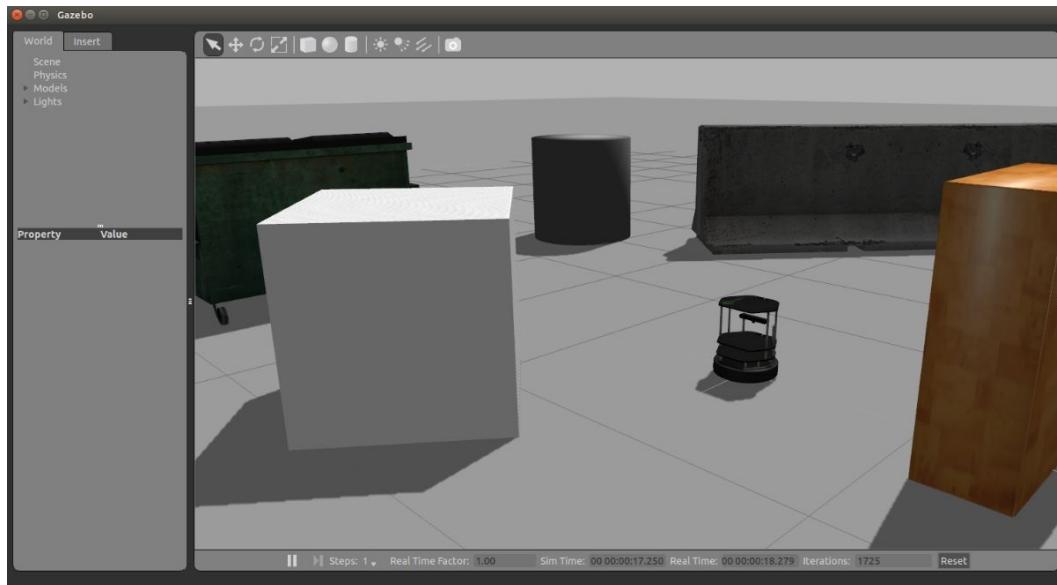


Figure 17: TurtleBot simulation in Gazebo

You can move the robot by launching the teleop node with the following command:

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

You can now move the robot using the keyboard. Launch the app again and connect to the ROS master running on the PC. Then, you can remap the Gazebo RGB image compressed data into an app image topic, like this:

```
| $ rosrun topic_tools relay /camera/rgb/image_raw/compressed /usb_cam/image_raw/compressed
```

Now, what happens is that the robot camera image is visualized in the app, and if you put the phone into a VR headset, it will simulate a 3D environment. The following screenshot shows the split view of the images from Gazebo:

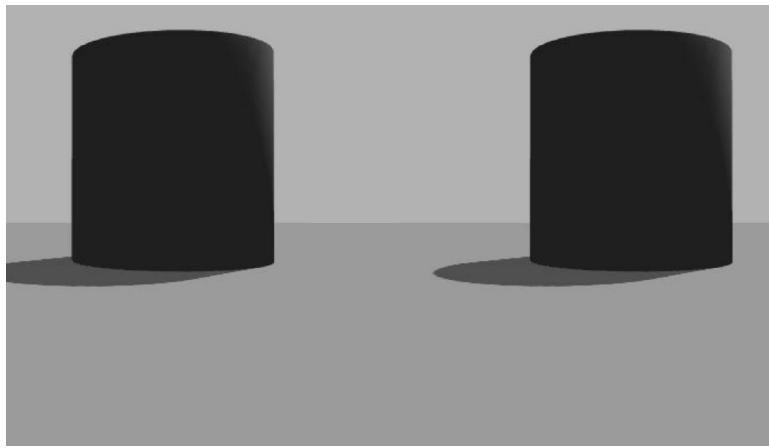


Figure 18: Gazebo image view in ROS-VR app

You can move the robot using a keyboard as of now. In the next section, we can see the possible issues and the solutions that you may encounter when you work with the application.

Troubleshooting the ROS-VR application

You may get issues working with ROS-VR applications. One of the issues may be the size of the image. The left and right image size can vary according to the device screen size and resolution. This project was tested on a full-HD 5-inch screen, and if you have a different screen size or resolution, you may need to hack the application code. You can go to the app's project folder and open the code:

`ros_cardboard/ros_cardboard_module/src/main/java/com/cloudspace/cardboard/CardboardOverlayEyeView.java`. You can change the `final float imageSize = 1.0f` value to `1.8f` or `2f`; this will stretch the image and fill the screen, but we might lose some part of the image. After this change, build it again and install it.

One of the other issues associated with the working of this app is that the app will not work until we set the `ROS_IP` value on the PC. So you should check whether `ROS_IP` is set.

If you want to change the topic name of the app, then go to

`ros_cardboard/ros_cardboard_module/src/main/java/com/cloudspace/cardboard/CardboardViewerActivity.java` and change this line:

```
| mOverlayView.setTopicInformation("/camera/image/compressed",  
| CompressedImage._TYPE);
```



If you want to work with other high-end VR headsets such as Oculus and HTC Vive, you can follow these links: https://github.com/OSUrobotics/ros_ovr_sdk https://github.com/robosavvy/vive_ros http://wiki.ros.org/oculus_rviz_plugins

In the next section, we will combine the power of the VR headset and Leap Motion robot controller node.

Integrating ROS-VR application and Leap Motion teleoperation

In this section, we are going to replace the keyboard teleoperation with Leap Motion-based teleoperation. When we roll our hand to the anticlockwise direction, the robot also rotate anticlockwise, and vice versa. If we pitch our hand down, the robot will move forward, and if we pitch it up, it will move backward. So we can start the VR application and Turtlebot simulation like the previous section and, instead of keyboard teleop, run the Leap teleop node.

So before starting the Leap teleop node, launch the PC driver and ROS driver using the following commands:

```
| $ sudo LeapControlPanel
```

Start the ROS driver using the following command:

```
| $ rosrun leap_motion sensor_sender.launch
```

Now launch Leap Motion on the Twist node using the following command:

```
| $ rosrun vr_leap_teleop vr_leap_teleop.py
```

Now you can put the VR headset on your head and control the robot using your hand.

Questions

- How does a virtual reality headset work?
- How does the Leap Motion controller work?
- What is the algorithm to map from hand coordinates to Twist commands?
- We have installed PC driver for Leap Motion and for working with ROS, we have installed ROS driver. What is the difference between a ROS driver and PC driver?

Summary

This chapter was about creating a project to teleoperate a robot using a Leap Motion controller and VR headset. The basic aim of the chapter was to teleoperate the robot with hand gestures using a Leap Motion controller. After that, we visualized the robot camera image in a VR headset with the help of an Android phone. We started with discussing the general idea of VR and the Leap Motion controller, and then we switched to the design of the project. Then, we discussed the Leap Motion interface with PC and the driver installation. Later, we saw how to build a ROS node to control the robot using our hand. After building a teleop node, we saw how to create a VR app for ROS and then integrated both the app and the teleop node to experience 3D control of the robot using our hands.

Controlling Your Robots over the Web

Until now, we have been controlling and interacting with robots from the command line. What about creating a frontend GUI? If your robot is in a distant location and you want to visualize and control it through the web, this chapter can help you. This is the final chapter of this module, and deals with building a cool interactive web application based on ROS and controlling a robot using it. The projects in this chapter can be mainly used for creating a frontend robot commander in your browser. We'll discuss a few projects using the ROS web framework. Here is a list of the projects and topics we are going to cover in this chapter:

- Getting started with ROS web packages
- Setting up ROS web packages
- Teleoperating and visualizing a robot from a web browser
- Controlling robot joints from a web browser
- Robot surveillance application
- Web-based speech-controlled robot application

Getting started with ROS web packages

ROS offers several powerful and very useful packages to communicate over the Web and interact with robots from web browsers . In the first section, we will discuss some of the open source modules and packages for building cool robot web applications.

The packages that we will discuss here are developed and maintained by the ROS web tools community (<http://robotwebtools.org/>). After discussing the basic web frameworks, we can start discussing projects that use it.

rosbridge_suite

If we want to interact with the ROS framework from our web browser, there should be some system that can convert the web browser commands to the ROS topics/services. `rosbridge` provides a JSON interface to ROS, allowing any client to send JSON commands (<http://www.json.org/>) to publish or subscribe to ROS topics, call ROS services, and more. `rosbridge` supports a variety of transport layers, including WebSockets (<https://en.wikipedia.org/wiki/WebSocket>) and TCP.

The `rosbridge_suite` (http://wiki.ros.org/rosbridge_suite) is a meta-ROS package having an implementation of the `rosbridge` protocol. The JSON commands are converted to ROS topics/services using a node called `rosbridge_server`. This node can send or receive JSON commands from web browsers to ROS over web sockets. The `rosbridge_server` is the intermediate layer between the ROS system and web browser. The complete description of the `rosbridge` and `rosbridge_suit` can be found at https://github.com/RobotWebTools/rosbridge_suite.

The following figure shows how the communication between the `rosbridge` server and web browser happens:

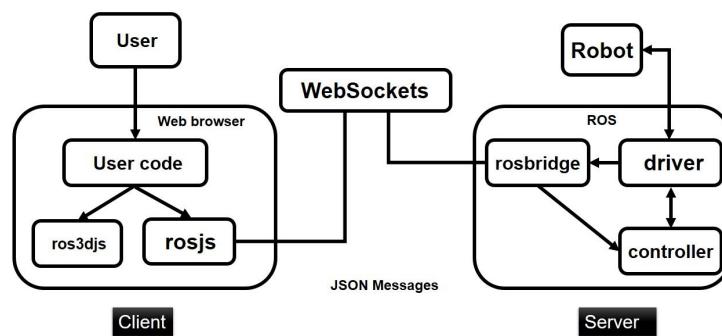


Figure 1: `rosbridge_suite` connection diagram

The `rosbridge` server can communicate with the ROS nodes. It can be a robot controller or ROS nodes. The `rosbridge` can also receive data from ROS which can send to web browser.

The `rosbridge_suite` collection consists of three packages:

- `rosbridge_library`: This package contains Python APIs to convert JSON messages to ROS messages and vice versa.
- `rosbridge_server`: This package has the WebSocket implementation of the `rosbridge` library. We have to start this tool for communicating with the web browser.
- `rosapi`: This provides service calls to fetch meta-information from ROS, such as a list of ROS topics and ROS parameters.

In the web browser, we can see `rosbridge` client. A `rosbridge` client is a program that communicates with `rosbridge` using its JSON API. In the preceding figure, we are using `roslibjs` as the client. Let's understand the main capabilities of these clients.

roslibjs, ros2djs, and ros3djs

In the previous section we have discussed about rosbridge, rosbridge server and rosbridge clients. In this section, we can see a list of rosbridge clients which can be used to send JSON commands from web browsers. Each client is used in different scenario.

Using **roslibjs** (<http://wiki.ros.org/roslibjs>), we can implement basic functionalities of ROS topics, services, actionlib, TF support, URDF, and many more ROS features--using this module.

The **ros2djs** (<http://wiki.ros.org/ros2djs>) is built on top of `roslibjs`. This library provides a 2D visualization manager for ROS. Using this library, we can visualize 2D maps in a web browser.

The **ros3djs** (<http://wiki.ros.org/ros3djs>) library is another cool JavaScript library that can visualize 3D data, such as URDF, TF, interactive markers, and maps. We can create a web-based Rviz instance using its APIs. We will look at some interesting projects using these libraries in the upcoming sections.

The tf2_web_republisher package

The `tf2_web_republisher` (http://wiki.ros.org/tf2_web_republisher) is a useful tool for interacting with robots via web browser. The main function of this package is to precompute TF data and send it via `rosbridge_server` to a `ros3djs` client. The TF data is essential to visualizing the posture and movement of the robot in a web browser.

Setting up ROS web packages on ROS Kinetic

In this section, we are going to see how to set up the previously mentioned libraries on our PC.

Installing rosbridge_suite

We can install `rosbridge_suite` using `apt-get` or build from the source code. First, let's see how to install it via `apt-get`.

Here are the commands to install it:

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

On ROS Kinetic:

```
| $ sudo apt-get install ros-kinetic-rosbridge-suite
```

On ROS Indigo:

```
| $ sudo apt-get install ros-indigo-rosbridge-suite
```

If you are looking for the latest package, you can clone it and install it.

You can switch to your catkin workspace's `src` folder and clone the source code using the following command:

```
| $ git clone https://github.com/RobotWebTools/rosbridge_suite
```

After cloning the folder, you can use `catkin_make`:

```
| $ catkin_make
```

If you encounter any dependency issues, install that package too.

Now we can work with the `rosbridge` client libraries `roslibjs`, `ros2djs`, and `ros3djs`.

Setting up rosbridge client libraries

To store all these library files, you can create a folder called `ros_web_ws`. Actually, there is no need to create a catkin workspace because we don't need to build the modules. You will get the prebuilt modules once you download it from the repositories.

Switch to the `ros_web_ws` folder and run the following commands to clone each ROS JavaScript library:

For `roslibjs`:

```
| $ git clone https://github.com/RobotWebTools/roslibjs.git
```

For `ros2djs`:

```
| $ git clone https://github.com/RobotWebTools/ros2djs
```

For `ros3djs`:

```
| $ git clone https://github.com/RobotWebTools/ros3djs
```

If you check out the `ros3djs` folder, you will see the following:



Figure 2: The `ros3d_js` module folder

The `build` folder contains the `ros3djs.js` modules, which can be used in our web application, and in the `examples` folder, you can find starter web applications. Similar to `ros3djs`, you can also find examples from `roslibjs` and `ros2djs`.



This is the API list of these three modules: `roslibjs` APIs: <http://robotwebtools.org/jsdoc/roslibjs/current/> `ros2djs` APIs: <http://robotwebtools.org/jsdoc/ros2djs/current/> `ros3djs` APIs: <http://robotwebtools.org/jsdoc/ros3djs/current/>

Installing tf2_web_republiser on ROS Kinetic

We can install the `tf2_web_republiser` package by following these steps:

First, switch to your ROS catkin workspace and clone the package code using the following command:

```
| $ git clone https://github.com/RobotWebTools/tf2_web_republiser
```

After cloning the code, install the following package, which may be required to build the preceding package:

```
| $ sudo apt-get install ros-kinetic-tf2-ros
```

After installing the dependent package, you can install `tf2_web_republiser`, by using the `catkin_make` command from the workspace.



If you are getting an error regarding message generation, you can add the following line of code to `package.xml`: `<build_depend>message_generation</build_depend>`

Teleoperating and visualizing a robot on a web browser

This is the first project in this chapter. As we have seen in the other chapters, we are starting with a simple project. This web application can teleoperate the robot from the web browser itself using a keyboard. Along with the teleoperation, we can also visualize the robot in the browser itself. Here is the working block diagram of this project:

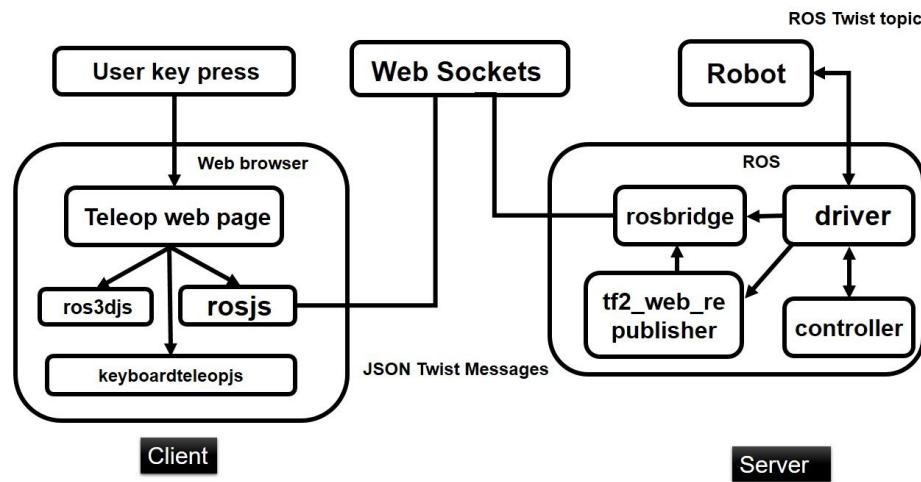


Figure 3: Working of web-based robot keyboard teleoperation project

Working of the project

In this section, we can see the basic working of this project. Imagine that a Turtlebot simulation is running on your PC. We have to control the robot from the web based teleoperation, so when we press a button from web browser, the key press is detected using JavaScript code and map each key press to ROS Twist message. This is done by using rosbridge clients. The rosbridge client sends Twist message as JSON command to the rosbridge server. The communication is happening over WebSockets as shown in the preceding image. When ROS system receives this topic, it can feed to the robot.

At the same time, the TF data and robot description are sending to the rosbridge client for visualizing the robot movement inside the browser. This is done by `tf2_web_republiser`.

We are using a ROS package called `keyboardteleopjs` (<http://wiki.ros.org/keyboardteleopjs>), which can send Twist messages from a web browser according to the key press. Along with that, we are using `ros3djs` to visualize the robot model in the browser. Upon getting the JSON Twist command, the rosbridge server will convert it into the corresponding ROS topic. You find web keyboard teleoperation application `keyboardteleop.html` from `chapter_12_code/ros_web_ws`.

Before running the application, let's discuss the code. Open `keyboardteleop.html` in a text editor, and let's now look at the use of each section of the code.

Basically, web applications are written in HTML/CSS and JavaScript. Similar to other programming languages, initially we have to include the CSS/JS modules that we are going to use in the HTML code. We can call the APIs of these modules and use them in our code. Let's go through the various modules we are using in this code.

The following code snippet includes a CSS file and standard jQuery (<https://jquery.com/>) modules into this code:

```
<link rel="stylesheet" type="text/css"
  href="http://ajax.googleapis.com/
  ajax/libs/jqueryui/1.8/themes/base/jquery-ui.css" /><script
  src="https://ajax.googleapis.com/
  ajax/libs/jquery/1.8.0/jquery.min.js"></script><script
  src="https://ajax.googleapis.com/
  ajax/libs/jqueryui/1.8.23/jquery-ui.min.js"></script>
```

The following code snippet loads JavaScript modules required for loading mesh files into the web browser. We can load mesh files such as STL and COLLADA files, primarily.

```
<script src="http://cdn.robotwebtools.org/
  threejs/current/three.js"></script><script
  src="http://cdn.robotwebtools.org/
  threejs/current/ColladaLoader.js"></script><script
  src="http://cdn.robotwebtools.org/  threejs/current/STLLoader.js">
</script><script src="http://cdn.robotwebtools.org/
  ColladaAnimationCompress/current/ColladaLoader2.js"></script>
```

The following JS modules are importing `roslibjs` and `ros3djs`. The `roslibjs` and `ros3djs` are imported from the `build` folder.

```
<script src="http://cdn.robotwebtools.org/  
EventEmitter2/current/eventemitter2.min.js"></script>  
<script src="../build/roslib.js"></script>  
<script src="../build/ros3d.js"></script>
```

We can also include this from web resources:

```
<script src="http://cdn.robotwebtools.org/  
roslibjs/current/roslib.js"></script>  
<script src="http://cdn.robotwebtools.org/  
ros3djs/current/ros3d.min.js"></script>
```

The following script will help you to perform keyboard teleoperation from a web browser. The script is actually got by downloading the `keyboardteleop` ROS package:

```
| <script src="../build/keyboardteleop.js"></script>
```

Alternatively, we can use the following line from the web resource:

```
| <script src="http://cdn.robotwebtools.org/  
keyboardteleopjs/current/keyboardteleop.js"></script>
```

So we are done with including necessary modules for this application. Next, we have to add JavaScript code inside this HTML code. Following are the main section of the code, which is doing tasks such as connecting to the WebSocket, creating a handler for sending Twist messages, creating a handler for keyboard teleoperation, and creating a new 3D viewer, URDF, and TF client. Let's go through each section one by one.

Connecting to rosbridge_server

The whole initialization of this project is written inside a single function called `init()`. Let's take a look at all the things inside this function.

The first part of the code connects to `rosbridge_server` if it is running. The following code snippet does this:

```
var ros = new ROSLIB.Ros({  
    url : 'ws://localhost:9090'  
});
```

As you can see, we are creating an object of `ROSLIB.Ros` for communicating with `rosbridge_server`. When this code runs, it will connect to `rosbridge_server`, which is listening on `ws://localhost:9090`. Instead of running both on the same system, we can provide the IP address of the system that is running ROS and `rosbridge_server`.

Initializing the teleop

In this section, we'll see how to initialize keyboard teleoperation. We've already discussed a JS module to handle keyboard teleoperation. The following code shows the initialization of that module:

```
var teleop = new KEYBOARDTELEOP.Teleop({  
    ros : ros,  
    topic : teleop_topic  
});
```

This will create a handler of the `KEYBOARDTELEOP.Teleop` class with the given topic name. The topic name is already defined in the beginning of the code. We also need to pass the ROS node object we created earlier.

Creating a 3D viewer inside a web browser

In this section, we will see how to create a 3D viewer for visualizing URDF models inside a web browser. We can define properties of the viewer and the HTML ID for displaying the viewer in the corresponding area:

```
var viewer = new ROS3D.Viewer({
  background : 000,
  divID : 'urdf',
  width : 1280,
  height : 600,
  antialias : true
});
```

The following line of code will add a 3D grid into the 3D viewer:

```
|   viewer.addObject(new ROS3D.Grid());
```

Creating a TF client

The following code creates a TF client, which can subscribe to the TF data from the `tf2_web_republisher` package and update the 3D viewer according to it. Here, we have to mention the fixed frame name, such as Rviz. The fixed frame is already defined in the beginning of our code. For a TurtleBot simulation, it will be `odom`.

```
var tfClient = new ROSLIB.TFClient({
  ros : ros,
  fixedFrame : base_frame,
  angularThres : 0.01,
  transThres : 0.01,
  rate : 10.0
});
```

Creating a URDF client

This section of code creates a URDF client, which is responsible for loading the robot's URDF file. For proper working of the URDF client, we should provide a ROS node object, TF client object, base URL for COLLADA files to load, and the 3D viewer scene object to render the URDF file. To load the meshes into the 3D viewer, we may have to use a mesh loader such as `ROS3D.COLLADA_LOADER` from `Three.js`, which is included in the beginning of the code. This loader can retrieve the COLLADA file from the `robot_description` parameter:

```
var urdfClient = new ROS3D.UrdfClient({
  ros : ros,
  tfClient : tfClient,
  path : 'http://resources.robotwebtools.org/',
  rootObject : viewer.scene,
  loader : ROS3D.COLLADA_LOADER
});
```

After the `init()` function, we can see two other functions. One is for handling the slider, which can set the speed of the robot, and next function is `submit_values()`, which will execute when the Submit button is clicked on. This function retrieves the teleop topic and base frame name from the input text box and calls the `init()` function using it. This tool can be used for teleoperating all robots without changing the code.

Creating text input

The following is the HTML snippet that creates a textbox to enter the teleoperation topic and base frame ID inside the web application. When the button is pressed, the teleop object will start publishing Twist messages to the given input teleoperation topic.

```
<form >
  Teleop topic:<br>
  <input type="text" name="Teleop Topic" id='tele_topic'
  value="/cmd_vel_mux/input/teleop">
  <br>
  Base frame:<br>
  <input type="text" name="Base frame" id='base_frame_name'
  value="/odom">
  <br>

  <input type="button" onmousedown="submit_values()" value="Submit">

</form>
```

The following code tries to load the `init()` function when the web page is loaded, but we've coded it in a way that it will initialize only when the Submit button is pressed:

```
|   <body onload="init()">
```

The slider and 3D viewer are displayed in the following HTML:

```
<div id="speed-label"></div>
<div id="speed-slider"></div>
<div id="urdf"></div>
```

Running the web teleop application

Let's see how we can run this web application.

First, we have to start a robot simulation in Gazebo. Here, we are testing with a TurtleBot simulation. You can launch the TurtleBot simulation using the following command:

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

Now, we can set the parameter `use_gui` to `true`. The robot will only visualize on the browser if this parameter is set.

```
| $ rosparam set use_gui true
```

After running this command, run `tf2_web_republisher` in another Terminal window, using the following command:

```
| $ rosrun tf2_web_republisher tf2_web_republisher
```

After launching it, let's launch the rosbridge server to start WebSocket communication. You can start it using the following command:

```
| $ roslaunch rosbridge_server rosbridge_websocket.launch
```

Congratulations; you are done with the commands that need to be executed from ROS; now, let's open `keyboardteleop.html` in Chrome or Firefox.

You will see the following window in the browser:

Web-browser keyboard teleoperation

Teleop topic:

Base frame:

Run the following commands in the terminal then refresh this page. Check the JavaScript console for the output.

1. `roslaunch turtlebot_gazebo turtlebot_world.launch`
2. `rosparam set use_gui true`
3. `rosrun tf2_web_republisher tf2_web_republisher`
4. `roslaunch rosbridge_server rosbridge_websocket.launch`
5. Use your arrow keys on your keyboard to move the robot (must have this browser window focused).

Figure 4: Initial components in keyboard teleoperation

When you submit the teleop topic and base frame, you can see the 3D visualizer appear in the same window with the robot model. Now you can use keys such as W, S, A, and D to move the robot around the workspace. You can adjust the speed of the robot by moving the slider. Here is the window you will get when you press the Submit button:

Web-browser keyboard teleoperation

Teleop topic:
/ardrone/telop/gpu/teleop
Base frame:
/odom

Run the following commands in the terminal then refresh this page. Check the JavaScript console for the output.

1. rosrun turtlebot teleop world.launch
2. roslaunch ardrone_gazebo gazebo.launch
3. roslaunch tf_web_publisher tf_web_publisher
4. rosrun rosbridge_server rosbridge_websocket.launch
5. Use your arrow keys on your keyboard to move the robot (must have this browser window focused).

Speed: 30%

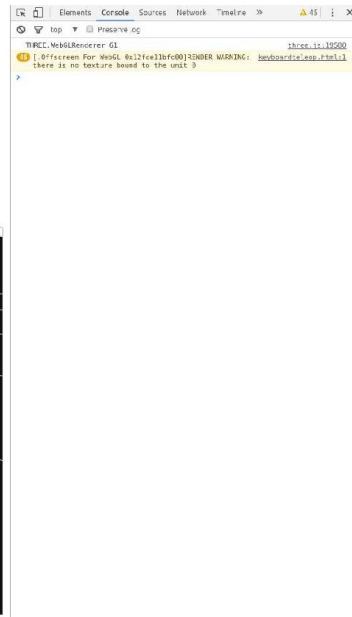
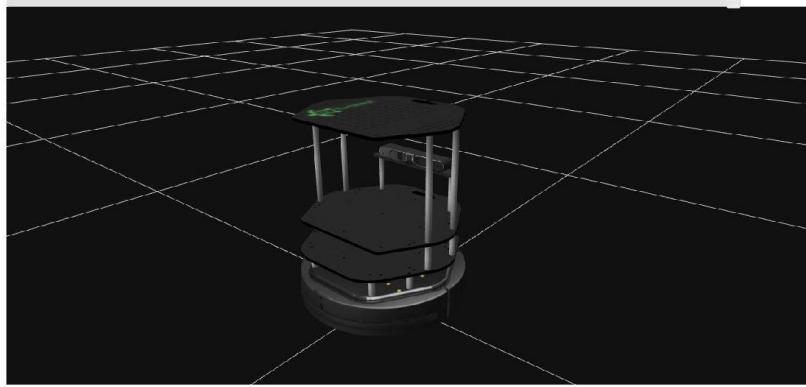


Figure 5: Web-based keyboard teleoperation

In the previous screenshot, you can see a JavaScript console window too. You can enable it by pressing **Ctrl + Shift + I** or right-clicking on the page and using the Inspect option. This window will be useful for debugging.



If you keep on clicking on the Submit button, a new 3D viewer will be created. So refresh the page to change the teleop topic and base frame ID.

Controlling robot joints from a web browser

This is the second project we are going to discuss in this chapter. The aim of this project is to control robot joints from the web browser itself.

Here is the block diagram of the working of a joint state publisher from the web browser:

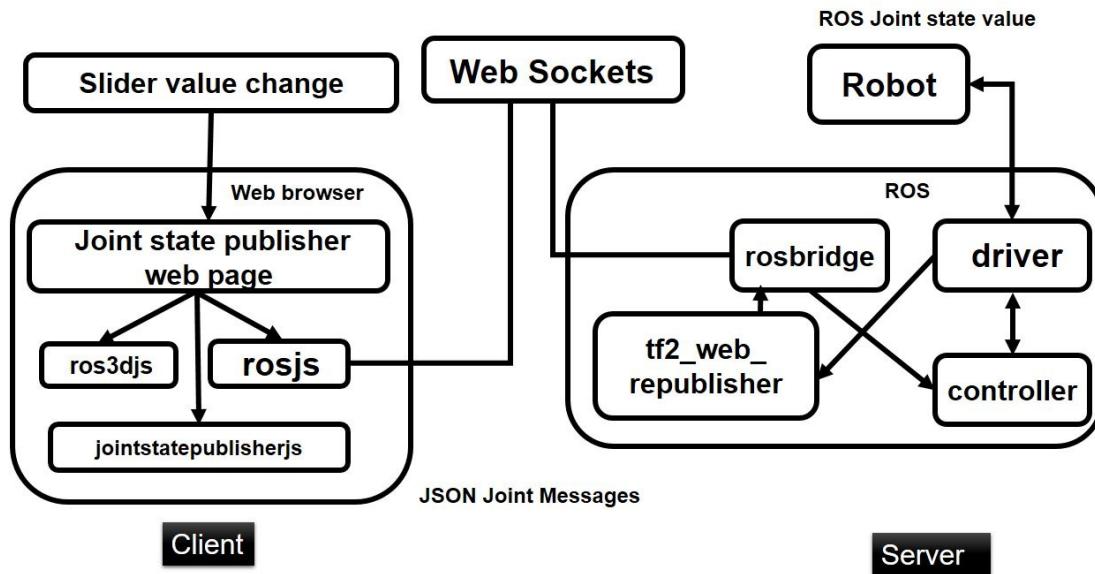


Figure 6: Block diagram of web-based joint state controller

From the block diagram, we can see that we are using another JavaScript module called `jointstatepublisherjs`. This module has a class to create a joint state publisher for all joints defined inside the URDF file.

Installing joint_state_publisher_js

To use this joint state control module, we need to clone the following package into the ROS catkin workspace. Here is the command:

```
| $ git clone https://github.com/DLu/joint_state_publisher_js
```

You can use `catkin_make` after executing the preceding command.

You can see the JavaScript module from the `joint_state_publisher_js/build` folder. You can copy this module and use it for your own applications.

You will get HTML code for the joint state publisher from `chapter_12_codes/ros_web_ws/joint_state_publisher.html`. The code is very similar to our first project, but you can see some new APIs inside this code. Let's look at the new code snippets that you can see in this code.

Including the joint state publisher module

As we have discussed, to enable sliders inside the web browser, we need to include some JavaScript modules. We can insert them from the `build` folder or directly from the Web itself.

Including from the `build` folder:

```
| <script src="../build/jointstatepublisher.js"></script>
```

Including directly from the Web:

```
| <script src="http://cdn.robotwebtools.org/
| jointstatepublisherjs/current/jointstatepublisher.min.js">
| </script>
```

Creating the joint state publisher object

Here is the code snippet for creating the joint state publisher. The sliders will be placed in the HTML `divID` called `sliders`.

```
var jsp = new JOINTSTATEPUBLISHER.JointStatePublisher({  
    ros : ros,  
    divID : 'sliders'  
});
```

Creating an HTML division for sliders

Here is the definition of the HTML `div` element with an `id` of `sliders`.

```
|     <div id="sliders" style="float: right"></div>
```

That's all about the code. Now let's go through the procedure to run this project.

Running the web-based joint state publisher

First, start a robot simulation or load a robot description. Here, we are using the robot model of the PR2 robot. If you don't have this model, you can install it using the following command:

```
| $ sudo apt-get install ros-kinetic-pr2-description
```

After installing, you can load the PR2 description using the following command:

```
| $ roslaunch pr2_description upload_pr2.launch
```

After uploading the code, you can set the ROS parameter called `use_gui` to `true` using the following command:

```
| $ rosparam set use_gui true
```

After doing this, you can start the `joint_state_publisher_js` node using the following command. This will launch joint state publisher, `rosbridge`, and `tf2_web_republiser` node in a single launch file.

```
| $ roslaunch joint_state_publisher_js core.launch
```

Okay, you are done launching the ROS nodes; now, it's time to open the HTML code in a web browser. You can open `joint_state_publisher.html` from `chapter_12_codes/ros_web_ws`.

You will get the following window if everything works fine:

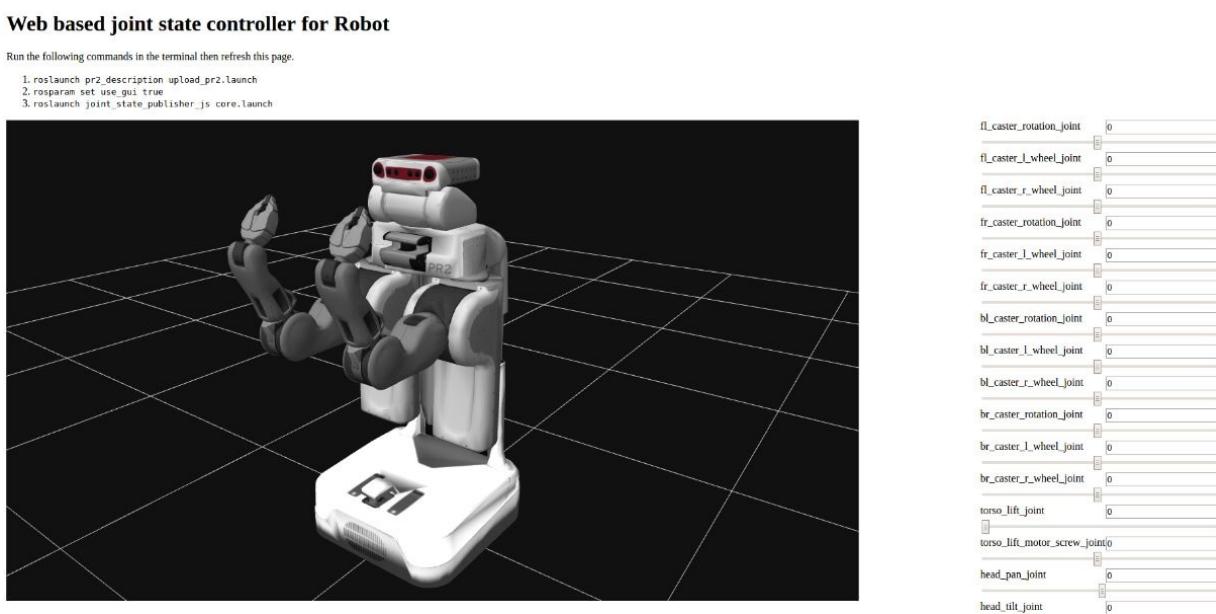


Figure 7: Web-based joint state publisher

Congratulations, you have successfully set up a joint state publisher inside a web browser. Now you can move robot joints by moving the sliders.

Robot surveillance application

This is another interesting web application, which can move a robot and display the camera view of the robot in a browser. This application is best used to teleoperate a robot for surveillance. Let's see how to set it up in ROS Kinetic.

Prerequisites

- `web_video_server`: This is a ROS package for the HTTP streaming of ROS images in multiple image formats. You can find the package from the following ROS wiki page: http://wiki.ros.org/web_video_server
- `mjpegcanvasjs`: This JavaScript module can display the MJPEG stream from `web_video_server` in an HTML canvas. You can get the code from the following link: <http://wiki.ros.org/mjpegcanvasjs>
- `keyboardteleopjs`: This JS module helps us teleoperate a robot from a web browser using a keyboard. We used this module in the first project. You can get it from here: <http://wiki.ros.org/keyboardteleopjs>

Installing prerequisites

Install the `web_video_server` package. Switch to your catkin workspace and clone the package code to the `src` folder:

```
| $ git clone https://github.com/RobotWebTools/web_video_server.git
```

Build the package using the `catkin_make` command.

Download the `mjpegcanvasjs` module. You can simply use the following command:

```
| $ git clone https://github.com/rctoris/mjpegcanvasjs
```

Okay, you are done with the packages and modules. Now you can check the code in `chapter_12_codes/ros_web_ws/ws/Robot_Surveillance.html`.

We'll now discuss the main parts of the code.

Explaining the code

Initially, we have to include the JS module, which is `mjpegcanvas.js`, to get streamer functionality inside the browser. The following code does this job:

```
<script src=" http://cdn.robotwebtools.org/
mjpegcanvasjs/current/mjpegcanvas.js">
</script>
```

The following is the function to start an MJPEG viewer inside the browser. You can set parameters such as width, height, and ROS image topic to display in the viewer.

```
var viewer = new MJPEGCANVAS.Viewer({
  divID : 'mjpeg',
  host : 'localhost',
  width : 640,
  height : 480,
  topic : '/camera/rgb/image_raw',
  interval : 200
});
```

To visualize multiple camera views, we can use code like this. Here, you can add any number of image topics. We also need to mention the image label. In the viewer, we have a provision to select the desired view from the list:

```
var viewer = new MJPEGCANVAS.MultiStreamViewer({
  divID : 'mjpeg',
  host : 'localhost',
  width : 640,
  height : 480,
  topics : [ '/camera/rgb/image_raw', '/camera/rgb/image_raw',
    '/camera/rgb/image_raw' ],
  labels : [ 'Robot View', 'Left Arm View', 'Right Arm View' ]
});
```

Running the robot surveillance application

Okay, so we are ready to run the application. Let's begin.

You can run any robot simulation that has some sort of image topic or camera topic.

For a demo, we will launch the TurtleBot simulation using the following command:

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

After launch the simulation, run the HTTPS streamer node from `web_video_server`:

```
| $ rosrun web_video_server web_video_server
```

After running `web_video_server`, launch `rosbridge_server` to send Twist messages to ROS from the keyboard teleoperation module:

```
| $ roslaunch rosbridge_server rosbridge_websocket.launch
```

Now, open `Robot_Surveillance.html` to look at the output.

Here is the output you will get for the `Robot_Surveillance` application.

Robot Surveillance from Web-browser

Run the following commands in the terminal then refresh this page.

1. roslaunch turtlebot_gazebo turtlebot_world.launch
2. rosrun web_video_server web_video_server
3. roslaunch rosbridge_server rosbridge_websocket.launch



Figure 8: The robot surveillance application

Now you can move the robot and look at the camera view from inside the browser itself.

Web-based speech-controlled robot

The next project we will discuss is to control a robot from a web browser using speech commands. It enables teleoperation of the robot using a button interface and speech. If we are not interested in moving the robot with voice commands, we can try moving the robot using buttons.

We can assign a set of voice commands in this application, and when a voice command is given, the robot will perform the corresponding task.

In this application, we are using basic commands such as `move forward`, `move backward`, `turn left`, and `turn right` to move the mobile robot. We will demo this application using the TurtleBot simulation.

Prerequisites

We need a few things installed for the proper working of this application.

We need to install the `apache2` webserver to run this application. We can install it using the following command:

```
| $ sudo apt-get install apache2
```

This project is actually adapted from a project from roswebtools. The current project can send the command velocity to the robot, but there is no visualization to get feedback of robot motion. So we are adding a 3D viewer inside this application. Here is the existing project:

https://github.com/UbiquityRobotics/speech_commands

You can find the new application's code from `chapter_12_codes/ros_web_ws/speech_commands/speechcommands.html`. We'll now look at the new APIs and code you may need to customize to work with your own robot.

Enabling speech recognition in the web application

Speech recognition functionality is something we haven't discussed yet in any of our projects. Actually, performing speech recognition from a web browser is better than using offline speech recognizers. The reason is that web-based speech recognition uses Google's speech recognition system, which is one of the best speech recognition systems available today. So let's see how we can implement speech recognition in our application.

The web speech API specification (<https://dvcs.w3.org/hg/speech-api/raw-file/tip/speechapi.html>) provides speech recognition and synthesis APIs for a web application, but the majority of web browsers don't support it anymore. Google introduced their own speech recognition and synthesis platform and integrated into these APIs. Now these APIs can work well with Google Chrome.

Let's look at the procedure to enable web-based speech recognition APIs.

The first step is to check whether the browser supports the speech APIs. We can check this using the following code:

```
| if (!('webkitSpeechRecognition' in window)) {  
|   //Speech API not supported here...  
| } else { //Let's do some cool stuff :)
```

If the browser supports speech recognition, we can start the speech recognizer.

First, we have to create a speech recognizer object, which will be used throughout the code:

```
| var recognition = new webkitSpeechRecognition();
```

Now we will configure the speech recognizer object. If we want to implement continuous recognition, we need to mark this as `true`. This is suitable for dictation.

```
| recognition.continuous = true;
```

The following settings enable intermediate speech recognition results even if they are not final:

```
| recognition.interimResults = true;
```

Now we configure the recognition language and accuracy of detection:

```
| recognition.lang = "en-US";  
| recognition.maxAlternatives = 1;
```

After configuring the speech recognition object, we can fill in the callback functions. The callback functions handle each speech recognition object event. Let's look at the main callback of the speech recognition object.

The `start()`callback function calls when the recognition starts, and we may add some visual feedback, such as flashing a red light or something here to alert the user:

```
recognition.onstart = function() {  
};  
};
```

Also, if the speech recognition is finished, the `onend()` callback will be called, and you can give some visual feedback here too:

```
recognition.onend = function() {  
};
```

The following callback, `onresult()`, give the final recognized results of speech recognition:

```
recognition.onresult = function(event) {  
    if (typeof(event.results) === 'undefined') {  
        recognition.stop();  
        return;  
    }  
}
```

After getting the results, we have to iterate inside the result object to get the text output:

```
for (var i = event.resultIndex; i < event.results.length; ++i) {  
    if (event.results[i].isFinal) {  
        console.log("final results: " +  
            event.results[i][0].transcript);  
    }  
    else {  
        console.log("interim results: " +  
            event.results[i][0].transcript);  
    }  
}
```

Now we can start the speech recognition through a user-defined function called `startButton()`. Whenever this function is called, the recognition will start.

```
<div onclick="startButton(event);"></div>  
  
function startButton(event) {  
    recognition.start();  
}
```

Running a speech-controlled robot application

To run the application, you have to copy the `chapter_12_codes/ros_web_ws/speech_commands` folder to `/var/www/html`. If you are in the `ros_web_ws` folder in Terminal, you can use the following command to do this:

```
| $ sudo cp -r speech_commands /var/www/html
```

Now run the following ROS launch files to start the TurtleBot simulation, rosbridge, and `tf2_republiser` nodes:

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

Launch the rosbridge server:

```
| $ roslaunch rosbridge_server rosbridge_websocket.launch
```

Now launch the `tf2_web_republiser` node using the following command:

```
| $ rosrun tf2_web_republiser tf2_web_republiser
```

Okay, you are done with launching all the ROS nodes. Now, let's open Chrome and enter the following address:

localhost/speech_commands/speechcommands.html

If everything works fine, you will get a window like this:

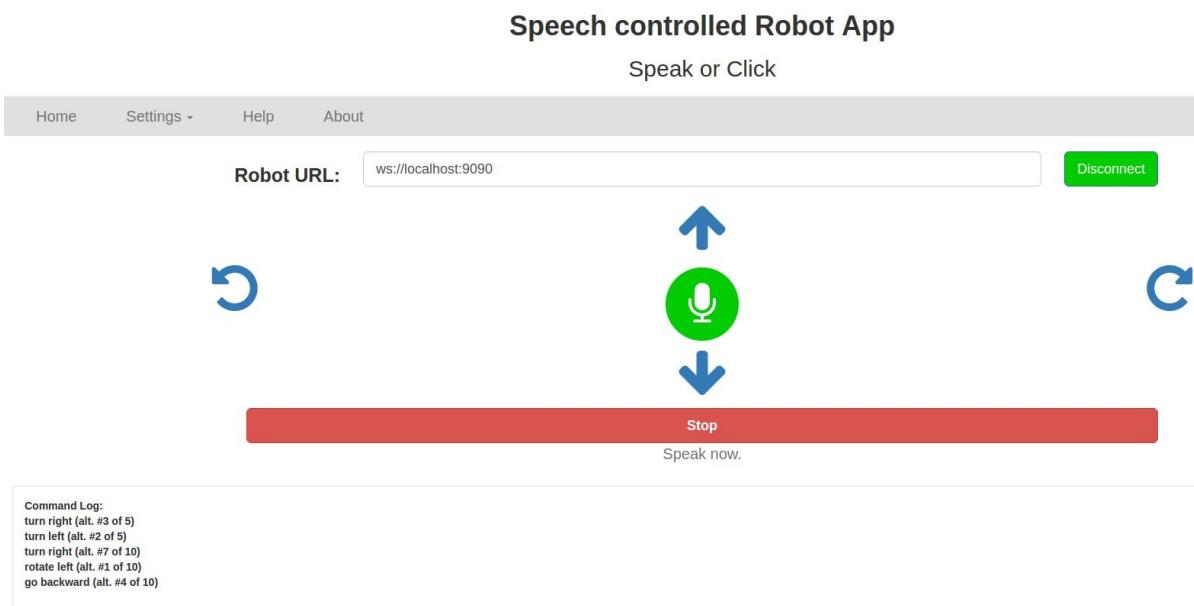


Figure 9: Speech controller Robot App screen

Here, find the Robot URL box, which has to be set to `ws://localhost:9090`, and click on the Connect button. If it connects, you'll get a confirmation, and the Connect button will become Disconnect. After connecting to rosbridge, you can see the 3D viewer inside the browser. Now you can click on the mic symbol. If the mic symbol turns green, then you are done. You can give it a command and the robot will start moving. If the

mic did not turn green, you may need to check Chrome's mic settings to allow mic access to this app. The complete app will look like this:

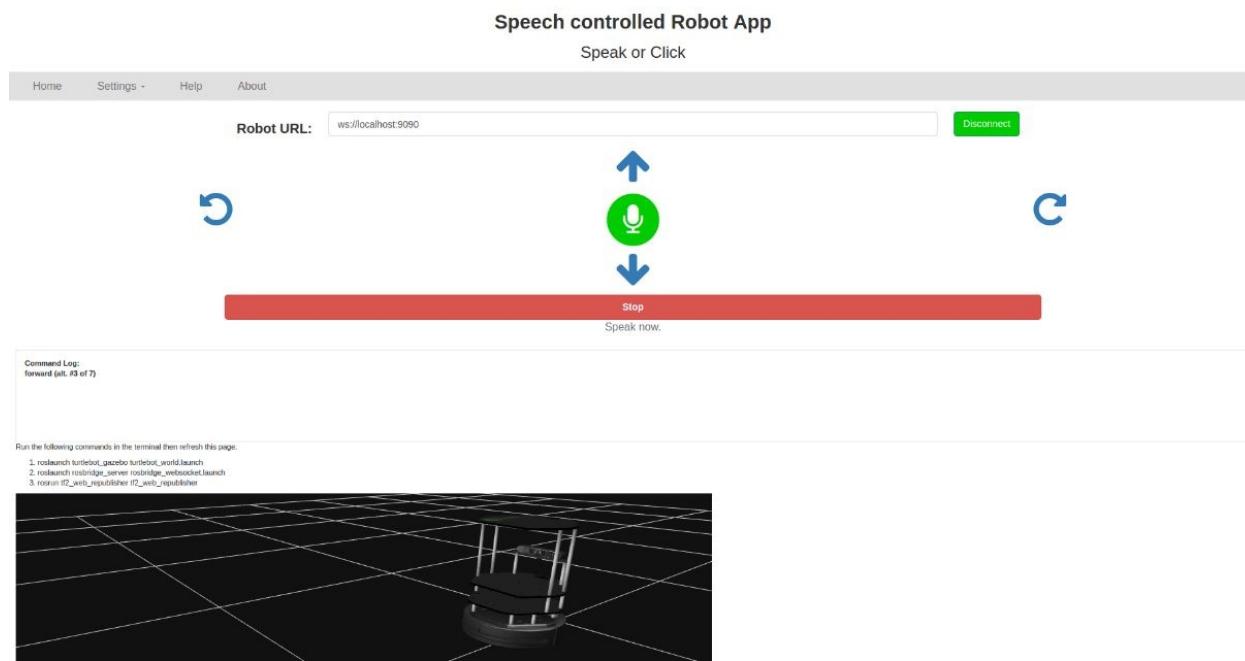


Figure 10: Speech controller Robot App screen

You can see the detected commands in the Command Log box. You can also move the robot using the arrow keys shown in the window.

Questions

- What is the main use of the `rosbridge_server` package?
- What is the use of `roslibjs` and `ros3djs`?
- What is the ROS package used to stream images from ROS to a web browser?
- How is the robot surveillance application used?

Summary

This chapter was about creating interactive web applications using ROS. The chapter was started with discussing basic ROS packages and JavaScript modules used for building a robot web application. After discussing the packages, we discussed how to install them. After setting up all the packages, we started our first project, teleoperating the robot from a web browser. In that application, we controlled the robot using a keyboard and visualized the robot at the same time. The next project was about controlling the joint state of the robot. We created the application and tested it on the PR2 robot. The third project was about creating a robot surveillance application, which combines keyboard teleoperation and image streaming from the robot. The last project was about creating a cool speech recognition-based robot controller application.

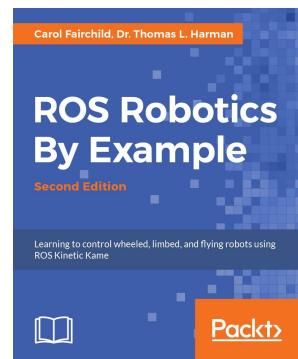
Bibliography

This Learning Path combines some of the best that Packt has to offer in one complete, curated package. It includes content from the following Packt products:

- *Effective Robotics Programming with ROS - Third Edition* by Anil Mahtani, Luis Sanchez, Enrique Fernandez and Aaron Martinez
- Mastering ROS for Robotics Programming - by Lentin Joseph
- *ROS Robotics Projects* - by Lentin Joseph

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:



ROS Robotics By Example, Second Edition

Carol Fairchild, Dr. Thomas L. Harman

ISBN: 978-1-78847-959-2

- Control a robot without requiring a PhD in robotics
- Simulate and control a robot arm
- Control a flying robot
- Send your robot on an independent mission
- Learning how to control your own robots with external devices
- Program applications running on your robot
- Extend ROS itself
- Extend ROS with the MATLAB Robotics System Toolbox

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!