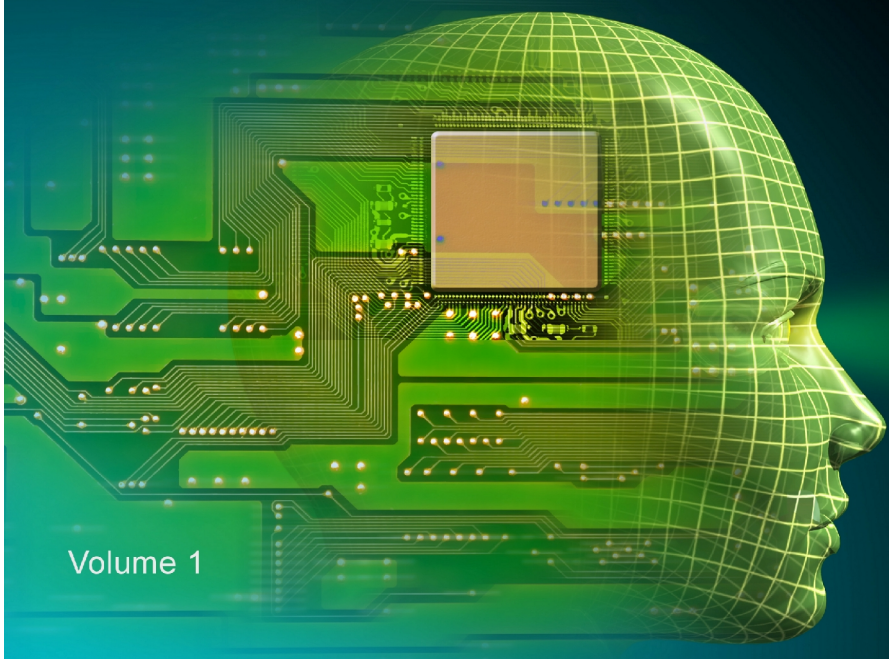
ROS By Example

A Do-It-Yourself Guide to the Robot Operating System



Volume 1

R. Patrick Goebel

ROS By Example

A Do-It-Yourself Guide to the Robot Operating System

VOLUME 1

A PI ROBOT PRODUCTION

R. PATRICK GOEBEL

Version 1.04

For ROS Hydro

ISBN: 5-800085-311092

Version 1.04 for ROS Hydro: January 2014

PREFACE

This book is about programming your robot to do amazing things, whether it be detecting people's faces and other visual objects, navigating autonomously around your house, or responding to spoken commands. We will be using some of the most advanced robotics software available today thanks to ROS, the Robot Operating System created by California-based Willow Garage and now maintained by the Open Source Robotics Foundation (OSRF).

The primary goal of ROS (pronounced "Ross") is to provide a unified and open source programming framework for controlling robots in a variety of real world and simulated environments. ROS is certainly not the first such effort; in fact, doing a Wikipedia search for "robot software" turns up 15 such projects. But Willow Garage is no ordinary group of programmers banging out free software. Propelled by some serious funding, strong technical expertise, and a well planned series of developmental milestones, Willow Garage has ignited a kind of programming fever among roboticists with hundreds of user-contributed ROS packages already created in just a few short years. ROS now includes software for tasks ranging from navigation and localization (SLAM), 3D object recognition, action planning, motion control for multi-jointed arms, machine learning and even playing billiards.

In the meantime, Willow Garage has also designed and manufactured a $400,000 robot called the PR2 to help showcase its operating system. Using the latest in robot hardware, including two stereo cameras, a pair of laser scanners, arms with 7 degrees of freedom, and an omni-directional drive system, only a lucky few will be able to run ROS directly on the PR2, including 11 research institutions that were awarded free PR2s as part of a beta-test contest. However, you do not need a PR2 to leverage the power of ROS; packages have already been created to support lower-cost platforms and components including the iRobot Create, TurtleBot, Arduino, WowWee Rovio, LEGO® NXT, Phidgets, ArbotiX, Serializer, Element and Robotis Dynamixels.

The guiding principle underlying ROS is "don't reinvent the wheel". Many thousands of very smart people have been programming robots for over fifty years—why not bring all that brain power together in one place? Fortunately, the Web is the perfect medium for sharing code. Many universities, companies and individuals now openly share their ROS code repositories, and with free cloud space available through services such as Google Code or GitHub, anyone can share their own ROS creations easily and at no cost.

Perhaps the best part of jumping on the ROS train is the excitement that comes from working with thousands of like-minded roboticists from around the world. Not only will you save many days of frustration by not duplicating someone else's work, you can also feel the satisfaction that comes from contributing back to this rapidly developing field.

Printed *vs* PDF Versions of the Book

The printed and PDF versions of this book are nearly the same with a few important differences. The page formatting is identical but most PDF readers start page numbering from the first page and ignore what is set by the document itself. Images and code syntax are in color in the PDF version but grayscale in the printed version to keep the cost reasonable. The PDF version is full of clickable links to other resources on the Web. In the printed version, these links appear as underlined text with a numbered superscript. The expanded URLs are then listed as endnotes at the end of the book.

Staying Up-To-Date: If you'd like to receive notifications of updates to both the book and the accompanying code, please join the ros-by-example Google Group.

CHANGES SINCE GROOVY

If Hydro is your first experience with ROS and this book, then you can safely skip this chapter. However, if you have already been using previous versions of this book with ROS Groovy or earlier ROS versions, there are a few changes to be noted. You can read the official list of differences between ROS Groovy and Hydro on the Groovy-> Hydro migration page. Here are some of the items that affect the code used with this book.

Changes to ROS

- The `catkin` build system is now required for packages submitted to the ROS build farm for conversion to Debian packages that can be installed using apt-get. The `ros-by-example` stack used with this book has therefore been converted to a catkin meta package. For instructions on creating your own `catkin` packages, see the catkin tutorials on the ROS Wiki. You can also refer to Chapter 4 in this book.

- Catkin requires Python modules in a package to be located under the directory `package/src/package_name`. For the `ros-by-example` code, this required moving several Python files out of the `nodes` subdirectory of some packages and into the corresponding `src/package_name` directory. In particular, the `rbx1_vision` package required moving `face_detector.py`, `good_features.py` and `lk_tracker.py` from `rbx1_vision/nodes` into `rbx1_vision/src/rbx1_vision`.

- For Python scripts, importing `roslib` and running `roslib.load_manifest()` is no longer necessary. Therefore, these lines have been removed from all Python nodes in the `ros-by-example` code.

- The various wxWidgets ROS tools such as `rxconsole`, `rxplot`, etc have been replaced by Qt versions called `rqt_console`, `rqt_plot`, etc. The basic functionality remains the same but you must now use the new versions. (Both versions were available in Groovy.) For a complete list of tools, see the `rqt_common_plugins` Wiki page.

- Thanks to David Lu!!, the ROS navigation stack can now use layered costmaps. While beyond the scope of this volume, this approach enables one to create custom costmaps for specific situations. Fortunately, ROS Hydro can still use the previous costmap format without having to worry about layering at this point.

- Another difference regarding costmaps is that they are now published and displayed as *maps* instead of *grid squares* in `RViz`. This requires changing the display type in `RViz` when using any previously saved configuration files that

display obstacles as grid squares. This update has already been done for the Hydro version of the `ros-by-example` config files.

- The `tf` library has been updated to `tf2`. The new library is a superset of the original and does not provide any new functionality for the code used in this book. We we will therefore continue to use `tf` in this volume.

- One casualty of the new `tf2` library is that the `tf_prefix` has been depreciated. This required re-programming the `skeleton_markers` and `pi_tracker` packages that are referred to in chapter 10.

- As of this writing, the `openni_tracker` package is broken in some Hydro installations. A workaround is provided in section 10.9 where we discuss skeleton tracking.

- The `<include>` tag used in URDF models has been depreciated and replaced with `<xacro:include>`. All URDF files included in the `ros-by-example` code have been updated accordingly.

Changes to the Sample Code

- The `head_tracker.py` script in the `rbx1_dynamixels` package has been updated to be thread safe. This should prevent some random servo motions that were appearing in previous versions.

- An new script named `object_follower.py` has been added to the `rbx1_apps` package. This script extends the `object_tracker.py` program to include depth information so that the robot can be made to follow a moving target.

- The `face_tracker2.py` script in the `rbx1_vision` package now properly responds to depth information when the parameter `use_depth_for_tracking` is set to `True`. A bug in earlier versions of the script caused this parameter to be silently ignored.

Main Chapter Headings

Preface.....vii

Printed vs PDF Versions of the Book.....ix

Changes Since Groovy.....xi

1. Purpose of this Book.....1

2. Real and Simulated Robots.....3

3. Operating Systems and ROS Versions.....5

4. Reviewing the ROS Basics.....9

5. Installing the ros-by-example Code.....29

6. Installing the Arbotix Simulator.....33

7. Controlling a Mobile Base.....37

8. Navigation, Path Planning and SLAM.....75

9. Speech Recognition and Synthesis.....123

10. Robot Vision.....139

11. Combining Vision and Base Control.....197

12. Dynamixel Servos and ROS.....219

13. Where to Go Next?.....249

Contents

Preface vii

Printed vs PDF Versions of the Book ix

Changes Since Groovy xi

1. Purpose of this Book 1

2. Real and Simulated Robots 3

2.1 Gazebo, Stage, and the ArbotiX Simulator 3
2.2 Introducing the TurtleBot, Maxwell and Pi Robot 4

3. Operating Systems and ROS Versions 5

3.1 Installing Ubuntu Linux 5
3.2 Getting Started with Linux 6
3.3 A Note about Updates and Upgrades 7

4. Reviewing the ROS Basics 9

4.1 Installing ROS 9
4.2 Installing rosinstall 9
4.3 Building ROS Packages with Catkin 10
4.4 Creating a catkin Workspace 10
4.5 Doing a "make clean" with catkin 11
4.6 Rebuilding a Single catkin Package 11
4.7 Mixing catkin and rosbuild Workspaces 12
4.8 Working through the Official ROS Tutorials 13
4.9 RViz: The ROS Visualization Tool 13
4.10 Using ROS Parameters in your Programs 14
4.11 Using rqt_reconfigure (formerly dynamic_reconfigure) to set ROS Parameters 14
4.12 Networking Between a Robot and a Desktop Computer 16
4.12.1 Time Synchronization 16
4.12.2 ROS Networking using Zeroconf 16
4.12.3 Testing Connectivity 17
4.12.4 Setting the ROS_MASTER_URI and ROS_HOS TNAME Variables 17
4.12.5 Opening New Terminals 18
4.12.6 Running Nodes on both Machines 19
4.12.7 ROS Networking across the Internet 20
4.13 ROS Recap 21
4.14 What is a ROS Application? 21
4.15 Installing Packages with SVN, Git, and Mercurial 22
4.15.1 SVN 23

4.15.2 Git 23
4.15.3 Mercurial 24
4.16 Removing Packages from your Personal catkin Directory 24
4.17 How to Find Third-Party ROS Packages 25
4.17.1 Searching the ROS Wiki 25
4.17.2 Using the roslocate Command 25
4.17.3 Browsing the ROS Software Index 27
4.17.4 Doing a Google Search 27
4.18 Getting Further Help with ROS 27

5. Installing the ros-by-example Code 29

5.1 Installing the Prerequisites 29
5.2 Cloning the Hydro ros-by-example Repository 29
5.2.1 Upgrading from Electric or Fuerte 29
5.2.2 Upgrading from Groovy 30
5.2.3 Cloning the rbx1 repository for Hydro 30
5.3 About the Code Listings in this Book 31

6. Installing the Arbotix Simulator 33

6.1 Installing the Simulator 33
6.2 Testing the Simulator 33
6.3 Running the Simulator with Your Own Robot 35

7. Controlling a Mobile Base 37

7.1 Units and Coordinate Systems 37
7.2 Levels of Motion Control 37
7.2.1 Motors, Wheels, and Encoders 38
7.2.2 Motor Controllers and Drivers 38
7.2.3 The ROS Base Controller 38
7.2.4 Frame-Base Motion using the move_base ROS Package 39
7.2.5 SLAM using the gmapping and amcl ROS Packages 39
7.2.6 Semantic Goals 40
7.2.7 Summary 40
7.3 Twisting and Turning with ROS 41
7.3.1 Example Twist Messages 41
7.3.2 Monitoring Robot Motion using RViz 42
7.4 Calibrating Your Robot's Odometry 44
7.4.1 Linear Calibration 45
7.4.2 Angular Calibration 46
7.5 Sending Twist Messages to a Real Robot 47
7.6 Publishing Twist Messages from a ROS Node 49
7.6.1 Estimating Distance and Rotation Using Time and Speed 49
7.6.2 Timed Out-and-Back in the ArbotiX Simulator 49
7.6.3 The Timed Out-and-Back Script 50
7.6.4 Timed Out and Back using a Real Robot 55
7.7 Are We There Yet? Going the Distance with Odometry 57
7.8 Out and Back Using Odometry 59
7.8.1 Odometry-Based Out and Back in the ArbotiX Simulator 59

7.8.2 Odometry-Based Out and Back Using a Real Robot 60
7.8.3 The Odometry-Based Out-and-Back Script 62
7.8.4 The /odom Topic versus the /odom Frame 67
7.9 Navigating a Square using Odometry 68
7.9.1 Navigating a Square in the ArbotiX Simulator 68
7.9.2 Navigating a Square using a Real Robot 69
7.9.3 The nav_square.py Script 71
7.9.4 The Trouble with Dead Reckoning 71
7.10 Teleoperating your Robot 71
7.10.1 Using the Keyboard 72
7.10.2 Using a Logitech Game Pad 73
7.10.3 Using the ArbotiX Controller GUI 73
7.10.4 TurtleBot Teleoperation Using Interactive Markers 74
7.10.5 Writing your Own Teleop Node 74

8. Navigation, Path Planning and SLAM 75

8.1 Path Planning and Obstacle Avoidance using move_base 75
8.1.1 Specifying Navigation Goals Using move_base 76
8.1.2 Configuration Parameters for Path Planning 77
8.1.2.1 base_local_planner_params.yaml 78
8.1.2.2 costmap_common_params.yaml 79
8.1.2.3 global_costmap_params.yaml 80
8.1.2.4 local_costmap_params.yaml 80
8.2 Testing move_base in the ArbotiX Simulator 81
8.2.1 Point and Click Navigation in RViz 86
8.2.2 Navigation Display Types for RViz 87
8.2.3 Navigating a Square using move_base 87
8.2.4 Avoiding Simulated Obstacles 94
8.2.5 Setting Manual Goals with Obstacles Present 97
8.3 Running move_base on a Real Robot 97
8.3.1 Testing move_base without Obstacles 97
8.3.2 Avoiding Obstacles using a Depth Camera as a Fake Laser 99
8.4 Map Building using the gmapping Package 101
8.4.1 Laser Scanner or Depth Camera? 102
8.4.2 Collecting and Recording Scan Data 104
8.4.3 Creating the Map 106
8.4.4 Creating a Map from Bag Data 106
8.4.5 Can I Extend or Modify an Existing Map? 108
8.5 Navigation and Localization using a Map and amcl 108
8.5.1 Testing amcl with Fake Localization 108
8.5.2 Using amcl with a Real Robot 110
8.5.3 Fully Autonomous Navigation 113
8.5.4 Running the Navigation Test in Simulation 113
8.5.5 Understanding the Navigation Test Script 115
8.5.6 Running the Navigation Test on a Real Robot 120
8.5.7 What's Next? 122

9. Speech Recognition and Synthesis 123

9.1 Installing PocketSphinx for Speech Recognition 123

9.2 Testing the PocketSphinx Recognizer 123
9.3 Creating a Vocabulary 125
9.4 A Voice-Control Navigation Script 127
9.4.1 Testing Voice-Control in the ArbotiX Simulator 132
9.4.2 Using Voice-Control with a Real Robot 133
9.5 Installing and Testing Festival Text-to-Speech 134
9.5.1 Using Text-to-Speech within a ROS Node 136
9.5.2 Testing the talkback.py script 138

10. Robot Vision 139

10.1 OpenCV, OpenNI and PCL 139
10.2 A Note about Camera Resolutions 140
10.3 Installing and Testing the ROS Camera Drivers 140
10.3.1 Installing the OpenNI Drivers 140
10.3.2 Installing Webcam Drivers 140
10.3.3 Testing your Kinect or Xtion Camera 141
10.3.4 Testing your USB Webcam 142
10.4 Installing OpenCV on Ubuntu Linux 143
10.5 ROS to OpenCV: The cv_bridge Package 144
10.6 The ros2opencv2.py Utility 149
10.7 Processing Recorded Video 151
10.8 OpenCV: The Open Source Computer Vision Library 152
10.8.1 Face Detection 152
10.8.2 Keypoint Detection using GoodFeaturesToTrack 158
10.8.3 Tracking Keypoints using Optical Flow 164
10.8.4 Building a Better Face Tracker 170
10.8.5 Dynamically Adding and Dropping Keypoints 173
10.8.6 Color Blob Tracking (CamShift) 175
10.9 OpenNI and Skeleton Tracking 181
10.9.1 Checking your OpenNI installation for Hydro 182
10.9.2 Viewing Skeletons in RViz 183
10.9.3 Accessing Skeleton Frames in your Programs 183
10.10 PCL Nodelets and 3D Point Clouds 191
10.10.1 The PassThrough Filter 191
10.10.2 Combining More than One PassThrough Filter 193
10.10.3 The VoxelGrid Filter 194

11. Combining Vision and Base Control 197

11.1 A Note about Camera Coordinate Axes 197
11.2 Object Tracker 197
11.2.1 Testing the Object Tracker with rqt_plot 197
11.2.2 Testing the Object Tracker with a Simulated Robot 198
11.2.3 Understanding the Object Tracker Code 199
11.2.4 Object Tracking on a Real Robot 205
11.3 Object Follower 206
11.3.1 Adding Depth to the Object Tracker 206
11.3.2 Testing the Object Follower with a Simulated Robot 210
11.3.3 Object Following on a Real Robot 211

11.4 Person Follower 212
11.4.1 Testing the Follower Application in Simulation 213
11.4.2 Understanding the Follower Script 213
11.4.3 Running the Follower Application on a TurtleBot 217
11.4.4 Running the Follower Node on a Filtered Point Cloud 218

12. Dynamixel Servos and ROS 219

12.1 A TurtleBot with a Pan-and-Tilt Head 220
12.2 Choosing a Dynamixel Hardware Controller 220
12.3 A Note Regarding Dynamixel Hardware 221
12.4 Choosing a ROS Dynamixel Package 221
12.5 Understanding the ROS JointState Message Type 221
12.6 Controlling Joint Position, Speed and Torque 222
12.6.1 Setting Servo Position 223
12.6.2 Setting Servo Speed 224
12.6.3 Controlling Servo Torque 224
12.7 Checking the USB2Dynamixel Connection 225
12.8 Setting the Servo Hardware IDs 226
12.9 Configuring and Launching dynamixel_controllers 227
12.9.1 The dynamixel_controllers Configuration File 227
12.9.2 The dynamixel_controllers Launch File 228
12.10 Testing the Servos 230
12.10.1 Starting the Controllers 230
12.10.2 Monitoring the Robot in RViz 231
12.10.3 Listing the Controller Topics and Monitoring Joint States 231
12.10.4 Listing the Controller Services 233
12.10.5 Setting Servo Position, Speed and Torque 234
12.10.6 Using the relax_all_servos.py Script 235
12.11 Tracking a Visual Target 236
12.11.1 Tracking a Face 236
12.11.2 The Head Tracker Script 237
12.11.3 Tracking Colored Objects 244
12.11.4 Tracking Manually Selected Targets 245
12.12 A Complete Head Tracking ROS Application 246

13. Where to Go Next? 249

1. Purpose of this Book

ROS is extremely powerful and continues to expand and improve at a rapidly accelerating pace. Yet one of the challenges facing many new ROS users is knowing where to begin. There are really two phases to getting started with ROS: Phase 1 involves learning the basic concepts and programming techniques while Phase 2 is about using ROS to control your own robot.

Phase 1 is best tackled by referring to the ROS Wiki where you will find a set of installation instructions and a collection of superbly written Beginner Tutorials. These tutorials have been battle-tested by hundreds if not thousands of users so there is no sense in duplicating them here. **These tutorials are considered a prerequisite for using this book.** We will therefore assume that the reader has worked through all the tutorials at least once. It is also essential to read the tf overview and do the tf Tutorials which will help you understand how ROS handles different frames of reference. If you run into trouble, check out the ROS Answers (http://answers.ros.org) forum where many of your questions may already have been answered. If not, you can post your new question there. (Please do not use the `ros-users` mailing list for such questions which is reserved for ROS news and announcements.)

Phase 2 is what this book is all about: using ROS to make your robot do some fairly impressive tasks. Each chapter will present tutorials and code samples related to different aspects of ROS. We will then apply the code to either a real-world robot, a pan-and-tilt head, or even just a camera (e.g. face detection). For the most part, you can do these tutorials in any order you like. At the same time, the tutorials will build on one another so that by the end of the book, your robot will be able to autonomously navigate your home or office, respond to spoken commands, and combine vision and motion control to track faces or follow a person around the house.

In this volume we will cover the following topics:

- Installing and configuring ROS (review).
- Controlling a mobile base at different levels of abstraction, beginning with motor drivers and wheel encoders, and proceeding upward to path planning and map building.
- Navigation and SLAM (Simultaneous Localization And Mapping) using either a laser scanner or a depth camera like the Microsoft Kinect or Asus Xtion.
- Speech recognition and synthesis, as well as an application for controlling your robot using voice commands.

- Robot vision, including face detection and color tracking using OpenCV, skeleton tracking using OpenNI, and a brief introduction to PCL for 3D vision processing.
- Combining robot vision with a mobile base to create two applications, one for tracking faces and colored objects, and another for following a person as they move about the room.
- Controlling a pan-and-tilt camera using Dynamixel servos to track a moving object.

Given the breadth and depth of the ROS framework, we must necessarily leave out a few topics in this introductory volume. In particular, we will not cover the Gazebo simulator, MoveIt! (formerly arm navigation), tabletop object manipulation, Ecto, creating your own URDF robot model, robot diagnostics, or the use of task managers such as SMACH. Nonetheless, as you can see from the list above, we still have much to do!

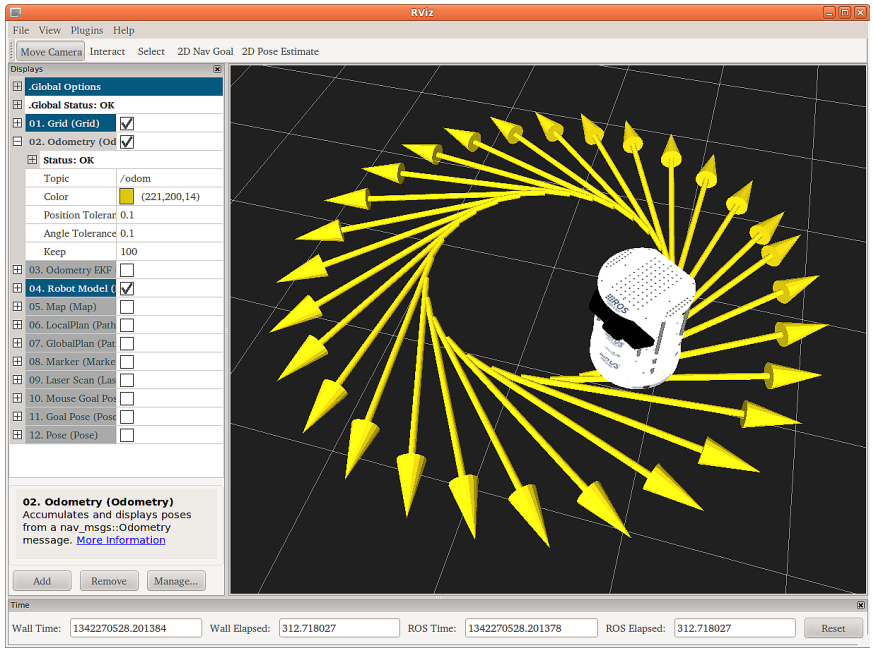
2. Real and Simulated Robots

While ROS can be used with a large variety of hardware, you don't need an actual robot to get started. ROS includes packages to run a number of robots in simulation so that you can test your programs before venturing into the real world.

2.1 Gazebo, Stage, and the ArbotiX Simulator

There are a number of ways to simulate a robot in ROS. The most sophisticated uses Gazebo, a full-featured 3-D physics simulator that pre-dates ROS but now integrates nicely with it. The second uses Stage, a simpler 2-D simulator that can manage multiple robots and various sensors such as laser scanners. The third uses Michael Ferguson's arbotix_python package that can run a fake simulation of a differential drive robot but without sensor feedback or any physics. We will use this last method as it is the simplest to set up and we don't need physics for our purposes. Of course, feel free to explore Gazebo and Stage but be prepared to spend a little time working on the details. Also, Gazebo in particular demands a fair bit of CPU power.

Even if you do have your own robot, it is a good idea to run some of the examples in this book in the simulator first. Once you are happy with the results, you can try it on your robot.

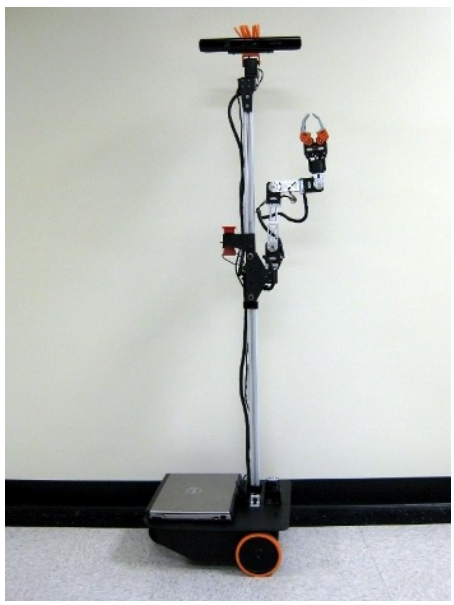


2.2 Introducing the TurtleBot, Maxwell and Pi Robot

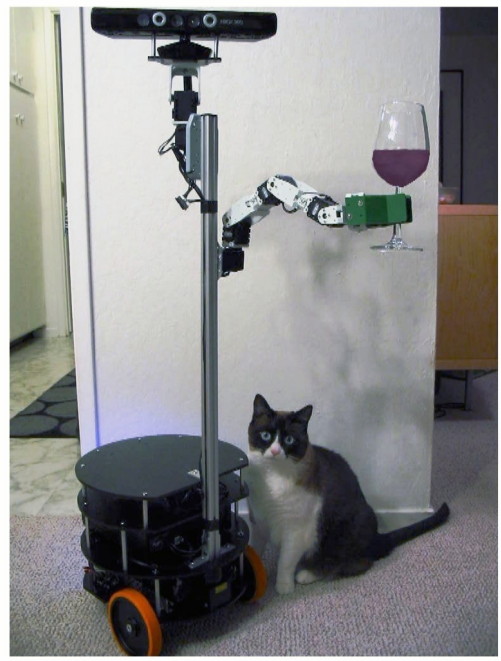
For the purposes of this book, we need a robot that we can at least run in simulation to test our code. The `ros-by-example` repository includes support for two test robots, the Willow Garage TurtleBot created by Melonee Wise and Tully Foote, and the author's own home-built robot called Pi Robot.

The original TurtleBot

Pi was inspired by Michael Ferguson's Maxwell which in turn was modeled after Georgia Tech's EL-E robot. If you have a URDF model of your own robot, you can use it instead of one of these. In any event, most of the code we develop will run on almost any robot that supports the basic ROS message interfaces.



Maxwell



Pi Robot

3. Operating Systems and ROS Versions

ROS can run on various flavors of Linux, MacOS X and partially on Microsoft Windows. However, the easiest way to get started is to use Ubuntu Linux as this is the OS officially supported by OSRF. Furthermore, Ubuntu is free and easy to install alongside other operating systems. In case you are not already using Ubuntu, we will list a few pointers on installing it in the next section.

ROS can be run on anything from a supercomputer to a Beagleboard. But much of the code we will develop will require some CPU-intensive processing. You will therefore save yourself some time and potential frustration if you use a laptop or other full-size PC to run the sample applications in this book. The Turtlebot, Maxwell, and Pi Robot are designed so that you can easily place a laptop on the robot. This means that you can develop your code directly on the laptop, then simply slide it onto the robot for testing autonomous behavior. You can even go half-way during development and tether the robot to your laptop or PC using some long USB cables.

The latest version of Ubuntu is 13.04 (Raring) and the latest version of ROS is Hydro. This version of the book was tested against ROS Hydro and Ubuntu 12.04 (Precise) which is the current Long Term Support (LTS) version of Ubuntu.

Multiple versions of ROS can happily coexist on your machine (if supported by your OS) but be sure that Hydro is your active version when running the commands and code samples in this book.

NOTE: It is generally required that you run the same version of ROS on all machines connected on the same ROS network. This includes the computer on your robot and your desktop workstation. The reason is that ROS message signatures have been known to change from one release to another so that different releases are generally unable to utilize each other's topics and services.

3.1 Installing Ubuntu Linux

If you already have Ubuntu Linux up and running on your machine, great. But if you're starting from scratch, installing Ubuntu is not difficult.

The latest version of Ubuntu is 13.04 (Raring) but as mentioned above, this book was tested against Ubuntu 12.04 (Precise). For a list of Ubuntu versions that are officially compatible with ROS Hydro, see the ROS Installation Guide.

Ubuntu can be installed on an existing Windows machine or an Intel-based Mac and you'll probably want to leave those installations intact while installing Ubuntu alongside them. You'll then have a choice of operating systems to use at boot time. On the other

hand, if you have a spare laptop lying around that you can dedicate to your robot, then you can install Ubuntu over the entire disk. (It will also tend to run faster this way.) In general, it is not a good idea to install Ubuntu inside a virtual machine like VMware. While this is a great way to run through the ROS tutorials, the virtual machine will likely get bogged down when trying to run graphics intensive programs like `RViz`. (In fact, such programs might not run at all.)

Here are a few links to get you started:

- Installing Ubuntu 12.04 on a Windows computer
- Alternatively, you can also try the Wubi Windows installer. (Make sure you select 12.04 as the default selection is 12.10.) This installer runs under Windows and will install Ubuntu as if it were just another Windows application. This means you can uninstall it at a later time using Add/Remove Programs (Windows XP) or Programs and Features (Windows Vista and above). This is the easiest way to get Ubuntu up and running alongside Windows but there are a few caveats: (1) the Ubuntu disk size may be limited, (2) you will not be able to Hibernate the machine from within Linux, though you will still be able to put it in Standby, and (3) Linux will not run quite as fast as regular installation to a raw partition.
- Dual-booting Ubuntu/MacOS X on an Apple/Intel machine

To keep this book focused on ROS itself, please use Google or the Ubuntu support forums if you have any trouble with the installation.

3.2 Getting Started with Linux

If you are already a veteran Linux user, you are well ahead of the game. If not, you might want to run through a tutorial or two before proceeding. Since web tutorials come and go every day, you can simply Google something like "Ubuntu tutorial" to find something you like. However, bear in mind that most of your work with ROS will take place at the command line or in some form of text editor. A good place to start on command line basics is Using the Ubuntu Terminal. The text editor you choose for programming is up to you. Choices include gedit, nano, pico, emacs, vim, Eclipse and many others. (See for example https://help.ubuntu.com/community/Programming.) Programs like Eclipse are actually full featured IDEs and can be used to organize projects, test code, manage SVN and Git repositories and so on. For more information on using different IDEs with ROS, see http://wiki.ros.org/IDEs.

3.3 A Note about Updates and Upgrades

There is a growing tendency by major software developers to shorten their release cycle. Some packages like Firefox are now on a 6-week cycle.

There also seems to be a growing tendency for these rapid upgrade cycles to break code that worked just fine the day before. A lot of time can be wasted chasing the latest release and then fixing all your code to make it work again. It is generally a good idea to check the Change List for a given product *before* you upgrade and make sure there really is something you need. Otherwise, if it ain't broke...

Ubuntu currently uses a 6-month release cycle and ROS originally followed the same pattern. A recent poll of ROS developers and users found that a longer support cycle was desired. You can find the details on the Distributions page of the ROS Wiki.

4. Reviewing the ROS Basics

4.1 Installing ROS

The easiest and quickest way to install ROS on your Ubuntu Linux machine is to use the Debian packages rather than compiling from source. Well-tested instructions can be found at Ubuntu Install of Hydro. Be sure to choose the correct instructions for the version of the OS you are using (e.g. Ubuntu 12.04 versus 13.04). It is also recommended that you do the Desktop-Full Install.

PLEASE NOTE: This version of the book is written for and tested against **ROS Hydro.**

One advantage of using the Debian package install (rather than compiling from source) is that you'll get updates automatically through the Ubuntu Update Manager. You can also pick and choose additional ROS packages using the Synaptics Package Manager or the Ubuntu Software Center. If you need to compile the bleeding edge version of some package or stack from source, you can install it in your own personal ROS directory as explained below.

4.2 Installing rosinstall

The rosinstall utility is not part of the ROS desktop installation and so we need to get it separately. If you follow the full set of instructions on the Hydro installation page, you will have already completed these steps. However, many users tend to miss these final steps:

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

The last command is run as a normal user—i.e., without sudo.

NOTE: If you have used earlier versions of ROS on the same machine on which you have installed Hydro, there can be problems running the newer `python-rosinstall` package alongside older versions of `rosinstall` that may have been installed using `pip` or `easy_install`. For details on how to remove these older versions, see this answer on answers.ros.org.

4.3 Building ROS Packages with Catkin

All versions of ROS prior to Groovy used `rosbuild` for creating and building ROS stacks and packages. Beginning with Groovy, a second build system called `catkin` was introduced. At the same time, stacks were replaced with "meta packages." Since hundreds of `rosbuild`-style packages still exist, both systems will coexist for some time to come. However, it is a good idea to start practicing with catkin, especially if you want to release any of your own ROS packages as Ubuntu Debian packages since only catkin packages can be converted and released this way.

A number of step-by-step catkin tutorials exist on the ROS wiki and the reader is encouraged to work through at least the first four. In this chapter, we only provide a brief summary of the highlights. We will also cover how to work with both `catkin` and `rosbuild` packages at the same time.

If you followed the Ubuntu ROS installation instructions, all ROS stacks, packages and meta-packages will be found under `/opt/ros/release`, where `release` is the named release of ROS you are using; e.g. `/opt/ros/hydro`. This is a read-only part of the file system and should not be changed except through the package manager, so you'll want to create a personal ROS directory in your home directory so you can (1) install third-party ROS packages that don't have Debian versions and (2) create your own ROS packages.

4.4 Creating a catkin Workspace

In *Volume One* we stored all our `rosbuild` packages under the directory `~/ros_workspace`, although you may have chosen a different name and location on your own computer. In any event, `catkin` packages must live in their own directory, so the first place to start is to create this workspace. By convention, the directory name chosen for catkin packages is `catkin_ws` so we will use that here

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

Notice how we create both the top level directory `~/catkin_ws` and a subdirectory called `src`. Note also that we run the `catkin_init_workspace` command in the `src` directory.

Even though the current workspace is empty, we run `catkin_make` to create some initial directories and setup files. `catkin_make` is always run in the top-level catkin workspace folder (not in the `src` folder!):

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

NOTE: After building any new catkin package(s), be sure to source the `devel/setup.bash` file as follows:

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

This will ensure that ROS can find any new message types and Python modules belonging to the newly built package(s).

4.5 Doing a "make clean" with catkin

When using the older `rosmake` build system, you could use the command:

```
$  rosmake --target=clean
```

in a top level stack or package directory to remove all build objects. Unfortunately, this feature does not exist when using `catkin`. The only way to start with a clean slate, is to remove all build objects from *all* your `catkin` packages using the following commands:

CAUTION! Do **not** include the `src` directory in the `rm` command below or you will lose all your personal catkin source files!

```
$ cd ~/catkin_ws
$ \rm -rf devel build install
```

You would then remake any packages as usual:

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

4.6 Rebuilding a Single catkin Package

If you update a single package in your catkin workspace and want to re-build just that package, use the following variation of `catkin_make`:

```
$ cd ~/catkin_ws
$ catkin_make --pkg package_name
```

4.7 Mixing catkin and rosbuild Workspaces

If you have been using ROS for awhile, you probably already have a ROS workspace and packages that use the earlier `rosbuild` make system rather than `catkin`. You can continue to use these packages and `rosmake` while still using `catkin` for new packages.

Assuming you have followed the steps in the previous section, and that your `rosbuild` workspace directory is `~/ros_workspace`, run the following command to allow the two systems to work together:

```
$ rosws init ~/ros_workspace ~/catkin_ws/devel
```

Of course, change the directory names in the command above if you created your `rosbuild` and/or `catkin` workspaces in other locations.

NOTE: If you receive the following error when running the above command:

```
rosws: command not found
```

it means you have not installed the rosinstall files during your initial ROS installation. (It is the final step in the installation guide.) If that is the case, install `rosinstall` now:

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

And try the previous `rosws` command again.

With this step complete, edit your `~/.bashrc` file and change the line that looks like this:

```
source /opt/ros/hydro/setup.bash
```

to this:

```
source ~/ros_workspace/setup.bash
```

Again, change the directory name for your rosbuild workspace if necessary.

Save the changes to `~/.bashrc` and exit your editor.

To make the new combined workspace active immediately, run the command:

```
$ source ~/ros_workspace/setup.bash
```

New terminal windows will execute this command automatically from your `~/.bashrc` file.

4.8 Working through the Official ROS Tutorials

Before diving into the Beginner Tutorials on the ROS Wiki, it is highly recommended that you look first at the ROS Start Guide. Here you will find an overall introduction as well as an explanation of the key concepts, libraries and technical design.

The official ROS Beginner Tutorials are superbly written and have been tested by many new ROS users. It is therefore essential to begin your introduction to ROS by running through these tutorials in sequence. Be sure to actually run the code samples—don't just read through them. ROS itself is not difficult: while it might seem unfamiliar at first, the more practice you have with the code, the easier it becomes. Expect to spend at least a few days or weeks going through the tutorials.

Once you have completed the beginner tutorials, it is essential to also read the tf overview and do the tf Tutorials which will help you understand how ROS handles different frames of reference. For example, the location of an object within a camera image is usually given relative to a frame attached to the camera, but what if you need to know the location of the object relative to the base of the robot? The ROS `tf` library does most of the hard work for us and such frame transformations become relatively painless to carry out.

Finally, we will need to understand the basics of ROS actions when we get to the Navigation Stack. You can find an introduction to ROS actions in the actionlib tutorials.

You will notice that most of the tutorials provide examples in both Python and C++. All of the code in this book is written in Python, but if you are a C++ programmer, feel free to do the tutorials in C++ as the concepts are the same.

Many individual ROS packages have their own tutorials on the ROS Wiki pages. We will refer to them as needed throughout the book, but for now, the Beginner, tf, and actionlib tutorials will suffice.

4.9 RViz: The ROS Visualization Tool

If you have already been working with ROS for awhile, you are probably familiar with `RViz`, the all-seeing ROS visualization utility. However, since `RViz` is not covered in the standard ROS beginners tutorials, you might not have been formally introduced. Fortunately, there is already an RViz User's Guide that will take you step-by-step through its features. Please be sure to read through the guide before going further as we will make heavy use of `RViz` in several parts of the book.

`RViz` can sometimes be a little finicky about running on different graphics cards. If you find that `RViz` aborts during start up, first just try launching it again. Keep trying a few times if necessary. (On one of my computers, it is usually the third attempt that works.)

If that fails, check out the RViz Troubleshooting Guide for possible solutions. In particular, the section on Segfaults during startup tends to solve the most common problems.

4.10 Using ROS Parameters in your Programs

As you know from completing the Beginner Tutorials, ROS nodes store their configuration parameters on the ROS Parameter Server where they can be read and modified by other active nodes. You will often want to use ROS parameters in your own scripts so that you can set them in your launch files, override them on the command line, or modify them through rqt_reconfigure (formerly dynamic_reconfigure) if you add dynamic support. We will assume you already know how to use ROS parameters through the rospy.get_param() function which is explained fully in this tutorial on Using Parameters in rospy.

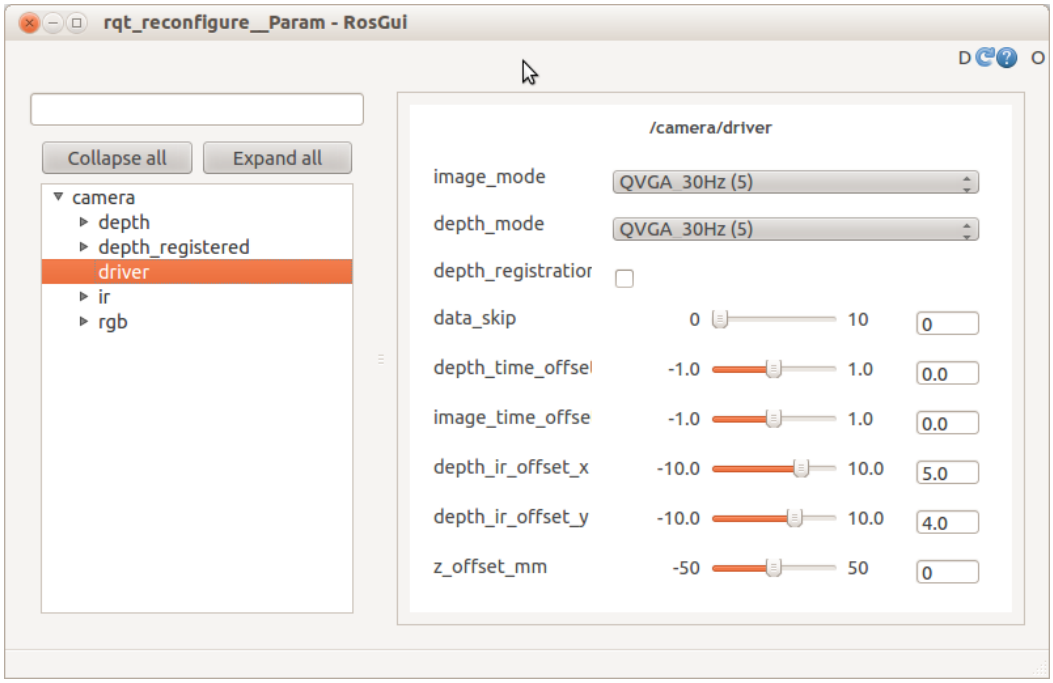
4.11 Using rqt_reconfigure (formerly dynamic_reconfigure) to set ROS Parameters

Being able to tweak ROS parameters on the fly is often useful for tuning or debugging a running application. As you will recall from the Services and Parameters Tutorial, ROS provides the command line tool rosparam for getting and setting parameters. However, parameter changes made this way will not be read by a node until the node is restarted.

The ROS rqt_reconfigure package (formerly called dynamic_reconfigure) provides an easy-to-use GUI interface to a subset of the parameters on the Parameter Server. It can be launched any time using the command:

```
$ rosrun rqt_reconfigure rqt_reconfigure
```

The image below shows how the GUI would look when configuring an OpenNI camera node that connects to a Kinect or Xtion Pro camera:



The `rqt_reconfigure` GUI allows you to change parameters for nodes dynamically—i.e., without having restart a node. However, there is one catch: only nodes that have been programmed using the `rqt_reconfigure` API will be visible in the `rqt_reconfigure` GUI. This includes most nodes in the key ROS stacks and packages such as Navigation, but many third-party nodes do not use the API and therefore can only be tweaked using the `rosparam` command line tool followed by a node restart.

NOTE: Unlike the `dynamic_reconfigure` package in previous ROS versions, `rqt_reconfigure` does not appear to dynamically detect new nodes if they are launched after you bring up the GUI. To see a freshly launched node in the `rqt_reconfigure` GUI, click the little blue refresh arrow in the upper right corner of the `rqt_reconfigure` window.

Adding `rqt_reconfigure` support to your own nodes is not difficult and if you would like to learn how, refer to the step-by-step Dynamic Reconfigure Tutorials on the ROS Wiki. (Yes, the tutorials still use the old name.) We will have the occasion to use the reconfigure GUI in several sections of the book. It would therefore be a good idea to familiarize yourself with its behavior by going through the `rqt_reconfigure` introduction on the ROS Wiki.

4.12 Networking Between a Robot and a Desktop Computer

A fairly typical setup when using ROS is to mount a laptop or single board computer on the robot while monitoring its actions on a desktop machine. ROS makes it relatively easy for multiple machines to view the same set of topics, services and parameters. This is particularly useful when your robot's onboard computer is not very powerful since it allows you to run the more demanding processes such as `RViz` on your desktop. (Programs like `RViz` can also run down a laptop battery very quickly.)

4.12.1 Time Synchronization

Time synchronization between machines is often critical in a ROS network since frame transformations and many message types are timestamped. An easy way to keep your computers synchronized is to install the Ubuntu `chrony` package on both your desktop and your robot. This package will keep your computer clocks synchronized with Internet servers and thus with each other.

To install `chrony`, run the command:

```
$ sudo apt-get install chrony
```

After installation, the `chrony` daemon will automatically start and begin synchronizing your computer's clock with a number of Internet servers.

4.12.2 ROS Networking using Zeroconf

Later versions of Ubuntu include support for Zeroconf, a technique that allows machines on the same subnet to reference each other using local hostnames instead of IP addresses. So you can use this method if your desktop computer and robot are connected to the same router on a home or office network—a common scenario for both hobby and research robots.

Use the `hostname` command to determine the short name of your machine. The output will be the name you chose during the original Ubuntu setup on that computer. For example, if you named your desktop computer "my_desktop", the output would look like this:

```
$ hostname
my_desktop
```

To get the Zeroconf hostname, simply add ".local" after the hostname so in this case, the Zeroconf hostname would be:

```
my_desktop.local
```

Next, run the `hostname` command on your robot's computer to get its hostname and add ".local" to get its Zeroconf name. Let's assume your robot's Zeroconf name is:

```
my_robot.local
```

Now we need to test to see if the machines can see each other on the network.

4.12.3 Testing Connectivity

Use the `ping` command to make sure you have basic connectivity between your two computers. From your desktop computer run the command:

```
$ ping my_robot.local
```

which should produce a result like the following:

```
PING my_robot.local (192.168.0.197) 56(84) bytes of data.
64 bytes from my_robot.local (192.168.0.197): icmp_req=1 ttl=64
time=1.65 ms
64 bytes from my_robot.local (192.168.0.197): icmp_req=2 ttl=64
time=0.752 ms
64 bytes from my_robot.local (192.168.0.197): icmp_req=3 ttl=64
time=1.69 ms
```

Type `Ctrl-C` to stop the test. The `icmp_req` variable counts the pings while the `time` variable indicates the round trip time in milliseconds. After stopping the test, you will a summary that looks like this:

```
--- my_robot.local ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2001ms
rtt min/avg/max/mdev = 0.752/1.367/1.696/0.436 ms
```

In general, you should see 0% packet loss and average delay times under about 5ms.

Now do the test in the other direction. Bring up a terminal on your robot (or use `ssh` if you know it already works) and ping your desktop:

```
$ ping my_desktop.local
```

Once again you should see 0% packet loss and short round trip times.

4.12.4 Setting the ROS_MASTER_URI and ROS_HOS TNAME Variables

In any ROS network, one machine is designated as the ROS master and it alone runs the `roscore` process. Other machines must then set the `ROS_MASTER_URI` environment

variable to point to the master host. Each computer must also set its ROS hostname appropriately as we will show.

In general, it does not matter which machine you choose to be the master. However, for a completely autonomous robot, you'll probably want to make the robot computer the master so that it does not depend in any way on the desktop.

If we want the robot to be the ROS master, we set its ROS_HOSTNAME to its Zeroconf name and run the `roscore` process:

On the robot:

```
$ export ROS_HOSTNAME=my_robot.local
$ roscore
```

Next, move to your desktop, set the `ROS_HOSTNAME` to its Zeroconf name and then set the `ROS_MASTER_URI` variable to point to your robot's Zeroconf URI.

On the desktop:

```
$ export ROS_HOSTNAME=my_desktop.local
$ export ROS_MASTER_URI=http://my_robot.local:11311
```

For an added check on time synchronization, we can run the `ntpdate` command to synchronize the desktop with the robot.

On the desktop:

```
$ sudo ntpdate -b my_robot.local
```

If all goes well, you should be able to see the `/rosout` and `/rosout_agg` topics on your desktop as follows:

```
$ rostopic list
```

```
/rosout
/rosout_agg
```

4.12.5 Opening New Terminals

In any new terminal window you open on your desktop or robot, you need to set the `ROS_HOSTNAME` variable to that machine's Zeroconf name. And for new desktop terminals, you also have to set the `ROS_MASTER_URI` to point to the robot. (Or more generally, on any non-master computer, new terminals must set the `ROS_MASTER_URI` to point to the master computer.)

If you will use the same robot and desktop for a while, you can save yourself some time by adding the appropriate export lines to the end of the `~/.bashrc` file on each computer. If the robot will always be the master, add the following line to the end of its `~/.bashrc` file:

```
export ROS_HOSTNAME=my_robot.local
```

And on the desktop, add the following two lines to the end of its `~/.bashrc` file:

```
export ROS_HOSTNAME=my_desktop.local
export ROS_MASTER_URI=http://my_robot.local:11311
```

(Of course, replace the Zeroconf names with those that match your setup.)

You can also set your desktop up as the ROS master instead of the robot. In this case, simply reverse the roles and Zeroconf hostnames in the examples above.

4.12.6 Running Nodes on both Machines

Now that you have your ROS network set up between your robot and desktop computer, you can run ROS nodes on either machine and both will have access to all topics and services.

While many nodes and launch files can be run on either computer, the robot's startup files must always be run on robot since these nodes provide drivers to the robot's hardware. This includes the drivers for the robot base and any cameras, laser scanners or other sensors you want to use. On the other hand, the desktop is a good place to run `RViz` since it is very CPU-intensive and besides, you'll generally want to monitor your robot from your desktop anyway.

Since the robot's computer may not always have a keyboard and monitor, you can use `ssh` to log into your robot and launch driver nodes from your desktop. Here's an example of how you might do this.

From your desktop computer, use `ssh` to log in to your robot.

On the desktop:

```
$ ssh my_robot.local
```

Once logged in to the robot, fire up roscore and your robot's startup launch file(s).

On the robot (via ssh):

```
$ export ROS_HOSTNAME=my_robot.local
$ roscore &
$ roslaunch my_robot startup.launch
```

(You can omit the first `export` line above if you have already included it in the robot's `~/.bashrc` file.)

Notice how we send the `roscore` process into the background using the `&` symbol after the command. This brings back the command prompt so we can launch our robot's startup file without having to open another `ssh` session. If possible, launch all your robot's hardware drivers in one `startup.launch` file (it can be named anything you like). This way you will not have to open additional terminals to launch other drivers.

Back on your desktop, open another terminal window, set the `ROS_MASTER_URI` to point to your robot, then fire up `RViz`:

On the desktop:

```
$ export ROS_HOSTNAME=my_desktop.local
$ export ROS_MASTER_URI=http://my_robot.local:11311
$ rosrun rviz rviz -d `rospack find rbx1_nav`/nav.rviz
```

(You can omit the two `export` lines if you have already included them in the desktop's `~/.bashrc` file.)

Here we are running `RViz` with one of the configuration files included in the ros-by-example navigation package but you can also simply launch `RViz` without any configuration file.

4.12.7 ROS Networking across the Internet

While outside the scope of this book, setting up ROS nodes to communicate over the Internet is similar to the instructions given above using Zeroconf. The main difference is that now you need to use fully qualified hostnames or IP addresses instead of local Zeroconf names. Furthermore, it is likely that one or more of the machines will be behind a firewall so that some form of VPN (e.g. OpenVPN) will have to be set up. Finally, since most machines on the ROS network will be connected to a local router (e.g. wifi access point), you will need to set up port forwarding on that router or use dynamic DNS. While all this is possible, it is definitely not trivial to set up.

4.13 ROS Recap

Since it might have been awhile since you did the Beginner and tf Tutorials, here is a brief recap of the primary ROS concepts. The core entity in ROS is called a node. A node is generally a small program written in Python or C++ that executes some relatively simple task or process. Nodes can be started and stopped independently of one another and they communicate by passing messages. A node can publish messages on certain topics or provide services to other nodes.

For example, a publisher node might report data from sensors attached to your robot's microcontroller. A message on the `/head_sonar` topic with a value of 0.5 would mean that the sensor is currently detecting an object 0.5 meters away. (Remember that ROS uses meters for distance and radians for angular measurements.) Any node that wants to know the reading from this sensor need only subscribe to the `/head_sonar` topic. To make use of these values, the subscriber node defines a callback function that gets executed whenever a new message arrives on the subscribed topic. How often this happens depends on the rate at which the publisher node updates its messages.

A node can also define one or more services. A ROS service produces some behavior or sends back a reply when sent a request from another node. A simple example would be a service that turns an LED on or off. A more complex example would be a service that returns a navigation plan for a mobile robot when given a goal location and the starting pose of the robot.

Higher level ROS nodes will subscribe to a number of topics and services, combine the results in a useful way, and perhaps publish messages or provide services of their own. For example, the object tracker node we will develop later in the book subscribes to camera messages on a set of video topics and publishes movement commands on another topic that are read by the robot's base controller to move the robot in the appropriate direction.

4.14 What is a ROS Application?

If you are not already familiar with a publish/subscribe architecture like ROS, programming your robot to do something useful might seem a little mysterious at first. For instance, when programming an Arduino-based robot using C, one usually creates a single large program that controls the robot's behavior. Moreover, the program will usually talk directly to the hardware, or at least, to a library specifically designed for the hardware you are using.

When using ROS, the first step is to divide up the desired behavior into independent functions that can be handled by separate nodes. For example, if your robot uses a webcam or a depth camera like a Kinect or Xtion Pro, one node will connect to the camera and simply publish the image and/or depth data so that other nodes can use it. If your robot uses a mobile base, a base controller node will listen for motion commands

on some topic and control the robot's motors to move the robot accordingly. These nodes can be used without modification in many different applications whenever the desired behavior requires vision and/or motion control.

An example of a complete application is the "follower" application we will develop later in the book. (The original C++ version by Tony Pratkanis can be found in the `turtlebot_follower` package.) The goal of the follower app is to program a Kinect-equipped robot like the TurtleBot to follow the nearest person. In addition to the camera and base controller nodes, we need a third node that subscribes to the camera topic and publishes on the motion control topic. This "follower" node must process the image data (using OpenCV or PCL for example) to find the nearest person-like object, then command the base to steer in the appropriate direction. One might say that the follower node is our ROS application; however, to be more precise, the application really consists of all three nodes running together. To run the application, we use a ROS launch file to fire up the whole collection of nodes as a group. Remember that launch files can also include other launch files allowing for even easier reuse of existing code in new applications.

Once you get used to this style of programming, there are some significant advantages. As we have already mentioned, many nodes can be reused without modification in other applications. Indeed, some ROS applications are little more than launch files combining existing nodes in new ways or using different values for the parameters. Furthermore, many of the nodes in a ROS application can run on different robots without modification. For example, the TurtleBot follower application can run on any robot that uses a depth camera and a mobile base. This is because ROS allows us to abstract away the underlying hardware and work with more generic messages instead.

Finally, ROS is a network-centric framework. This means that you can distribute the nodes of your application across multiple machines as long as they can all see each other on the network. For example, while the camera and motor control nodes have to run on the robot's computer, the follower node and `RViz` could run on any machine on the Internet. This allows the computational load to be distributed across multiple computers if necessary.

4.15 Installing Packages with SVN, Git, and Mercurial

Once in awhile the ROS package you need won't be available as a Debian package and you will need to install it from source. There are three major source control systems popular with code developers: SVN, Git and Mecurial. The type of system used by the developer determines how you install the source. To make sure you are ready for all three systems, run the following install command on your Ubuntu machine:

```
$ sudo apt-get install git subversion mercurial
```

For all three systems, there are two operations you will use most of the time. The first operation allows you to check out the software for the first time, while the second is used for getting updates that might be available later on. Since these commands are different for all three systems, let's look at each in turn.

4.15.1 SVN

Let's assume that the SVN source you would like to check out is located at **http://repository/svn/package_name**. To do the initial checkout and build the package in your personal catkin directory, run the following commands. (If necessary, change the first command to reflect the actual location of your catkin source directory.)

```
$ cd ~/catkin_ws/src
$ svn checkout http://repository/svn/package_name
$ cd ~/catkin_ws
$ catkin_make
$ source devel/setup.bash
```

To update the package later on, run the commands:

```
$ cd ~/catkin_ws/src/package_name
$ svn update
$ cd ~/catkin_ws
$ catkin_make
$ source devel/setup.bash
```

4.15.2 Git

Let's assume that the Git source you would like to check out is located at **git://repository/package_name**. To do the initial checkout and build the package in your personal catkin directory, run the following commands. (If necessary, change the first command to reflect the actual location of your personal catkin source directory.)

```
$ cd ~/catkin_ws/src
$ git clone git://repository/package_name
$ cd ~/catkin_ws
$ catkin_make
$ source devel/setup.bash
```

To update the package later on, run the commands:

```
$ cd ~/catkin_ws/src/package_name
$ git pull
$ cd ~/catkin_ws
$ catkin_make
$ source devel/setup.bash
```

4.15.3 Mercurial

Let's assume that the Mercurial source you'd like to check out is located at **http://repository/package_name**. To do the initial checkout and build the package in your personal catkin directory, run the following commands. (If necessary, change the first command to reflect the actual location of your personal catkin source directory.)

```
$ cd ~/catkin_ws/src
$ hg clone http://repository/package_name
$ cd ~/catkin_ws
$ catkin_make
$ source devel/setup.bash
```

(In case you are wondering why Mercurial uses `hg` for its main command name, Hg is the symbol for the element Mercury on the Periodic Table in chemistry.) To update the package later on, run the commands:

```
$ cd ~/catkin_ws/src/package_name
$ hg update
$ cd ~/catkin_ws
$ catkin_make
$ source devel/setup.bash
```

4.16 Removing Packages from your Personal catkin Directory

To remove a package installed in your personal catkin directory, first remove the entire package source directory or move it to a different location outside of your `ROS_PACKAGE_PATH`. For example, to remove a package called `my_catkin_package` from your `~/catkin_ws/src` directory, run the commands:

```
$ cd ~/catkin_ws/src
$ \rm -rf my_catkin_package
```

You also have to remove all catkin build objects. Unfortunately, there is no (easy) way to do this for just the package you removed—you have to remove all build objects for all packages and then rerun `catkin_make`:

CAUTION! Do **not** include the `src` directory in the `rm` command below or you will lose all your personal catkin source files!

```
$ cd ~/catkin_ws
$ \rm -rf devel build install
$ catkin_make
$ source devel/setup.bash
```

You can test that the package has been removed using the `roscd` command:

```
$ roscd my_ros_package
```

which should produce the output:

```
roscd: No such package 'my_ros_package'
```

4.17 How to Find Third-Party ROS Packages

Sometimes the hardest thing to know about ROS is what's available from other developers. For example, suppose you are interested in running ROS with an Arduino and want to know if someone else has already created a ROS package to do the job. There are a few ways to do the search.

4.17.1 Searching the ROS Wiki

The ROS Wiki at wiki.ros.org includes a searchable index to many ROS packages and stacks. If a developer has created some ROS software they'd like to share with others, they tend to post an announcement to the `ros-users` mailing list together with the link to their repository. If they have also created documentation on the ROS Wiki, the package should show up in a search of the index shortly after the announcement.

The end result is that you can often find what you are looking for by simply doing a keyword search on the ROS Wiki. Coming back to our Arduino example, if we type "Arduino" (without the quotes) into the Search box, we find links referring to two packages: rosserial_arduino and ros_arduino_bridge.

4.17.2 Using the `roslocate` Command

If you know the exact package name you are looking for and you want to find the URL to the package repository, use the `roslocate` command. (This command is only available if you installed `rosinstall` as described earlier.) For example, to find the location of the `ros_arduino_bridge` package for ROS Groovy, run the command:

```
$ roslocate uri ros_arduino_bridge
```

which should yield the result:

```
Using ROS_DISTRO: hydro
Not found via rosdistro - falling back to information provided by
rosdoc
https://github.com/hbrobotics/ros_arduino_bridge.git
```

This means that we can install the package into our personal catkin directory using the `git` command:

```
$ cd ~/catkin_ws/src
$ git clone git://github.com/hbrobotics/ros_arduino_bridge.git
$ cd ~/catkin_ws
$ catkin_make
$ source devel/setup.bash
```

NOTE: Starting with ROS Groovy, the `roslocate` command will only return a result if the package or stack has be submitted to the indexer by the package maintainer for the ROS distribution you are currently using. If it has only be indexed for an earlier release (e.g. Electric or Fuerte) you will get a result like the following:

```
$ roslocate uri cob_people_perception
```

```
error contacting
http://ros.org/doc/groovy/api/cob_people_perception/stack.yaml:
HTTP Error 404: Not Found
error contacting
http://ros.org/doc/groovy/api/cob_people_perception/manifest.yaml
:
HTTP Error 404: Not Found
cannot locate information about cob_people_perception
```

To see if the `cob_people_perception` package has been indexed for Electric (for example), try the `--distro` option with `roslocate`:

```
$ roslocate uri --distro=electric cob_people_perception
```

which returns the result:

```
Not found via rosdistro - falling back to information provided by
rosdoc
https://github.com/ipa320/cob_people_perception.git
```

4.17.3 Browsing the ROS Software Index

To browse through the complete list of ROS packages, stacks and repositories as indexed on the ROS Wiki, click on the Browse Software link found in the banner at the top of every Wiki page.

4.17.4 Doing a Google Search

If all else fails, you can always try a regular Google search. For example, searching for "ROS face recognition package" returns a link to a face recognition package at http://wiki.ros.org/face_recognition.

4.18 Getting Further Help with ROS

There are several sources for additional help with ROS. Probably the best place to start is at the main ROS wiki at http://wiki.ros.org. As described in the previous section, be sure to use the Search box at the top right of the page.

If you can't find what you are looking for on the Wiki, try the ROS Questions and Answers forum at http://answers.ros.org. The answers site is a great place to get help. You can browse through the list of questions, do keyword searches, look up topics based on tags, and even get email notifications when a topic is updated. But be sure to do some kind of search before posting a new question to avoid duplication.

Next, you can search one of the ROS mailing list archives:

- ros-users: for general ROS news and announcements
- ros-kinect: for Kinect related issues
- pcl-users: for PCL related issues

NOTE: Please do not use the `ros-users` mailing list to post questions about using ROS or debugging packages. Use http://answers.ros.org instead.

If you want to subscribe to one or more of these lists, use the appropriate link listed below:

- ros-users: Subscription page
- ros-kinect – Subscription page
- pcl_users - Subscription page

5. Installing the ros-by-example Code

5.1 Installing the Prerequisites

Before installing the ROS By Example code itself, it will save some time if we install most of the additional ROS packages we will need later. (Instructions will also be provided for installing individual packages as needed throughout the book.) Simply copy and paste the following command (without the $ sign) into a terminal window to install the Debian packages we will need. The \ character at the end of each line makes the entire block appear like a single line to Linux when doing a copy-and-paste:

```
$ sudo apt-get install ros-hydro-turtlebot-* \
ros-hydro-openni-camera ros-hydro-openni-launch \
ros-hydro-openni-tracker ros-hydro-laser-* \
ros-hydro-audio-common ros-hydro-joystick-drivers \
ros-hydro-orocos-kdl ros-hydro-python-orocos-kdl \
ros-hydro-dynamixel-motor-* ros-hydro-pocketsphinx \
gstreamer0.10-pocketsphinx python-setuptools python-rosinstall \
ros-hydro-opencv2 ros-hydro-vision-opencv \
ros-hydro-depthimage-to-laserscan ros-hydro-arbotix-* \
git subversion mercurial
```

5.2 Cloning the Hydro ros-by-example Repository

IMPORTANT: If you have installed previous versions of the `ros-by-example` repository for ROS Electric, Fuerte or Groovy, follow the appropriate instructions below to replace or override your installation with the Hydro version. Note that all three previous versions of the `ros-by-example` code used the `rosbuild` system whereas the Hydro version now uses `catkin`. If this is your first time installing the `ros-by-example code`, you can skip the first two sections and go straight to section 5.2.3.

5.2.1 Upgrading from Electric or Fuerte

The `ros-by-example` stack for ROS Electric and Fuerte is distributed as an SVN repository called `rbx_vol_1` on Google Code. Either remove the this old stack from your `~/ros_workspace` directory or, if you've made modifications you don't want to lose, move the old `rbx_vol_1` directory out of your ROS_PACKAGE_PATH before installing the new repository.

5.2.2 Upgrading from Groovy

The `ros-by-example` packages for ROS Groovy and Hydro are distributed as a Git repository called `rbx1` on GitHub. The default branch is called `groovy-devel` and it is a `rosbuild` version of the repository used with the Groovy version of the book. For ROS Hydro, you need to use the `hydro-devel` branch of the repository which as been converted to the newer `catkin` build system. This means you also need to install the code in your personal `~/catkin_ws/src` directory rather than `~/ros_workspace`.

If you have been using the Groovy version of the `rbx1` code, either delete the old repository from your `~/ros_workspace` directory or, if you've made modifications you don't want to lose, move the old `rbx1` directory out of your ROS_PACKAGE_PATH before installing the new repository.

5.2.3 Cloning the rbx1 repository for Hydro

To clone the `rbx1` repository for Hydro, follow these steps:

```
$ cd ~/catkin_ws/src
$ git clone https://github.com/pirobot/rbx1.git
$ cd rbx1
$ git checkout hydro-devel
$ cd ~/catkin_ws
$ catkin_make
$ source ~/catkin_ws/devel/setup.bash
```

NOTE 1: The fourth command above (`git checkout hydro-devel`) is critical—this is where you select the Hydro branch of the repository. (By default, the clone operation checks out the Groovy branch for the benefit of those still using Groovy.)

NOTE 2: The last line above should be added to the end of your `~/.bashrc` file if you haven't done so already. This will ensure that your `catkin` packages are added to your ROS_PACKAGE_PATH whenever you open a new terminal. If you will be running a mix of `rosbuild` and `catkin` packages, refer back to section 4.7 in Chapter 4 on how to set up your `~/.bashrc` file so that both types of packages can be found.

If the ROS By Example code is updated at a later time, you can merge the updates with your local copy of the repository by using the following commands:

```
$ cd ~/catkin_ws/src/rbx1
$ git pull
$ cd ~/catkin_ws
$ catkin_make
$ source devel/setup.bash
```

Staying Up-To-Date: If you'd like to receive notifications of updates to both the book and the accompanying code, please join the ros-by-example Google Group.

All of the ROS By Example packages begin with the letters `rbx1`. To list the packages, move into the parent of the `rbx1` meta-package and use the Linux `ls` command:

```
$ roscd rbx1
$ cd ..
$ ls -F
```

which should result in the following listing:

```
rbx1/       rbx1_bringup/      rbx1_dynamixels/    rbx1_nav/     rbx1_vision/
rbx1_apps/  rbx1_description/  rbx1_experimental/  rbx1_speech/  README.md
```

Throughout the book we will be using the `roscd` command to move from one package to another. For example, to move into the `rbx1_speech` package, you would use the command:

```
$ roscd rbx1_speech
```

Note that you can run this command from any directory and ROS will find the package.

IMPORTANT: If you are using two computers to control or monitor your robot, such as a laptop on the robot together with a second computer on your desktop, be sure to clone and build the Hydro branch of the `rbx1` repository on both machines.

5.3 About the Code Listings in this Book

For the benefit of those who would like to work from a printed version of the book instead of or in addition to the PDF version, most of the sample programs are displayed in their entirety as well as broken down line-by-line for analysis. At the top of each sample script is a link to the online version of the file as found in the ROS By Example repository. If you are reading the PDF version of the book, clicking on the link will bring you to a nicely formatted and color coded version of the program. Be aware though that the line numbers in the online version will not match those in the book since the printed version leaves out some extraneous comments to save space and make things easier to read. Of course, once you have downloaded the code from the ROS By Example repository, you can also bring up your local copy of the same files in your favorite editor.

As with most programming code, there is usually more than one way to solve a problem. The sample programs described here are no exception and if you can think of a better way to accomplish the same goal, great. The code included in the ROS By Example repository is meant only as a guide and certainly does not represent the best or only way to accomplish a given task using ROS.

6. Installing the Arbotix Simulator

To test our code on a simulated robot, we will use the `arbotix_python` simulator found in the ArbotiX stack by Michael Ferguson.

6.1 Installing the Simulator

To install the simulator in your personal ROS directory, run the following commands:

```
$ sudo apt-get install ros-hydro-arbotix-*
```

NOTE: Be sure to remove any earlier versions of the arbotix stack that you might have installed using SVN as was done in the Electric and Fuerte versions of this book.

Finally, run the command:

```
$ rospack profile
```

6.2 Testing the Simulator

To make sure everything is working, make sure `roscore` is running, then launch the simulated TurtleBot as follows:

```
$ roslaunch rbx1_bringup fake_turtlebot.launch
```

which should result in the following startup messages:

```
process[arbotix-1]: started with pid [4896]
process[robot_state_publisher-2]: started with pid [4897]
[INFO] [WallTime: 1338681385.068539] ArbotiX being simulated.
[INFO] [WallTime: 1338681385.111492] Started DiffController
(base_controller). Geometry: 0.26m wide, 4100.0 ticks/m.
```

To use a model of Pi Robot instead, run the command:

```
$ roslaunch rbx1_bringup fake_pi_robot.launch
```

Next, bring up `RViz` so we can observe the simulated robot in action:

```
$ rosrun rviz rviz -d `rospack find rbx1_nav`/sim.rviz
```

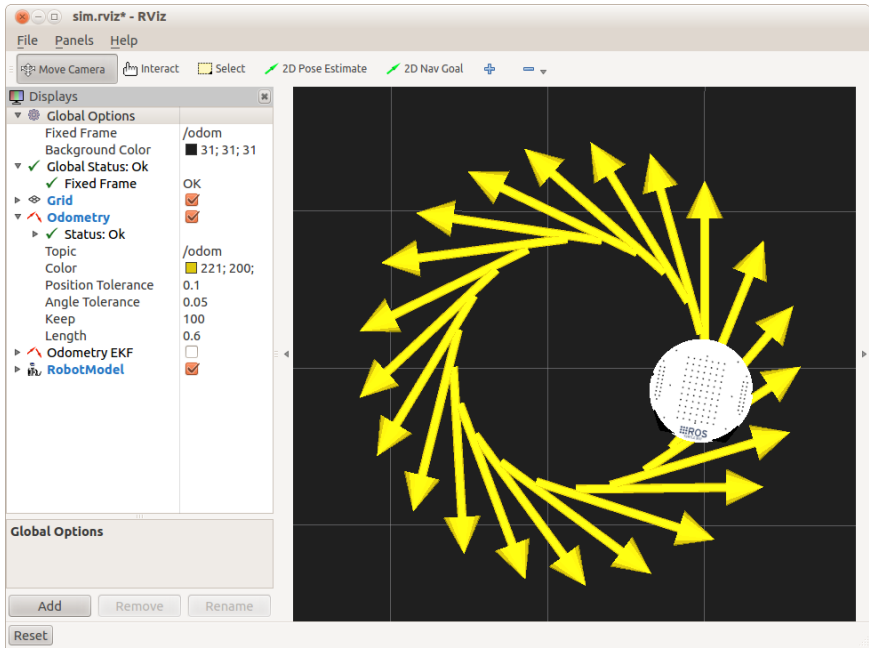
(Note how we use the Linux backtick operator together with the `rospack find` command to locate the `rbx1_nav` package without having to type the entire path.)

If everything went right, you should see the TurtleBot (or Pi Robot) in `RViz`. (The default view is top down. To change views, click on the **Panels** menu in `RViz` and select the **Views** menu item.)

To test the simulation, open another terminal window or tab and run the following command which should cause the simulated robot to move in a counter-clockwise circle:

```
$ rostopic pub -r 10 /cmd_vel geometry_msgs/Twist '{linear: {x: 0.2, y:
  0, z: 0}, angular: {x: 0, y: 0, z: 0.5}}'
```

The view in `RViz` should look something like this. (The image below was zoomed using the mouse scroll wheel.)



To stop the rotation, type `Ctrl-C` in the same terminal window, then publish the empty `Twist` message:

```
$ rostopic pub -1 /cmd_vel geometry_msgs/Twist '{}'
```

6.3 Running the Simulator with Your Own Robot

If you have a URDF model of your own robot, you can run it in the ArbotiX simulator instead of the TurtleBot or Pi Robot. First, make a copy of the fake TurtleBot launch file:

```
$ roscd rbx1_bringup/launch
$ cp fake_turtlebot.launch fake_my_robot.launch
```

Then bring up your launch file in your favorite editor. At first it will look like this:

```
<launch>
  <param name="/use_sim_time" value="false" />

  <!-- Load the URDF/Xacro model of our robot -->
  <arg name="urdf_file" default="$(find xacro)/xacro.py '$(find
rbx1_description)/urdf/turtlebot.urdf.xacro'" />

  <param name="robot_description" command="$(arg urdf_file)" />

  <node name="arbotix" pkg="arbotix_python" type="arbotix_driver"
output="screen">
      <rosparam file="$(find rbx1_bringup)/config/fake_turtlebot_arbotix.yaml"
command="load" />
      <param name="sim" value="true"/>
  </node>

  <node name="robot_state_publisher" pkg="robot_state_publisher"
type="state_publisher">
      <param name="publish_frequency" type="double" value="20.0" />
  </node>

</launch>
```

As you can see, the URDF model is loaded near the top of the file. Simply replace the package and path names to point to your own URDF/Xacro file. You can leave most of the rest of the launch file the same. The result would look something like this:

```
<launch>
  <!-- Load the URDF/Xacro model of our robot -->
  <arg name="urdf_file" default="$(find xacro)/xacro.py '$(find
YOUR_PACKAGE_NAME)/YOUR_URDF_PATH'" />

  <param name="robot_description" command="$(arg urdf_file)" />

  <node name="arbotix" pkg="arbotix_python" type="arbotix_driver"
output="screen">
      <rosparam file="$(find rbx1_bringup)/config/fake_turtlebot_arbotix.yaml"
command="load" />
      <param name="sim" value="true"/>
  </node>

  <node name="robot_state_publisher" pkg="robot_state_publisher"
type="state_publisher">
      <param name="publish_frequency" type="double" value="20.0" />
```

```
  </node>
</launch>
```

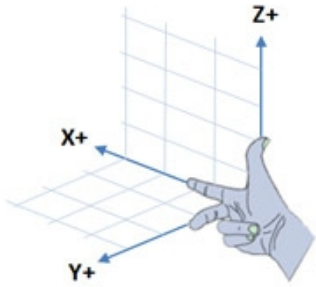
If your robot has an arm or a pan-and-tilt head, you can start with the `fake_pi_robot.launch` file as a template.

7. Controlling a Mobile Base

In this chapter we will learn how to control a mobile base that uses a pair of differential drive wheels and a passive caster wheel for balance. ROS can also be used to control an omni-directional base as well as flying robots or underwater vehicles but a land-based differential drive robot is a good place to start.

7.1 Units and Coordinate Systems

Before we can send movement commands to our robot, we need to review the measurement units and coordinate system conventions used in ROS.



When working with reference frames, keep in mind that ROS uses a right-hand convention for orienting the coordinate axes as shown on left. The index and middle fingers point along the positive x and y axes and the thumb points in the direction of the positive z axis. The direction of a rotation about an axis is defined by the right-hand rule shown on the right: if you point your thumb in the positive direction of any axis, your fingers curl in the direction of a positive rotation. For a mobile robot using ROS, the x-axis points forward, the y-axis points to the left and the z-axis points upward. Under the right-hand rule, a positive rotation of the robot about the z-axis is counterclockwise while a negative rotation is clockwise.

Remember also that ROS uses the metric system so that linear velocities are always specified in meters per second (m/s) and angular velocities are given in radians per second (rad/s). A linear velocity of 0.5 m/s is actually quite fast for an indoor robot (about 1.1 mph) while an angular speed of 1.0 rad/s is equivalent to about one rotation in 6 seconds or 10 RPM. When in doubt, start slowly and gradually increase speed. For an indoor robot, I tend to keep the maximum linear speed at or below 0.2 m/s.

7.2 Levels of Motion Control

Controlling a mobile robot can be done at a number of levels and ROS provides methods for most of them. These levels represent different degrees of abstraction, beginning with direct control of the motors and proceeding upward to path planning and SLAM (Simultaneous Localization and Mapping).

7.2.1 Motors, Wheels, and Encoders

Most differential drive robots running ROS use encoders on the drive motors or wheels. An encoder registers a certain number of ticks (usually hundreds or even thousands) per revolution of the corresponding wheel. Knowing the diameter of the wheels and the distance between them, encoder ticks can be converted to the distance traveled in meters or the angle rotated in radians. To compute speed, these values are simply divided by the time interval between measurements.

This internal motion data is collectively known as **odometry** and ROS makes heavy use of it as we shall see. It helps if your robot has accurate and reliable encoders but wheel data can be augmented using other sources. For example, the original TurtleBot uses a single-axis gyro to provide an additional measure of the robot's rotational motion since the iRobot Create's encoders are notably inaccurate during rotations.

It is important to keep in mind that no matter how many sources of odometry data are used, the *actual* position and speed of the robot in the world can (and probably will) differ from the values reported by the odometry. The degree of discrepancy will vary depending on the environmental conditions and the reliability of the odometry sources.

7.2.2 Motor Controllers and Drivers

At the lowest level of motion control we need a driver for the robot's motor controller that can turn the drive wheels at a desired speed, usually using internal units such as encoder ticks per second or a percentage of max speed. With the exception of the Willow Garage PR2 and TurtleBot, the core ROS packages do not include drivers for specific motor controllers. However, a number of third-party ROS developers have published drivers for some of the more popular controllers and/or robots such as the Arduino, ArbotiX, Serializer, Element, LEGO® NXT and Rovio. (For a more complete list of supported platforms, see Robots Using ROS.)

7.2.3 The ROS Base Controller

At the next level of abstraction, the desired speed of the robot is specified in real-world units such as meters and radians per second. It is also common to employ some form of PID control. PID stands for "Proportional Integral Derivative" and is so-named because the control algorithm corrects the wheel velocities based not only on the difference (proportional) error between the actual and desired velocity, but also on the derivative and integral over time. You can learn more about PID control on Wikipedia. For our purposes, we simply need to know that the controller will do its best to move the robot in the way we have requested.

The driver and PID controller are usually combined inside a single ROS node called the **base controller**. The base controller must always run on a computer attached directly to the motor controller and is typically one of the first nodes launched when bringing up

the robot. A number of base controllers can also be simulated in Gazebo including the TurtleBot, PR2 and Erratic.

The base controller node typically publishes odometry data on the `/odom` topic and listens for motion commands on the `/cmd_vel` topic. At the same time, the controller node typically (but not always) publishes a transform from the `/odom` frame to the base frame—either `/base_link` or `/base_footprint`. We say "not always" because some robots like the TurtleBot, uses the `robot_pose_ekf` package to combine wheel odometry and gyro data to get a more accurate estimate of the robot's position and orientation. In this case, it is the `robot_pose_ekf` node that publishes the transform from `/odom` to `/base_footprint`. (The `robot_pose_ekf` package implements an Extended Kalman Filter as you can read about on the Wiki page linked to above.)

Once we have a base controller for our robot, ROS provides the tools we need to issue motion commands either from the command line or by using other ROS nodes to publish these commands based on a higher level plan. At this level, it does not matter what hardware we are using for our base controller: our programming can focus purely on the desired linear and angular velocities in real-world units and any code we write should work on any base controller with a ROS interface.

7.2.4 Frame-Base Motion using the move_base ROS Package

At the next level of abstraction, ROS provides the `move_base` package that allows us to specify a target position and orientation of the robot with respect to some frame of reference; `move_base` will then attempt to move the robot to the goal while avoiding obstacles. The `move_base` package is a very sophisticated path planner and combines odometry data with both local and global cost maps when selecting a path for the robot to follow. It also controls the linear and angular velocities and accelerations automatically based on the minimum and maximum values we set in the configuration files.

7.2.5 SLAM using the gmapping and amcl ROS Packages

At an even higher level, ROS enables our robot to create a map of its environment using the SLAM `gmapping` package. The mapping process works best using a laser scanner but can also be done using a Kinect or Asus Xtion depth camera to provide a simulated laser scan. If you own a TurtleBot, the TurtleBot stack includes all the tools you need to do SLAM.

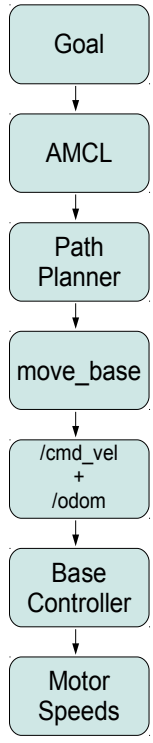
Once a map of the environment is available, ROS provides the `amcl` package (adaptive Monte Carlo localization) for automatically localizing the robot based on its current scan and odometry data. This allows the operator to point and click on any location on a map and the robot will find its way there while avoiding obstacles. (For a superb introduction to the mathematics underlying SLAM, check out Sebastian Thrun's online Artificial Intelligence course on Udacity.)

7.2.6 Semantic Goals

Finally, at the highest level of abstraction, motion goals are specified semantically such as "go to the kitchen and bring me a beer", or simply, "bring me a beer". In this case, the semantic goal must be parsed and translated into a series of actions. For actions requiring the robot to move to a particular location, each location can be passed to the localization and path planning levels for implementation. While beyond the scope of this volume, a number of ROS packages are available to help with this task including `smach`, `executive_teer`, `worldmodel`, `semantic_framer`, and `knowrob`.

7.2.7 Summary

In summary, our motion control hierarchy looks something like this:



In this chapter and the next, we will learn how to use these levels of motion control. But before we can understand the more powerful features provided by `move_base`, `gmapping` and `amcl`, we need to start with the basics.

7.3 Twisting and Turning with ROS

ROS uses the `Twist` message type (see details below) for publishing motion commands to be used by the base controller. While we could use almost any name for a topic, it is usually called `/cmd_vel` which is short for "command velocities". The base controller node subscribes to the `/cmd_vel` topic and translates `Twist` messages into motor signals that actually turn the wheels.

To see the components of a `Twist` message, run the command:

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

which will produce the following output:

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

As you can see, the `Twist` message is composed of two sub-messages with type `Vector3`, one for the x, y and z linear velocity components and another for the x, y and z angular velocity components. Linear velocities are specified in meters per second and angular velocities are given in radians per second. (1 radian equals approximately 57 degrees.)

For a differential drive robot operating in a two-dimensional plane (such as the floor), we only need the linear x component and the angular z component. This is because this type of robot can only move forward/backward along its longitudinal axis and rotate only around its vertical axis. In other words, the linear y and z components are always zero (the robot cannot move sideways or vertically) and the angular x and y components are always zero since the robot cannot rotate about these axes. An omni-directional robot would also use the linear y component while a flying or underwater robot would use all six components.

7.3.1 Example Twist Messages

Suppose we want the robot to move straight ahead with a speed of 0.1 meters per second. This would require a `Twist` message with linear values `x=0.1`, `y=0` and `z=0`, and angular values `x=0`, `y=0` and `z=0`. If you were to specify this `Twist` message on the command line, the message part would take the form:

```
'{linear: {x: 0.1, y: 0, z: 0}, angular: {x: 0, y: 0, z: 0}}'
```

Notice how we delineate the sub-messages using braces and separate the component names from their values with a colon and a space (the space is required!) While this might seem like a lot of typing, we will rarely control the robot this way. As we shall see later on, `Twist` messages will be sent to the robot using other ROS nodes.

To rotate counterclockwise with an angular velocity of 1.0 radians per second, the required `Twist` message would be:

```
'{linear: {x: 0, y: 0, z: 0}, angular: {x: 0, y: 0, z: 1.0}}'
```

If we were to combine these two messages, the robot would move forward while turning toward the left. The resulting `Twist` message would be:

```
'{linear: {x: 0.1, y: 0, z: 0}, angular: {x: 0, y: 0, z: 1.0}}'
```

The larger the angular `z` value compared to the linear `x` value, the tighter the turn.

7.3.2 Monitoring Robot Motion using RViz

We will use `RViz` to visualize the robot's motion as we try out various `Twist` commands and motion control scripts. Recall from the RViz User's Guide that we can use the Odometry Display type to track the pose (position and orientation) of the robot at various points along its path. Each pose of the robot is indicated by a large arrow pointing in the direction that the robot was facing at that point. Note that these poses reflect what is reported by the robot's odometry which could differ—sometimes substantially—from how the robot is positioned in the real world. However, if the robot is well-calibrated and operating on a relatively hard surface, the odometry data is usually good enough to get a rough idea of how the robot is doing. Furthermore, when running a fake robot in the ArbotiX simulator where there is no physics, the correspondence will be exact.

Let's try a couple of examples using the ArbotiX simulator. First, fire up the fake TurtleBot using the command:

```
$ roslaunch rbx1_bringup fake_turtlebot.launch
```

In another terminal, bring up `RViz` with the included configuration file that is already set up with the Odometry display:

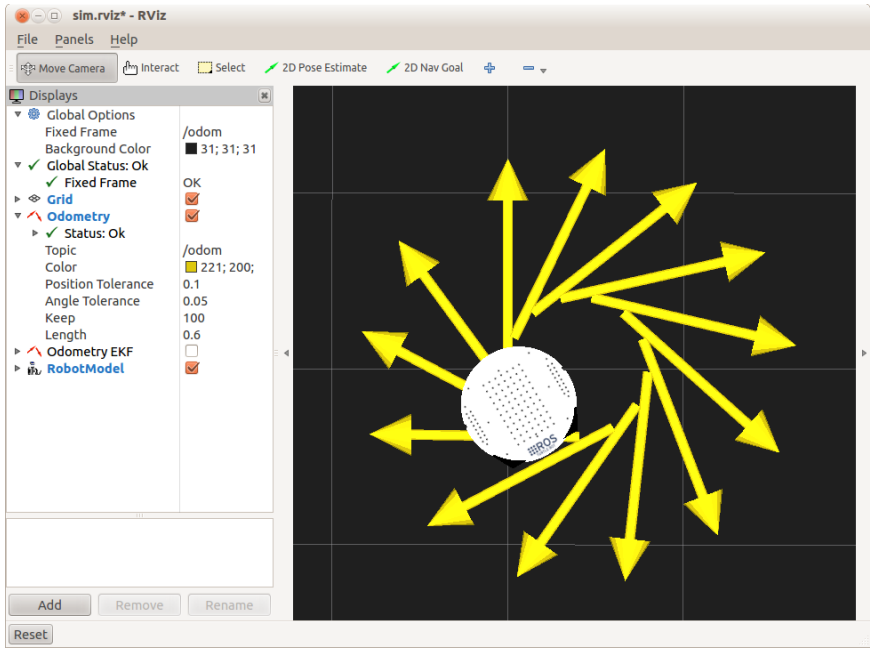
```
$ rosrun rviz rviz -d `rospack find rbx1_nav`/sim.rviz
```

Finally, bring up yet another terminal window and set the robot moving in a clockwise circle by publishing the following `Twist` message:

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

We use the `-r` parameter to publish the `Twist` message continually at 10Hz. Some robots like the TurtleBot require the movement command to be continually published or the robot will stop: a nice safety feature. While this parameter is not necessary when running the ArbotiX simulator, it doesn't hurt either.

If everything is working OK, the result in `RViz` should look something like this:



Note that we have set the **Keep** field to 100 for the **Odometry** Display which indicates that we want to display up to the last 100 arrows before the oldest one drops off. The **Position Tolerance** (in meters) and **Angle Tolerance** (in radians) allow you to control how often a new arrow will be displayed.

To clear the arrows at any point, either click the **Reset** button or un-check the checkbox beside the **Odometry** display, then check it again. To turn off the arrows altogether, leave the checkbox un-checked.

To stop the robot from rotating, type `Ctrl-C` in the same terminal window, then publish the empty `Twist` message:

```
$ rostopic pub -1 /cmd_vel geometry_msgs/Twist '{}'
```

Now let's try a second example. First clear the odometry arrows by clicking the **Reset** button in `RViz`. The following pair of commands (separated by a semi-colon) will first move the robot straight for about 3 seconds (the "`-1`" option means "publish once"), then continue indefinitely in a counter-clockwise circle:

```
$ rostopic pub -1 /cmd_vel geometry_msgs/Twist '{linear: {x: 0.2, y: 0,
  z: 0}, angular: {x: 0, y: 0, z: 0}}'; rostopic pub -r 10 /cmd_vel
  geometry_msgs/Twist '{linear: {x: 0.2, y: 0, z: 0}, angular: {x: 0,
  y: 0, z: 0.5}}'
```

To stop the robot, type `Ctrl-C` in the same terminal window and publish the empty `Twist` message:

```
$ rostopic pub -1 /cmd_vel geometry_msgs/Twist '{}'
```

Before we try out some `Twist` messages on a real robot, we have to spend a few moments talking about calibration.

7.4 Calibrating Your Robot's Odometry

If you don't have a robot, you can skip this section altogether. If you have an original TurtleBot (using the iRobot Create as a base), be sure use to use the automated calibration routine to set the angular correction factors for your robot. You can still use the first part of this section to set the linear correction factor. Note that in all cases you might have to use different calibration parameters for different types of floor surfaces; e.g. carpet versus hardwood. The easiest way to manage the different parameters is to use different launch files for each surface.

If you are using your own custom built robot, you might already have your own calibration method. If so, you can safely skip this section. Otherwise, read on.

Before running the calibration routines, be sure to get the Orocos kinematics packages using the command:

```
$ sudo apt-get install ros-hydro-orocos-kdl ros-hydro-python-orocos-kdl
```

The `rbx1_nav` package includes two calibration scripts: `calibrate_linear.py` and `calibrate_angular.py`. The first attempts to move the robot 1 meter forward by monitoring the `/odom` topic and stopping when the reported distance is within 1 cm of the target. You can adjust the target distance and movement speed by editing the script

or by using `rqt_reconfigure`. The second script rotates the robot 360 degrees, again by monitoring the `/odom` topic. We'll describe how to adjust your PID parameters based on the results in the next two sections.

7.4.1 Linear Calibration

First make sure you have plenty of room in front of your robot—at least 2 meters for the default test distance of 1.0 meters. Using a tape measure, lay out at least 1 meter of tape on the floor and align the starting end of the tape with some identifiable mark on your robot. Rotate the robot so that it is aimed parallel to the tape.

Next, bring up your robot's base controller with the appropriate launch file. For an iRobot Create based TurtleBot, `ssh` into the robot's laptop and run:

```
$ roslaunch rbx1_bringup turtlebot_minimal_create.launch
```

Next, run the linear calibration node:

```
$ rosrun rbx1_nav calibrate_linear.py
```

Finally, run `rqt_reconfigure`:

```
$ rosrun rqt_reconfigure rqt_reconfigure
```

Select the `calibrate_linear` node in the `rqt_reconfigure` window. (If you do not see the `calibrate_linear` node listed, click the blue refresh icon in the upper right corner of the GUI.) To start the test, check the checkbox beside `start_test`. (If the robot does not start moving, un-check the checkbox then check it again.) Your robot should move forward approximately 1 meter. To get the correction factor, follow these steps:

- Measure the actual distance with the tape and make a note of it.
- Divide the actual distance by the target distance and make a note of the ratio.
- Return to the `rqt_reconfigure` GUI and *multiply* the `odom_linear_scale_correction` value by the ratio you just computed. Set the parameter to the new value.
- Repeat the test by moving the robot back to the start of the tape, then checking the `start_test` checkbox in the `rqt_reconfigure` window.
- Continue repeating the test until you are satisfied with the result. An accuracy of 1 cm in 1 meter is probably good enough.

With your final correction factor in hand, you need to apply it to the parameters of your robot's base controller using the appropriate launch file. For a TurtleBot, add the following line to your `turtlebot.launch` file:

```
<param name="turtlebot_node/odom_linear_scale_correction"
value="X"/>
```

where X is your correction factor.

If your robot uses the ArbotiX base controller, edit your YAML configuration file and change the `ticks_meter` parameter by *dividing* it by your correction factor.

As a final check, launch your robot's startup file with the new correction factor. Then run the `calibration_linear.py` script but with the `odom_linear_scale_correction` parameter set to 1.0. Your robot should now go 1.0 meters without further correction.

7.4.2 Angular Calibration

If you have an iRobot Create-based TurtleBot, do not use this method. Instead, run the TurtleBot's automated calibration procedure.

In this test, your robot will only rotate in place so space is not so much an issue. Place a marker (e.g. a piece of tape) on the floor and align it with the front center of the robot. We will rotate the robot 360 degrees and see how close it comes back to the mark.

Bring up up your robot's base controller with the appropriate launch file. For an original TurtleBot (iRobot Create base), `ssh` into the robot's laptop and run:

```
$ roslaunch rbx1_bringup turtlebot_minimal_create.launch
```

Next, run the angular calibration node:

```
$ rosrun rbx1_nav calibrate_angular.py
```

Finally, run `rqt_reconfigure`:

```
$ rosrun rqt_reconfigure rqt_reconfigure
```

Return to the `rqt_reconfigure` window and select the `calibrate_angular` node. (If you do not see the `calibrate_angular` node listed, click the blue refresh icon in the upper right corner of the GUI.) To start the test, check the checkbox beside `start_test`. (If the robot does not start moving, un-check the checkbox then check it again.) Your robot should rotate approximately 360 degrees. Don't worry if it seems to rotate significantly more or less than a full rotation. That's what we're about to fix. To get the correction factor, follow these steps:

- If the actual rotation falls short of a full 360 degrees, eyeball the fraction it did rotate and enter the estimated fraction in the `odom_angular_scale_correction` field in the `rqt_reconfigure` window. So if it looks like the robot rotated about 85% of the way, enter something like 0.85. If it rotated about 5% too far, enter something like 1.05.

- Repeat the test by realigning the marker with the front center of the robot, then check the `start_test` checkbox in the `rqt_reconfigure` window.

- Hopefully the rotation will be closer to 360 degrees. If it is still short, decrease the `odom_angular_scale_correction` parameter a little and try again. If it rotates too far, increase it a little and try again.

- Continue the procedure until you are happy with the result.

What you do with your final correction factor depends on your base controller's PID parameters. For a robot controlled by the ArbotiX base controller, edit your YAML configuration file and change the `base_width` parameter by *dividing* it by your correction factor.

As a final check, launch your robot's startup file with the new correction factor. Then run the `calibration_angular.py` script but with the `odom_angular_scale_correction` parameter set to 1.0. Your robot should now rotate 360 degrees without further correction.

7.5 Sending `Twist` Messages to a Real Robot

If you have a TurtleBot or any other robot that listens on the `/cmd_vel` topic for motion commands, you can try some `Twist` messages in the real world. Always be sure to start with small values for the linear and angular speeds. Try a pure rotation first so that your robot will not suddenly fly across the room and ruin the furniture.

First power on your robot and launch the appropriate startup files. If you have an original TurtleBot (iRobot Create base), `ssh` into the robot's laptop and run:

```
$ roslaunch rbx1_bringup turtlebot_minimal_create.launch
```

If you already have your own launch file that includes your calibration parameters, run that file instead. The launch file used above includes calibration parameters that work well on a low ply carpet for my own TurtleBot. You will likely have to adjust them for your own robot and the type of floor surface it will run on. Use the calibration procedure described earlier to find the parameters that work best for you.

To rotate the robot counterclockwise in place at 1.0 radians per second (about 6 seconds per revolution), run the command:

```
$ rostopic pub -r 10 /cmd_vel geometry_msgs/Twist '{linear: {x: 0, y: 0,
  z: 0}, angular: {x: 0, y: 0, z: 1.0}}'
```

You can run this command either on the TurtleBot (after using `ssh` in another terminal) or you can run it on your workstation assuming you have already set up ROS networking.

To stop the robot, type `Ctrl-C` and, if necessary, publish the empty `Twist` message:

```
$ rostopic pub -1 /cmd_vel geometry_msgs/Twist '{}'
```

(A TurtleBot should stop on its own after typing `Ctrl-C` the first time.)

Next, to move the robot forward at a speed of 0.1 meters per second (about 4 inches per second or 3 seconds per foot), make sure you have lots of space in front of the robot, then run the command:

```
$ rostopic pub -r 10 /cmd_vel geometry_msgs/Twist '{linear: {x: 0.1, y:
  0, z: 0}, angular: {x: 0, y: 0, z: 0}}'
```

To stop the movement, type `Ctrl-C`, and, if necessary, publish the empty `Twist` message:

```
$ rostopic pub -1 /cmd_vel geometry_msgs/Twist '{}'
```

If you feel satisfied with the result, try other combinations of linear x and angular z values. For example, the following command should send the robot turning in a clockwise circle:

```
$ rostopic pub -r 10 /cmd_vel geometry_msgs/Twist '{linear: {x: 0.15, y:
  0, z: 0}, angular: {x: 0, y: 0, z: -0.4}}'
```

To stop the movement, type `Ctrl-C`, and, if necessary, publish the empty `Twist` message:

```
$ rostopic pub -1 /cmd_vel geometry_msgs/Twist '{}'
```

As we mentioned earlier, we don't often publish `Twist` messages to the robot directly from the command line although it can be useful for debugging and testing purposes. More often, we will send such messages programmatically from a ROS node that controls the robot's behavior in some interesting way. Let's look at that next.

7.6 Publishing `Twist` Messages from a ROS Node

So far we have been moving our robot from the command line, but most of the time you will rely on a ROS node to publish the appropriate `Twist` messages. As a simple example, suppose you want to program your robot to move one 1.0 meters forward, turn around 180 degrees, then come back to the starting point. We will attempt to accomplish this task a number of different ways which will nicely illustrate the different levels of motion control in ROS.

7.6.1 Estimating Distance and Rotation Using Time and Speed

Our first attempt will be to use timed `Twist` commands to move the robot forward a certain distance, rotate 180 degrees, then move forward again for the same time and at the same speed where it will hopefully end up where it started. Finally, we will rotate the robot 180 degrees one more time to match the original orientation.

The script can be found in the file `timed_out_and_back.py` in the `rbx1_nav/nodes` subdirectory. Before looking at the code, let's try it in the ArbotiX simulator.

7.6.2 Timed Out-and-Back in the ArbotiX Simulator

To make sure the fake Turtlebot is repositioned at the start location, use `Ctrl-C` to terminate the fake Turtlebot launch file it you already have it running, then bring it up again with the command:

```
$ roslaunch rbx1_bringup fake_turtlebot.launch
```

(If desired, replace the `fake_turtlebot.launch` file with the one for Pi Robot or your own robot. It won't make a difference to the results.)

If `RViz` is not already running, fire it up now:

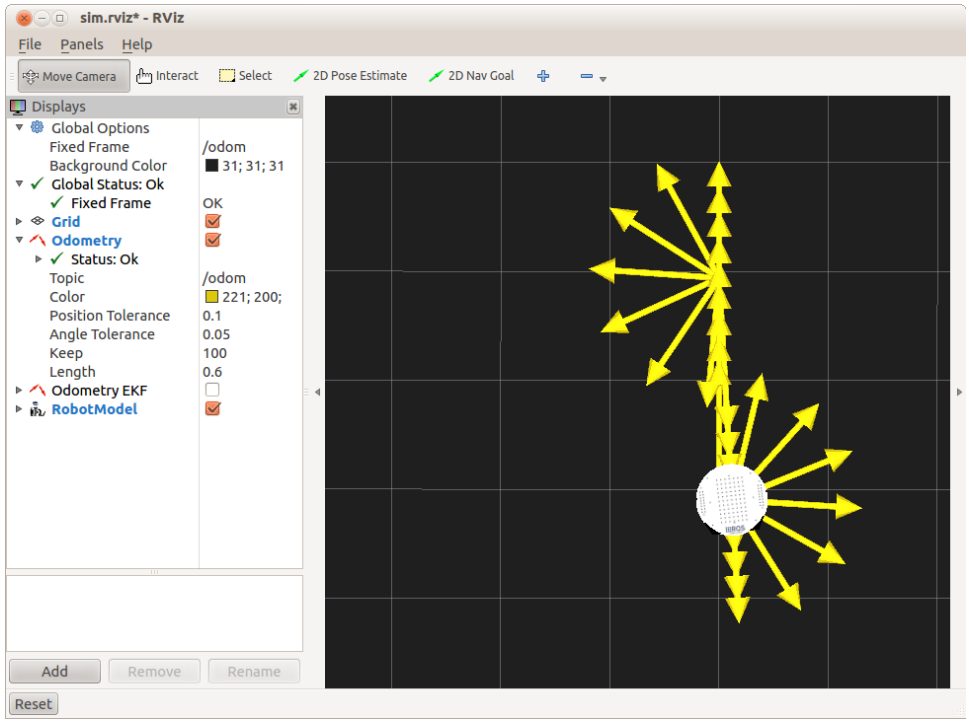
```
$ rosrun rviz rviz -d `rospack find rbx1_nav`/sim.rviz
```

or click the **Reset** button to clear any **Odometry** arrows from the previous section.

Finally, run the `timed_out_and_back.py` node:

```
$ rosrun rbx1_nav timed_out_and_back.py
```

Hopefully `RViz` will show your robot executing an out-and-back maneuver and the end result will look something like the following:



The large arrows represent the position and orientation of the robot at various points along its trajectory as reported by the robot's (fake) internal odometry. We will learn how to make use of this odometry information in the next few sections.

So far, things look pretty good in the ideal simulator. But before trying it out on a real robot, let's look at the code.

7.6.3 The Timed Out-and-Back Script

Here is the full script for the timed out and back node. After the listing, we will break it down into smaller pieces.

Link to source: [timed_out-and-back.py]

```
1  #!/usr/bin/env python
2
3  import rospy
4  from geometry_msgs.msg import Twist
5  from math import pi
6
7  class OutAndBack():
8      def __init__(self):
9          # Give the node a name
10         rospy.init_node('out_and_back', anonymous=False)
```

```
11
12          # Set rospy to execute a shutdown function when exiting
13          rospy.on_shutdown(self.shutdown)
14
15          # Publisher to control the robot's speed
16          self.cmd_vel = rospy.Publisher('/cmd_vel', Twist)
17
18          # How fast will we update the robot's movement?
19          rate = 50
20
21          # Set the equivalent ROS rate variable
22          r = rospy.Rate(rate)
23
24          # Set the forward linear speed to 0.2 meters per second
25          linear_speed = 0.2
26
27          # Set the travel distance to 1.0 meters
28          goal_distance = 1.0
29
30          # How long should it take us to get there?
31          linear_duration = goal_distance / linear_speed
32
33          # Set the rotation speed to 1.0 radians per second
34          angular_speed = 1.0
35
36          # Set the rotation angle to Pi radians (180 degrees)
37          goal_angle = pi
38
39          # How long should it take to rotate?
40          angular_duration = goal_angle / angular_speed
41
42          # Loop through the two legs of the trip
43          for i in range(2):
44              # Initialize the movement command
45              move_cmd = Twist()
46
47              # Set the forward speed
48              move_cmd.linear.x = linear_speed
49
50              # Move forward for a time to go 1 meter
51              ticks = int(linear_duration * rate)
52
53              for t in range(ticks):
54                  self.cmd_vel.publish(move_cmd)
55                  r.sleep()
56
57              # Stop the robot before the rotation
58              move_cmd = Twist()
59              self.cmd_vel.publish(move_cmd)
60              rospy.sleep(1)
61
62              # Now rotate left roughly 180 degrees
63
64              # Set the angular speed
65              move_cmd.angular.z = angular_speed
66
67              # Rotate for a time to go 180 degrees
```

```
68              ticks = int(goal_angle * rate)
69
70              for t in range(ticks):
71                  self.cmd_vel.publish(move_cmd)
72                  r.sleep()
73
74              # Stop the robot before the next leg
75              move_cmd = Twist()
76              self.cmd_vel.publish(move_cmd)
77              rospy.sleep(1)
78
79          # Stop the robot
80          self.cmd_vel.publish(Twist())
81
82      def shutdown(self):
83          # Always stop the robot when shutting down the node.
84          rospy.loginfo("Stopping the robot...")
85          self.cmd_vel.publish(Twist())
86          rospy.sleep(1)
87
88  if __name__ == '__main__':
89      try:
90          OutAndBack()
91      except:
92          rospy.loginfo("Out-and-Back node terminated.")
```

Since this is our first script, let's take it line-by-line starting from the top:

```
1  #!/usr/bin/env python
2
3  import rospy
```

If you did the ROS Beginner Tutorials in Python, you'll already know that all of our ROS nodes begin with these two lines. The first line ensures that the program will run as a Python script while the second imports the main ROS library for Python.

```
4  from geometry_msgs.msg import Twist
5  from math import pi
```

Here we take care of any other imports we need for the script. In this case, we will need the `Twist` message type from the ROS `geometry_msgs` package and the constant `pi` from the Python `math` module. Note that a common source of import errors is to forget to include the necessary ROS `<run_depend>` line in your package's `package.xml` file. In this case, our `package.xml` file has to include the line:

```
<run_depend>geometry_msgs</run_depend>
```

so that we can import `Twist` from `geometry_msgs.msg`.

```
7  class OutAndBack():
```

```
8       def __init__(self):
```

Here we begin the main body of our ROS node by defining it as a Python class along with the standard class initialization line.

```
9           # Give the node a name
10          rospy.init_node('out_and_back', anonymous=False)
11
12          # Set rospy to execute a shutdown function when exiting
13          rospy.on_shutdown(self.shutdown)
```

Every ROS node requires a call to `rospy.init_node()` and we also set a callback for the `on_shutdown()` function so that we can perform any necessary cleanup when the script is terminated—e.g. when the user hits `Ctrl-C`. In the case of a mobile robot, the most important cleanup task is to stop the robot! We'll see how to do this later in the script.

```
15          # Publisher to control the robot's speed
16          self.cmd_vel = rospy.Publisher('/cmd_vel', Twist)
17
18          # How fast will we update the robot's movement?
19          rate = 50
20
21          # Set the equivalent ROS rate variable
22          r = rospy.Rate(rate)
```

Here we define our ROS publisher for sending `Twist` commands to the `/cmd_vel` topic. We also set the rate at which we want to update the robot's movement—50 times per second in this case.

```
24          # Set the forward linear speed to 0.2 meters per second
25          linear_speed = 0.2
26
27          # Set the travel distance to 1.0 meters
28          goal_distance = 1.0
29
30          # How long should it take us to get there?
31          linear_duration = linear_distance / linear_speed
```

We initialize the forward speed to a relatively safe 0.2 meters per second and the target distance to 1.0 meters. We then compute how long this should take.

```
33          # Set the rotation speed to 1.0 radians per second
34          angular_speed = 1.0
35
36          # Set the rotation angle to Pi radians (180 degrees)
37          goal_angle = pi
38
39          # How long should it take to rotate?
40          angular_duration = angular_distance / angular_speed
```

Similarly, we set the rotation speed to 1.0 radians per second and the target angle to 180 degrees or Pi radians.

```
42          # Loop through the two legs of the trip
43          for i in range(2):
44              # Initialize the movement command
45              move_cmd = Twist()
46
47              # Set the forward speed
48              move_cmd.linear.x = linear_speed
49
50              # Move forward for a time to go the desired distance
51              ticks = int(linear_duration * rate)
52
53              for t in range(ticks):
54                  self.cmd_vel.publish(move_cmd)
55                  r.sleep()
```

This is the loop that actually moves the robot—one cycle for each of the two legs. Recall that some robots require a `Twist` message to be continually published to keep it moving. So to move the robot forward `linear_distance` meters at a speed of `linear_speed` meters per second, we publish the `move_cmd` message every `1/rate` seconds for the appropriate duration. The expression `r.sleep()` is a shorthand for `rospy.sleep(1/rate)` since we defined the variable `r = rospy.Rate(rate)`.

```
62              # Now rotate left roughly 180 degrees
63
64              # Set the angular speed
65              move_cmd.angular.z = angular_speed
66
67              # Rotate for a time to go 180 degrees
68              ticks = int(goal_angle * rate)
69
70              for t in range(ticks):
71                  self.cmd_vel.publish(move_cmd)
72                  r.sleep()
```

In the second part of the loop, we rotate the robot at a rate of `angular_speed` radians per second for the appropriate duration (Pi seconds in this case) which should yield 180 degrees.

```
79          # Stop the robot.
80          self.cmd_vel.publish(Twist())
```

When the robot has completed the out-and-back journey, we stop it by publishing an empty `Twist` message (all fields are 0).

```
82      def shutdown(self):
83          # Always stop the robot when shutting down the node.
84          rospy.loginfo("Stopping the robot...")
```

```
85          self.cmd_vel.publish(Twist())
86          rospy.sleep(1)
```

This is our shutdown callback function. If the script is terminated for any reason, we stop the robot by publishing an empty `Twist` message.

```
88 if __name__ == '__main__':
89     try:
90         OutAndBack()
91     except rospy.ROSInterruptException:
92         rospy.loginfo("Out-and-Back node terminated.")
```

Finally, this is the standard Python block for running the script. We simply instantiate the `OutAndBack` class which sets the script (and robot) in motion.

7.6.4 Timed Out and Back using a Real Robot

If you have a robot like the TurtleBot, you can try the `timed_out_and_back.py` script in the real world. Remember that we are only using time and speed to estimate distance and rotation angles. So we expect that the robot's inertia will throw things off compared to the ideal ArbotiX simulation (which, as you will recall, does not model any physics.)

First, terminate any running simulations. Next, make sure your robot has plenty of room to work in—at least 1.5 meters ahead of it and a meter on either side. Then bring up your robot's startup launch file(s). If you have an original TurtleBot (iRobot Create base), `ssh` into the robot's laptop and run:

```
$ roslaunch rbx1_bringup turtlebot_minimal_create.launch
```

Or use your own launch file if you have created one to store your calibration parameters.

We will also use an auxiliary script so that we can see the TurtleBot's combined odometry frame in `RViz`. (This will become clearer in the next section.) You can skip this if you are not using a TurtleBot. This launch file should be run on the TurtleBot's laptop using another `ssh` terminal:

```
$ roslaunch rbx1_bringup odom_ekf.launch
```

Next we're going to configure `RViz` to display the combined odometry data (encoders + gyro) rather than `/odom` which only shows the encoder data. If you already have `RViz` running from the previous test, you can simply un-check the **Odometry** display and check the **`Odometry EKF`** display, then skip the following step.

If RViz is not already up, run it now on your workstation with the nav_ekf config file. This file simply pre-selects the /odom_ekf topic for displaying the combined odometry data:

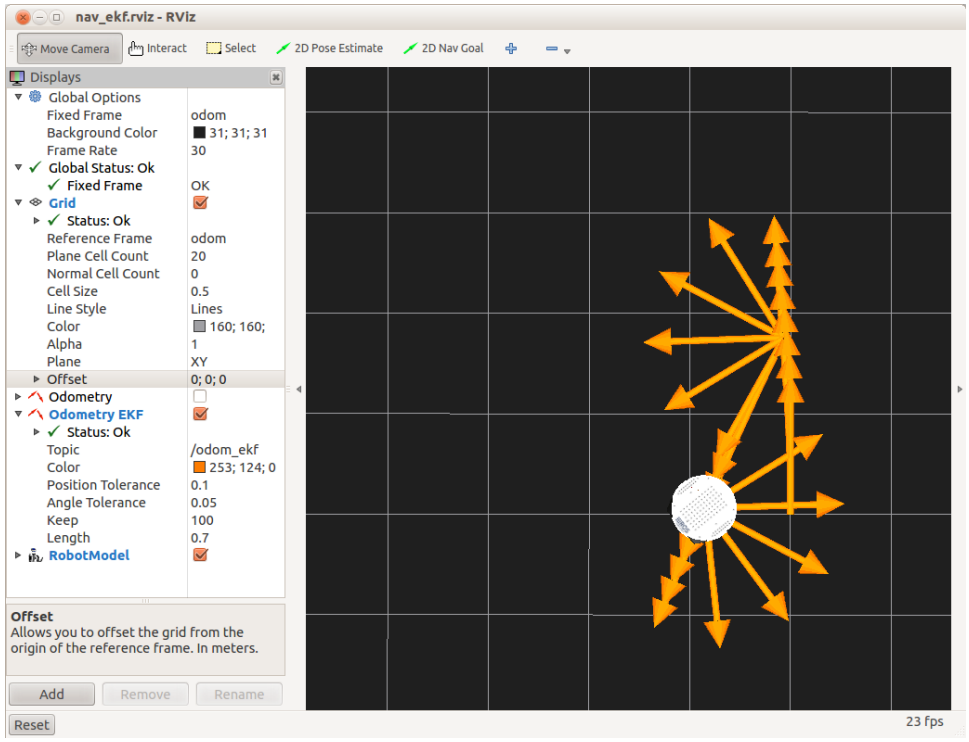
```
$ rosrun rviz rviz -d `rospack find rbx1_nav`/nav_ekf.rviz
```

The only difference between this configuration and the previous one is that we are now displaying the combined odometry data on the /odom_ekf topic rather than just the wheel encoder data published on the /odom topic. You can check both displays if you want to compare the two.

Finally, we run the out and back script just like we did before. Note that the script itself does not care if we are running a simulation or controlling a real robot. It is simply publishing Twist messages on the /cmd_vel topic for anyone who cares to listen. This is one example of how ROS allows us to abstract away the lower levels of the motion control hierarchy. You can run the following command either on your workstation or on the robot's laptop after logging in with ssh:

```
$ rosrun rbx1_nav timed_out_and_back.py
```

Here is the result for my own TurtleBot when operating on a low-ply carpet:



As you can tell from the picture, the robot did not end up very close to the starting position. First of all, it did not go far enough before turning around (the grid squares are 0.5 meters apart). Then it did not rotate a full 180 degrees before heading back. The result is that the robot is about 0.5m to the left of the starting point, and it is oriented in the wrong direction.

Fortunately, the data we need to correct the problem is staring us right in the face. The large odometry arrows in the image above indicate the robot's position and orientation as reported by its internal odometry. In other words, the robot "knows" it messed up but we have unfairly handicapped it by not using the odometry data in our script. While the odometry data will not match the real motion exactly, it should give us a better result if we use it.

7.7 Are We There Yet? Going the Distance with Odometry

When we ask our robot to move or rotate at a certain speed, how do we know that it is actually doing what we asked? For example, if we publish a `Twist` message to move the robot forward at 0.2 m/s, how do we know that the robot isn't really going 0.18 m/s? In fact, how do we know that both wheels are even traveling at the same speed?

As we explained earlier in this chapter, the robot's base controller node uses odometry and PID control to turn motion requests into real-world velocities. The accuracy and reliability of this process depends on the robot's internal sensors, the accuracy of the calibration procedure, and environmental conditions. (For example, some surfaces may allow the wheels to slip slightly which will mess up the mapping between encoder counts and distance traveled.)

The robot's internal odometry can be supplemented with external measures of the robot's position and/or orientation. For example, one can use wall-mounted visual markers such as fiducials together with the ROS packages `ar_pose`, `ar_kinect` or `ar_track_alvar` to provide a fairly accurate localization of the robot within a room. A similar technique uses visual feature matching without the need for artificial markers (`ccny_rgbd_tools`, `rgbdslam`), and yet another package (`laser_scan_matcher`) uses laser scan matching. Outdoor robots often use GPS to estimate position and indoor robots can use other active localization techniques like the NorthStar system by Evolution Robotics.

For the purposes of this book, we will use the term "odometry" to mean internal position data. However, regardless of how one measures odometry, ROS provides a message type to store the information; namely `nav_msgs/Odometry`. The abbreviated definition of the `Odometry` message type is shown below:

```
Header header
string child_frame_id
geometry_msgs/PoseWithCovariance pose
geometry_msgs/TwistWithCovariance twist
```

Here we see that the `Odometry` message is composed of a `Header`, a `string` identifying the `child_frame_id`, and two sub-messages, one for `PoseWithCovariance` and one for `TwistWithCovariance`.

To see the expanded version of the definition, run the command:

```
$ rosmsg show nav_msgs/Odometry
```

which should yield the following output:

```
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
```

The PoseWithCovariance sub-message records the position and orientation of the robot while the TwistWithCovariance component gives us the linear and angular speeds as we have already seen. Both the pose and twist can be supplemented with a covariance matrix which measures the uncertainty in the various measurements.

The Header and child_frame_id define the reference frames we are using to measure distances and angles. It also provides a timestamp for each message so we know not only where we are but when. By convention, odometry measurements in ROS use /odom as the parent frame id and /base_link (or /base_footprint) as the child frame id. While the /base_link frame corresponds to a real physical part of the robot, the /odom frame is defined by the translations and rotations encapsulated in the odometry data. These transformations move the robot relative to the /odom frame. If we display the robot model in RViz and set the fixed frame to the /odom frame, the robot's position will reflect where the robot "thinks" it is relative to its starting position.

7.8 Out and Back Using Odometry

Now that we understand how odometry information is represented in ROS, we can be more precise about moving our robot on an out-and-back course. Rather than guessing distances and angles based on time and speed, our next script will monitor the robot's position and orientation as reported by the transform between the /odom and /base_link frames.

The new file is called odom_out_and_back.py in the rbx1_nav/nodes directory. Before looking at the code, let's compare the results between the simulator and a real robot.

7.8.1 Odometry-Based Out and Back in the ArbotiX Simulator

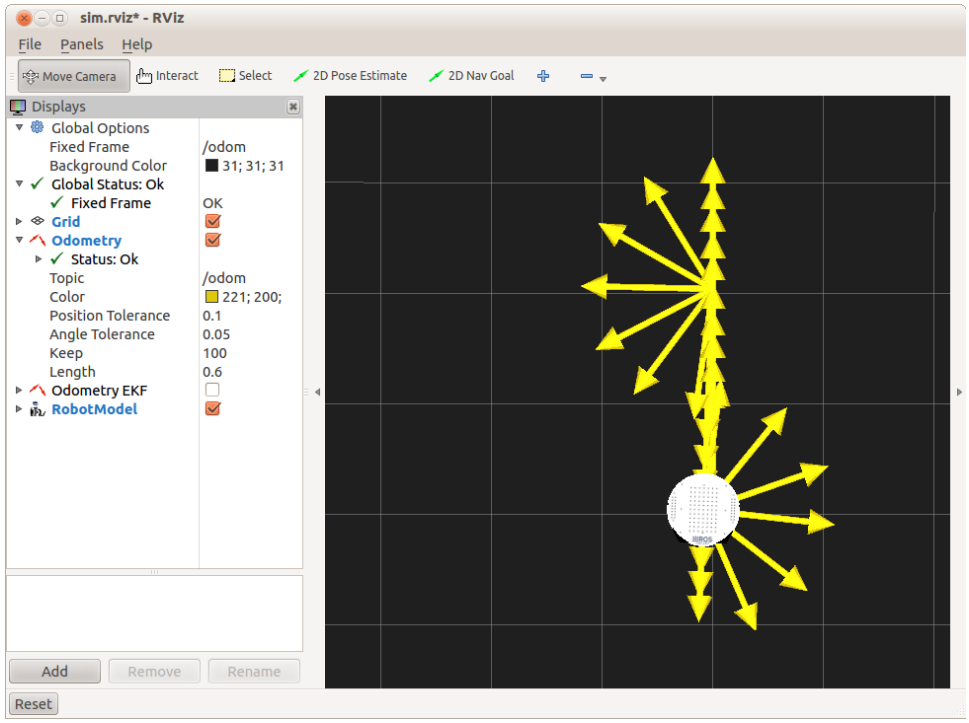
If you already have a simulated robot running, first Ctrl-C out of the simulation so we can start the odometry readings from scratch. Then bring up the simulated robot again, run RViz, then run the odom_out_and_back.py script as follows:

```
$ roslaunch rbx1_bringup fake_turtlebot.launch
```

```
$ rosrun rviz rviz -d `rospack find rbx1_nav`/sim.rviz
```

```
$ rosrun rbx1_nav odom_out_and_back.py
```

A typical result is shown below:



As you can see, using odometry in an ideal simulator without physics produces basically perfect results. This should not be terribly surprising. So what happens when we try it on a real robot?

7.8.2 Odometry-Based Out and Back Using a Real Robot

If you have a TurtleBot or other ROS-compatible robot, you can try the odometry based out-and-back script in the real world.

First make sure you terminate any running simulations. Then bring up your robot's startup launch file(s). For a TurtleBot you would run:

```
$ roslaunch rbx1_bringup turtlebot_minimal_create.launch
```

(Or use your own launch file if you have created one to store your calibration parameters.)

Make sure your robot has plenty of room to work in—at least 1.5 meters ahead of it and a meter on either side.

If you are using a TurtleBot, we will also run the `odom_ekf.py` script (included in the `rbx1_bringup` package) so we can see the TurtleBot's combined odometry frame in

RViz. You can skip this if you are not using a TurtleBot. This launch file should be run on the TurtleBot's laptop:

```
$ roslaunch rbx1_bringup odom_ekf.launch
```

If you already have RViz running from the previous test, you can simply un-check the **Odometry** display and check the **Odometry EKF** display, then skip the following step.

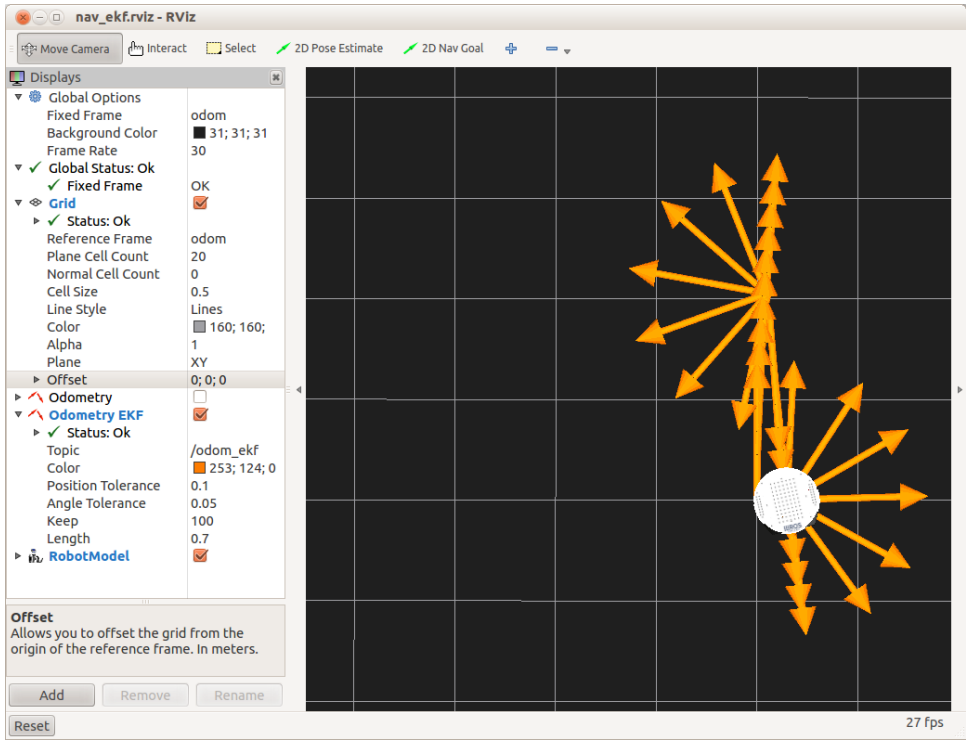
If RViz is not already up, run it now on your workstation with the nav_ekf config file. This file simply pre-selects the /odom_ekf topic for displaying the combined odometry data:

```
$ rosrun rviz rviz -d `rospack find rbx1_nav`/nav_ekf.rviz
```

Finally, launch the odometry-based out-and-back script just like we did in simulation. You can run the following command either on your workstation or on the robot's laptop after logging in with ssh:

```
$ rosrun rbx1_nav odom_out_and_back.py
```

Here is the result for my own TurtleBot when operating on a low-ply carpet:



As you can see from the picture, the result is much better than the timed out-and-back case. If fact, in the real world, the result was even better than shown in `RViz`. (Remember that the odometry arrows in `RViz` won't line up exactly with the actual position and orientation of the robot in the real world.) In this particular run, the robot ended up less than 1 cm from the starting position and only a few degrees away from the correct orientation. Of course, to get results even this good, you will need to spend some time carefully calibrating your robot's odometry as described earlier.

7.8.3 The Odometry-Based Out-and-Back Script

Here now is the full odometry-based out-and-back script. The embedded comments should make the script fairly self-explanatory but we will describe the key lines in greater detail following the listing.

```
1  #!/usr/bin/env python
2
3  import rospy
4  from geometry_msgs.msg import Twist, Point, Quaternion
5  import tf
6  from rbx1_nav.transform_utils import quat_to_angle, normalize_angle
7  from math import radians, copysign, sqrt, pow, pi
8
9  class OutAndBack():
10     def __init__(self):
11         # Give the node a name
12         rospy.init_node('out_and_back', anonymous=False)
13
14         # Set rospy to execute a shutdown function when exiting
15         rospy.on_shutdown(self.shutdown)
16
17         # Publisher to control the robot's speed
18         self.cmd_vel = rospy.Publisher('/cmd_vel', Twist)
19
20         # How fast will we update the robot's movement?
21         rate = 20
22
23         # Set the equivalent ROS rate variable
24         r = rospy.Rate(rate)
25
26         # Set the forward linear speed to 0.2 meters per second
27         linear_speed = 0.2
28
29         # Set the travel distance in meters
30         goal_distance = 1.0
31
32         # Set the rotation speed in radians per second
33         angular_speed = 1.0
34
35         # Set the angular tolerance in degrees converted to radians
36         angular_tolerance = radians(2.5)
37
38         # Set the rotation angle to Pi radians (180 degrees)
39         goal_angle = pi
```

```
40
41          # Initialize the tf listener
42          self.tf_listener = tf.TransformListener()
43
44          # Give tf some time to fill its buffer
45          rospy.sleep(2)
46
47          # Set the odom frame
48          self.odom_frame = '/odom'
49
50          # Find out if the robot uses /base_link or /base_footprint
51          try:
52              self.tf_listener.waitForTransform(self.odom_frame,
'/base_footprint', rospy.Time(), rospy.Duration(1.0))
53              self.base_frame = '/base_footprint'
54          except (tf.Exception, tf.ConnectivityException, tf.LookupException):
55              try:
56                  self.tf_listener.waitForTransform(self.odom_frame,
'/base_link', rospy.Time(), rospy.Duration(1.0))
57                  self.base_frame = '/base_link'
58              except (tf.Exception, tf.ConnectivityException,
tf.LookupException):
59                  rospy.loginfo("Cannot find transform between /odom and
/base_link or /base_footprint")
60                  rospy.signal_shutdown("tf Exception")
61
62          # Initialize the position variable as a Point type
63          position = Point()
64
65          # Loop once for each leg of the trip
66          for i in range(2):
67              # Initialize the movement command
68              move_cmd = Twist()
69
70              # Set the movement command to forward motion
71              move_cmd.linear.x = linear_speed
72
73              # Get the starting position values
74              (position, rotation) = self.get_odom()
75
76              x_start = position.x
77              y_start = position.y
78
79              # Keep track of the distance traveled
80              distance = 0
81
82              # Enter the loop to move along a side
83              while distance < goal_distance and not rospy.is_shutdown():
84                  # Publish the Twist message and sleep 1 cycle
85                  self.cmd_vel.publish(move_cmd)
86
87                  r.sleep()
88
89                  # Get the current position
90                  (position, rotation) = self.get_odom()
91
92                  # Compute the Euclidean distance from the start
```

```
93                  distance = sqrt(pow((position.x - x_start), 2) +
94                                  pow((position.y - y_start), 2))
95
96              # Stop the robot before the rotation
97              move_cmd = Twist()
98              self.cmd_vel.publish(move_cmd)
99              rospy.sleep(1)
100
101             # Set the movement command to a rotation
102             move_cmd.angular.z = angular_speed
103
104     # Track the last angle measured
105     last_angle = rotation
106
107     # Track how far we have turned
108     turn_angle = 0
109
110     while abs(turn_angle + angular_tolerance) < abs(goal_angle) and not
rospy.is_shutdown():
111         # Publish the Twist message and sleep 1 cycle
112         self.cmd_vel.publish(move_cmd)
113         r.sleep()
114
115         # Get the current rotation
116         (position, rotation) = self.get_odom()
117
118         # Compute the amount of rotation since the last loop
119         delta_angle = normalize_angle(rotation - last_angle)
120
121         # Add to the running total
122         turn_angle += delta_angle
123         last_angle = rotation
124
125         # Stop the robot before the next leg
126         move_cmd = Twist()
127        self.cmd_vel.publish(move_cmd)
128        rospy.sleep(1)
129
130        # Stop the robot for good
131        self.cmd_vel.publish(Twist())
132
133   def get_odom(self):
134     # Get the current transform between the odom and base frames
135     try:
136         (trans, rot)  = self.tf_listener.lookupTransform(self.odom_frame,
self.base_frame, rospy.Time(0))
137     except (tf.Exception, tf.ConnectivityException, tf.LookupException):
138         rospy.loginfo("TF Exception")
139         return
140
141     return (Point(*trans), quat_to_angle(Quaternion(*rot)))
142
143   def shutdown(self):
144     # Always stop the robot when shutting down the node.
145     rospy.loginfo("Stopping the robot...")
146     self.cmd_vel.publish(Twist())
147     rospy.sleep(1)
```

```
148
149 if __name__ == '__main__':
150     try:
151         OutAndBack()
152     except:
153         rospy.loginfo("Out-and-Back node terminated.")
```

Let's now look at the key lines in the script.

```
4  from geometry_msgs.msg import Twist, Point, Quaternion
5  import tf
6  from rbx1_nav.transform_utils import quat_to_angle, normalize_angle
```

We will need the `Twist`, `Point` and `Quaternion` data types from the `geometry_msgs` package. We will also need the `tf` library to monitor the transformation between the `/odom` and `/base_link` (or `/base_footprint`) frames. The `transform_utils` library is a small module that you can find in the `rbx1_nav/src/rbx1_nav` directory and contains a couple of handy functions borrowed from the TurtleBot package. The function `quat_to_angle` converts a quaternion to an Euler angle (yaw) while the `normalize_angle` function removes the ambiguity between 180 and -180 degrees as well as 0 and 360 degrees.

```
35          # Set the angular tolerance in degrees converted to radians
36          angular_tolerance = radians(2.5)
```

Here we define an `angular_tolerance` for rotations. The reason is that it is very easy to overshoot the rotation with a real robot and even a slight angular error can send the robot far away from the next location. Empirically it was found that a tolerance of about 2.5 degrees gives acceptable results.

```
41          # Initialize the tf listener
42          self.tf_listener = tf.TransformListener()
43
44          # Give tf some time to fill its buffer
45          rospy.sleep(2)
46
47          # Set the odom frame
48          self.odom_frame = '/odom'
49
50          # Find out if the robot uses /base_link or /base_footprint
51          try:
52              self.tf_listener.waitForTransform(self.odom_frame,
'/base_footprint', rospy.Time(), rospy.Duration(1.0))
53              self.base_frame = '/base_footprint'
54          except (tf.Exception, tf.ConnectivityException, tf.LookupException):
55              try:
56                  self.tf_listener.waitForTransform(self.odom_frame,
'/base_link', rospy.Time(), rospy.Duration(1.0))
57                  self.base_frame = '/base_link'
58              except (tf.Exception, tf.ConnectivityException,
tf.LookupException):
```

```
59                  rospy.loginfo("Cannot find transform between /odom and
/base_link or /base_footprint")
60                  rospy.signal_shutdown("tf Exception")
```

Next we create a `TransformListener` object to monitor frame transforms. Note that `tf` needs a little time to fill up the listener's buffer so we add a call to `rospy.sleep(2)`. To obtain the robot's position and orientation, we need the transform between the `/odom` frame and either the `/base_footprint` frame as used by the TurtleBot or the `/base_link` frame as used by Pi Robot and Maxwell. First we test for the `/base_footprint` frame and if we don't find it, we test for the /`base_link` frame. The result is stored in the `self.base_frame` variable to be used later in the script.

```
63          for i in range(2):
64              # Initialize the movement command
65              move_cmd = Twist()
```

As with the timed out-and-back script, we loop through the two legs of the trip: first moving the robot 1 meter forward, then rotating it 180 degrees.

```
71              (position, rotation) = self.get_odom()
72
73              x_start = position.x
74              y_start = position.y
```

At the start of each leg, we record the starting position and orientation using the `get_odom()` function. Let's look at that next so we understand how it works.

```
130   def get_odom(self):
131     # Get the current transform between the odom and base frames
132     try:
133         (trans, rot)  = self.tf_listener.lookupTransform(self.odom_frame,
self.base_frame, rospy.Time(0))
134     except (tf.Exception, tf.ConnectivityException, tf.LookupException):
135         rospy.loginfo("TF Exception")
136         return
137
138     return (Point(*trans), quat_to_angle(Quaternion(*rot)))
```

The `get_odom()` function first uses the `tf_listener` object to look up the current transform between the odometry and base frames. If there is a problem with the lookup, we throw an exception. Otherwise, we return a `Point` representation of the translation and a `Quaternion` representation of the rotation. The `*` in front of the `trans` and `rot` variables is Python's notation for passing a list of numbers to a function and we use it here since `trans` is a list of x, y, and z coordinates while `rot` is a list of x, y, z and w quaternion components.

Now back to the main part of the script:

```
76              # Keep track of the distance traveled
77              distance = 0
78
79              # Enter the loop to move along a side
80              while distance < goal_distance and not rospy.is_shutdown():
81                  # Publish the Twist message and sleep 1 cycle
82                  self.cmd_vel.publish(move_cmd)
83
84                  r.sleep()
85
86                  # Get the current position
87                  (position, rotation) = self.get_odom()
88
89                  # Compute the Euclidean distance from the start
90                  distance = sqrt(pow((position.x - x_start), 2) +
91                                  pow((position.y - y_start), 2))
```

This is just our loop to move the robot forward until we have gone 1.0 meters.

```
104     # Track how far we have turned
105     turn_angle = 0
106
107     while abs(turn_angle + angular_tolerance) < abs(goal_angle) and not
rospy.is_shutdown():
108         # Publish the Twist message and sleep for 1 cycle
109         self.cmd_vel.publish(move_cmd)
110         r.sleep()
111
112         # Get the current rotation
113         (position, rotation) = self.get_odom()
114
115         # Compute the amount of rotation since the last loop
116         delta_angle = normalize_angle(rotation - last_angle)
117
118         # Add to the running total
119         turn_angle += delta_angle
120         last_angle = rotation
```

And this is our loop for rotating through 180 degrees within the angular tolerance we set near the beginning of the script.

7.8.4 The `/odom` Topic versus the `/odom` Frame

The reader may be wondering why we used a `TransformListener` in the previous script to access odometry information rather than just subscribing to the `/odom` topic. The reason is that the data published on the `/odom` topic is not always the full story. For example, the TurtleBot uses a single-axis gyro to provide an additional estimate of the robot's rotation. This is combined with the data from the wheel encoders by the `robot_pose_ekf` node (which is started in the TurtleBot launch file) to get a better estimate of rotation than either source alone.

However, the robot_pose_ekf node does not publish its data back on the /odom topic which is reserved for the wheel encoder data. Instead, it publishes it on the /odom_combined topic. Furthermore, the data is published not as an Odometry message but as a PoseWithCovarianceStamped message. It *does* however, publish a transform from the /odom frame to the /base_link frame which provides the information we need. As a result, it is generally safer to use tf to monitor the transformation between the /odom and /base_link (or /base_footprint) frames than to rely on the /odom message topic.

In the rbx1_bringup/nodes directory you will find a node called odom_ekf.py that republishes the PoseWithCovarianceStamped message found on the /odom_combined topic as an Odometry type message on the topic called /odom_ekf. This allows us to view both the /odom topic and the /odom_ekf topic in RViz to compare the TurtleBot's wheel-based odometry with the combined odometry that includes the gyro data.

7.9 Navigating a Square using Odometry

As we did with the odometry-based out-and-back script, we will monitor the position and orientation of the robot using the tf transform between the /odom and /base_link (or /base_footprint) frames. However, this time we will attempt to move the robot in a square by setting four waypoints, one at each corner. At the end of the run, we can see how close the robot gets back to the starting location and orientation. Let's start with a simulation and then try it on a real robot.

7.9.1 Navigating a Square in the ArbotiX Simulator

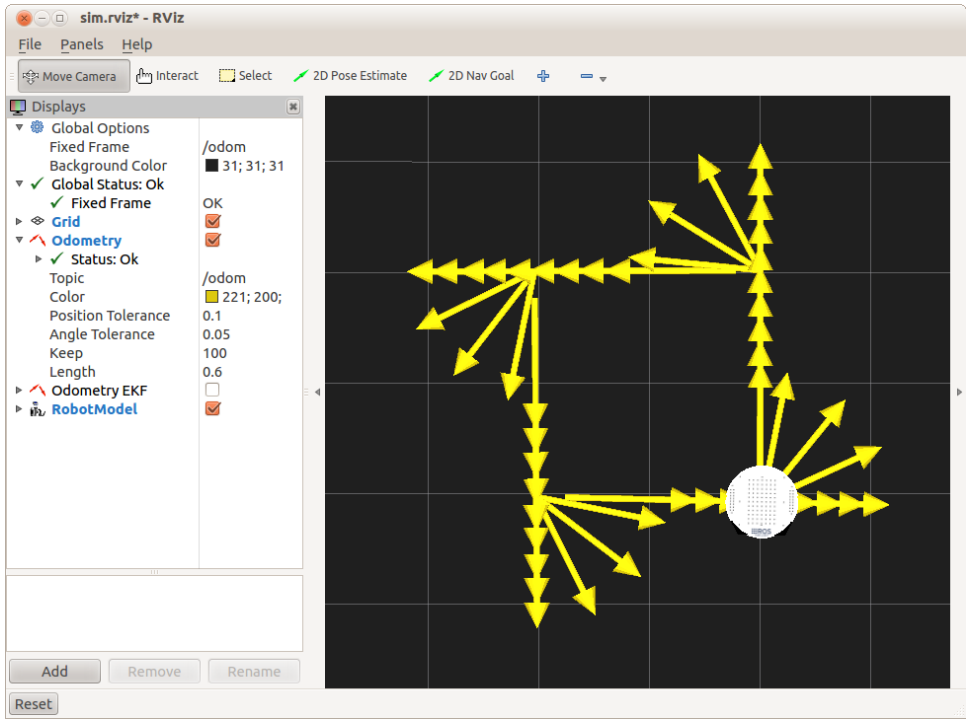
If you already have a simulated TurtleBot or Pi Robot running, first Ctrl-C out of the simulation so we can start the odometry readings from scratch. Then bring up the simulated robot again, run RViz, then run the nav_square.py script as follows:

```
$ roslaunch rbx1_bringup fake_turtlebot.launch
```

```
$ rosrun rviz rviz -d `rospack find rbx1_nav`/sim.rviz
```

```
$ rosrun rbx1_nav nav_square.py
```

A typical result is shown below:



As before, the odometry arrows illustrate the heading of the robot at various points along its route. As you can see, the square is not perfectly aligned with the grid lines, but it's not bad.

7.9.2 Navigating a Square using a Real Robot

If you have a robot, try out the `nav_square` script now to see how well it does with real-world odometry. First terminate any running simulated robots, then launch the startup file(s) for your robot. For a TurtleBot you would run:

```
$ roslaunch rbx1_bringup turtlebot_minimal_create.launch
```

(Or use your own launch file if you have created one to store your calibration parameters.)

Make sure your robot has plenty of room to work in—at least 1.5 meters ahead of it and on either side.

If you are using a TurtleBot, we also need to run the `odom_ekf.py` script to be able to see the TurtleBot's combined odometry frame in `RViz`. You can skip this if you are not using a TurtleBot. The launch file should be run on the TurtleBot's laptop:

```
$ roslaunch rbx1_bringup odom_ekf.launch
```

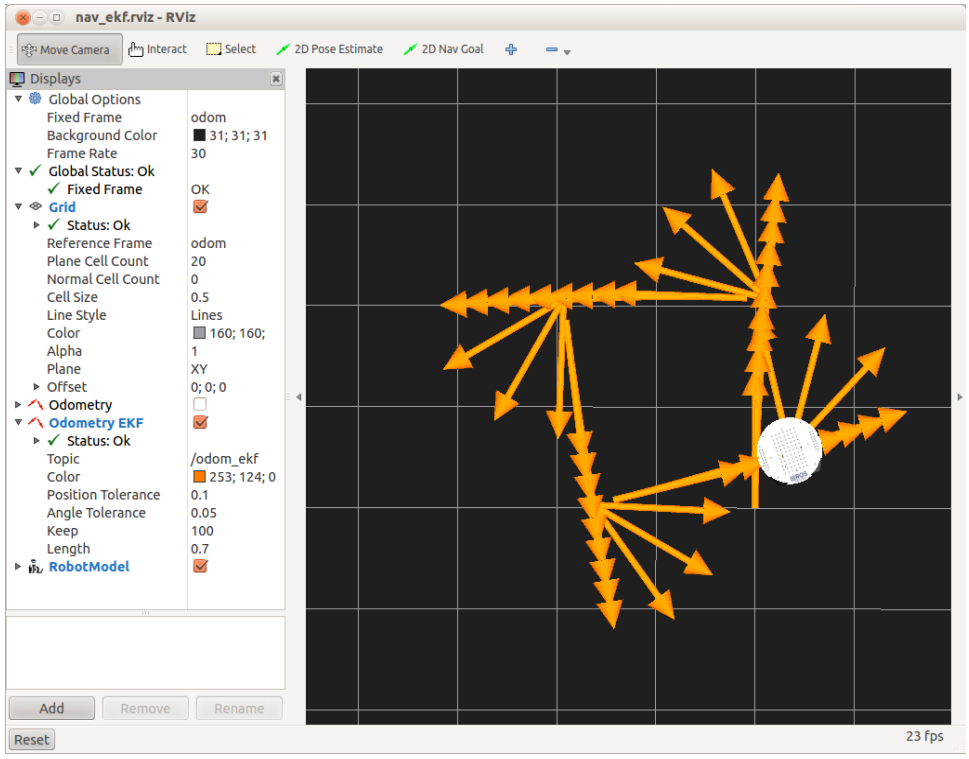
If `RViz` is still running on your workstation, shut it down and bring it back up with the `ekf` configuration file. Alternatively, simply un-check the **Odometry** display and check the **Odometry EKF** display.

```
$ rosrun rviz rviz -d `rospack find rbx1_nav`/nav_ekf.rviz
```

Finally, run the `nav_square` script again:

```
$ rosrun rbx1_nav nav_square.py
```

The following image shows the results when using my own TurtleBot on a low-ply carpet:



As you can see, the result is not too bad. In the real world, the robot ended up 11 cm away from the starting point and about 15 degrees off the original orientation. But of course, if we were to run the script a second time without repositioning the robot, the error in the starting orientation would throw off the entire trajectory.

7.9.3 The `nav_square.py` Script

The `nav_square.py` script is nearly identical to the odometry-based out-and-back script that we just looked at so we won't display it here. The only difference is that we now send the robot along four 1 meter legs using 90 degree rotations instead of two 1 meter legs using 180 rotations. You can look at the code at the link below or by viewing your own copy in the `rbx1_nav/nodes` directory.

Link to source: nav_square.py

7.9.4 The Trouble with Dead Reckoning

The processing of navigating a course while relying only on internal motion data and without reference to external landmarks is referred to as dead reckoning. Any robot that relies entirely on dead reckoning for any length of time will eventually become completely lost. The root of the problem is that even small errors in odometry accumulate over time. Imagine for example an error of even 1 degree in the robot's estimated heading just before it moves straight for 3 meters. At the end of the run, the robot will be have accumulated an error of over 5 cm in its position. Since the robot does not know that it is 5 cm away from its intended destination, the calculation of the next movement will be off by this amount before it even gets started.

Fortunately, roboticists long ago started working on various ways to incorporate landmarks or other external references into robot navigation and we will do the same with ROS in the chapter on SLAM.

7.10 Teleoperating your Robot

It is always a good idea to maintain some kind of manual control over your robot, especially when testing new code. We have seen that the base controller for a robot subscribes to the `/cmd_vel` topic and maps any `Twist` messages published there into motor signals. If we can coax a remote control device such as a joystick or game controller to publish `Twist` messages on the `/cmd_vel` topic, we can use it to teleoperate the robot. (This is another good example of how ROS enables us to abstract away from the underlying hardware.)

Fortunately, the `turtlebot_teleop` package already includes teleoperation nodes for using the keyboard, a joystick or a PS3 controller. First make sure you have the required ROS packages:

```
$ sudo apt-get install ros-hydro-joystick-drivers \
  ros-hydro-turtlebot-apps
```

Before teleoperating a real robot, try it out using the ArbotiX simulator. Bring up the fake TurtleBot if it is not already running:

```
$ roslaunch rbx1_bringup fake_turtlebot.launch
```

And it RViz is not still running, fire it up now:

```
$ rosrun rviz rviz -d `rospack find rbx1_nav`/sim.rviz
```

Now let's take a look at teleoperating the simulated robot using either the keyboard or a joystick.

7.10.1 Using the Keyboard

The turtlebot_teleop package includes a keyboard_teleop.launch file that has been copied to the rbx1_nav/launch directory so that we can edit a couple of parameters described below. Use the following command to run this copy of the launch file:

```
$ roslaunch rbx1_nav keyboard_teleop.launch
```

You should then see the following instructions on your terminal screen:

```
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%
anything else : stop

CTRL-C to quit
```

With your cursor over the teleop terminal window, try tapping the letter i. You should see the simulated TurtleBot move forward in RViz. Try some of the other keys to make sure everything is working.

If you look at the keyboard_teleop.launch file in the rbx1_nav/launch directory, you will see that the keyboard teleop node takes two parameters: scale_linear and scale_angular for determining the default linear and angular speed of the robot. When first testing teleoperating on a real robot, it is a good idea to set these values smaller than the defaults so that the robot will move slowly. Unfortunately, at the time of this writing, there is a bug in the turtlebot_teleop package that prevents the two parameters from being read by the keyboard_teleop node, so for the time being, we have to accept the default values.

7.10.2 Using a Logitech Game Pad

If you have a joystick or game pad you can use the `joystick_teleop.launch` file in the `turtlebot_teleop` package. We have made a local copy of this file in the `rbx1_nav/launch` directory so that various parameters can be edited to suit your needs. The following description applies specifically to a Logitech wireless game pad.

To use the joystick teleop node, run the following command:

```
$ roslaunch rbx1_nav joystick_teleop.launch
```

If you get an error such as:

```
[ERROR] [1332893303.227744871]: Couldn't open joystick
/dev/input/js0. Will retry every second.
```

then your joystick or game pad is either not plugged in to a USB port or it is not recognized by Ubuntu. If you do not receive any errors, press the "dead man" button (see Note below) and try moving the joystick or the left-hand toggle stick on the game pad.

NOTE: If you are using a Logitech game pad, you must first press and hold the right index finger button before the robot will respond to the left toggle stick. This button is called the "dead man" switch since the robot will stop moving if you release it.

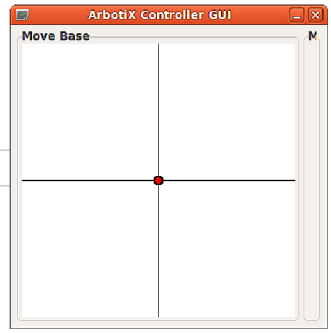
You can edit the `joystick_teleop.launch` file to change the scale factors for linear and angular speed. You can also map the dead man switch to a different button. To find out the numbers corresponding to each button, try the `jstest` program:

```
$ sudo apt-get install joystick
$ jstest /dev/input/js0
```

Then press various buttons and look at the numbers at the bottom of the screen to see which one turns from "off" to "on". Type `Ctrl-C` to exit the test screen.

7.10.3 Using the ArbotiX Controller GUI

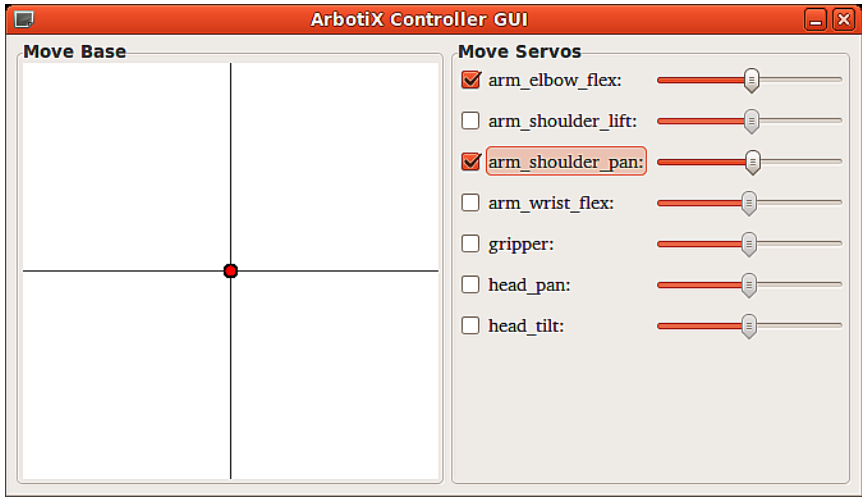
The `arbotix` package by Michael Ferguson includes a very nice graphical display for teleoperating a robot. You can bring up the interface with the following command:




```
$ arbotix_gui
```

You should see a window appear that looks like the image on the right.

To move your robot (either real or simulated), click and hold the red dot with your mouse. Then move move the dot in the direction you want your robot to move. Take care not to move it too far as the robot's speed is controlled by the distance from the origin. If you are using the Pi Robot simulator, you will also see controls for the arm joints as shown below:



To move one of the joints in the simulator, check the box beside the servo name, then use the slider.

7.10.4 TurtleBot Teleoperation Using Interactive Markers

The Turtlebot stack includes a very cool interactive markers package that allows you to move the real robot by dragging controls in `RViz`. Instructions on how to use it are provided on the `turtlebot_interactive_markers` tutorials Wiki page.

7.10.5 Writing your Own Teleop Node

For an example of how to write your own teleop node for a PS3 game controller, see the tutorial on the ROS Wiki.

8. Navigation, Path Planning and SLAM

Now that we have covered the basics of how to control a differential drive robot, we are ready to try one of the more powerful features in ROS; namely, Simultaneous Localization and Mapping or SLAM.

A SLAM-capable robot can build a map of an unknown environment while simultaneously locating itself in that map. Until recently, about the only way to do reliable SLAM was to use a fairly expensive laser scanner to collect the data. With the arrival of the Microsoft Kinect and Asus Xtion cameras, one can now do more affordable SLAM by using the 3D point cloud from the camera to generate a "fake" laser scan. (See the `depthimage_to_laserscan` and `kinect_2d_scanner` packages for two ways to do this.) The TurtleBot is configured to do this out of the box. If you own a TurtleBot, you might want to skip directly to the TurtleBot SLAM tutorial on the ROS Wiki.

Another affordable SLAM robot is the Neato XV-11 vacuum cleaner which includes a 360-degree laser scanner. In fact, you can run the complete Navigation Stack using the XV-11 thanks to the `neato_robot` ROS stack by Michael Ferguson.

In this chapter, we will cover the three essential ROS packages that make up the core of the Navigation Stack:

- `move_base` for moving the robot to a goal pose within a given reference frame
- `gmapping` for creating a map from laser scan data (or simulated laser data from a depth camera)
- `amcl` for localization using an existing map

When we are finished, we will be able to command the robot to go to any location or series of locations within the map, all while avoiding obstacles. Before going further, it is highly recommended that the reader check out the Navigation Robot Setup tutorial on the ROS Wiki. This tutorial provides an excellent overview of the ROS navigation stack. For an even better understanding, check out all of the Navigation Tutorials. And for a superb introduction to the mathematics underlying SLAM, check out Sebastian Thrun's online Artificial Intelligence course on Udacity.

8.1 Path Planning and Obstacle Avoidance using `move_base`

In the previous chapter, we wrote a script to move the robot in a square. In that script, we monitored the `tf` transform between the `/odom` frame and the `/base_link` (or `/base_footprint`) frame to keep track of the distance traveled and the angles rotated.

ROS provides a much more elegant way to do the same thing using the move_base package. (Please see the move_base Wiki page for a full explanation including an excellent diagram of the various components of the package.)

The move_base package implements a ROS action for reaching a given navigation goal. You should be familiar with the basics of ROS actions from the actionlib tutorials on the ROS Wiki. Recall that actions provide feedback as progress is made toward the goal. This means we no longer have to query the odometry topic ourselves to find out if we have arrived at the desired location.

The move_base package incorporates the base_local_planner that combines odometry data with both global and local cost maps when selecting a path for the robot to follow. The global plan or path is computed before the robot starts moving toward the next destination and takes into account known obstacles or areas marked as "unknown". To actually move the robot, the local planner monitors incoming sensor data and chooses appropriate linear and angular velocities for the robot to traverse the current segment of the global path. How these local path segments are implemented over time is highly configurable as we shall see.

8.1.1 Specifying Navigation Goals Using `move_base`

To specify a navigation goal using move_base, we provide a target pose (position and orientation) of the robot with respect to a particular frame of reference. The move_base package uses the MoveBaseActionGoal message type for specifying the goal. To see the definition of this message type, run the command:

```
$ rosmsg show MoveBaseActionGoal
```

which should produce the following output:

```
Header header
  uint32 seq
  time stamp
  string frame_id
actionlib_msgs/GoalID goal_id
  time stamp
  string id
move_base_msgs/MoveBaseGoal goal
  geometry_msgs/PoseStamped target_pose
    Header header
      uint32 seq
      time stamp
      string frame_id
    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
```

As you can see, the goal consists of a standard ROS `header` including a `frame_id`, a `goal_id`, and the `goal` itself which is given as a `PoseStamped` message. The `PoseStamped` message type in turn consists of a `header` and a `pose` which itself consists of a `position` and an `orientation`.

While the `MoveBaseActionGoal` message might seem a little complicated when written out in full, in practice we will only have to set a few of the fields when specifying `move_base` goals.

8.1.2 Configuration Parameters for Path Planning

The `move_base` node requires four configuration files before it can be run. These files define a number of parameters related to the cost of running into obstacles, the radius of the robot, how far into the future the path planner should look, how fast we want the robot to move and so on. The four configuration files can be found in the `config` subdirectory of the `rbx1_nav` package and are called:

- `base_local_planner_params.yaml`
- `costmap_common_params.yaml`
- `global_costmap_params.yaml`

- `local_costmap_params.yaml`

We will describe just a few of the parameters here; in particular, those that you will be most likely to tweak for your own robot. To learn about all of the parameters, please refer to the Navigation Robot Setup on the ROS Wiki as well as the parameters section of the `costmap_2d` and `base_local_planner` Wiki pages.

8.1.2.1 base_local_planner_params.yaml

The values listed here are taken from `base_local_planner_params.yaml` in the `rbx1_nav/config/turtlebot` directory and have been found to work fairly well for the TurtleBot and Pi Robot. (Values that work well in the ArbotiX simulator can be found in the directory `rbx1_nav/config/fake`.)

- `controller_frequency: 3.0` – How many times per second should we update the planning process? Setting this value too high can overload a underpowered CPU. A value of 3 to 5 seems to work fairly well for a typical laptop.

- `max_vel_x: 0.3` – The maximum linear velocity of the robot in meters per second. A value of 0.5 is rather fast for an indoor robot so a value of 0.3 is chosen to be conservative.

- `min_vel_x: 0.05`: – The minimum linear velocity of the robot.

- `max_rotational_vel: 1.0` – The maximum rotational velocity of the robot in radians per second. Don't set this too high or the robot will overshoot its goal orientation.

- `min_in_place_vel_theta: 0.5` – The minimum in-place rotational velocity of the robot in radians per second.

- `escape_vel: -0.1` – Speed used for driving during escapes in meters/sec. Note that it must be negative in order for the robot to actually reverse.

- `acc_lim_x: 2.5` – The maximum linear acceleration in the x direction.

- `acc_lim_y: 0.0` – The maximum linear acceleration in the y direction. We set this to 0 for a differential drive (non-holonomic) robot since such a robot can only move linearly in the `x` direction (as well as rotate).

- `acc_lim_theta: 3.2` – The maximum angular acceleration.

- `holonomic_robot: false` – Unless you have an omni-directional drive robot, this is always set to false.

- `yaw_goal_tolerance: 0.1` – How close to the goal orientation (in radians) do we have to get? Setting this value to small may cause the robot to oscillate near the goal.

- `xy_goal_tolerance: 0.1` – How close (in meters) do we have to get to the goal? If you set the tolerance too small, your robot may try endlessly to make small adjustments around the goal location. **NOTE**: Do not set the tolerance less than the resolution of your map (described in a later section) or your robot will end up spinning in place indefinitely without reaching the goal.

- `pdist_scale: 0.8` – The relative importance of sticking to the global path as opposed to getting to the goal. Set this parameter larger than the gdist_scale to favor following the global path more closely.

- `gdist_scale: 0.4` – The relative importance of getting to the goal rather than sticking to the global path. Set this parameter larger than the `pdist_scale` to favor getting to the goal by whatever path necessary.

- `occdist_scale: 0.1` – The relative importance of avoiding obstacles.

- `sim_time: 1.0` – How many seconds into the future should the planner look? This parameter together with the next (`dwa`) can greatly affect the path taken by your robot toward a goal.

- `dwa: true` – Whether or not to use the Dynamic Window Approach when simulating trajectories into the future. See the Base Local Planner Overview for more details.

8.1.2.2 `costmap_common_params.yaml`

There are only two parameters in this file that you might have to tweak right away for your own robot:

- `robot_radius: 0.165` – For a circular robot, the radius of your robot in meters. For a non-circular robot, you can use the `footprint` parameter instead as shown next. The value here is the radius of the original TurtleBot in meters.

- `footprint: [[x0, y0], [x1, y1], [x2, y2], [x3, y3], etc]` – Each coordinate pair in the list represents a point on the boundary of the robot with the robot's center assumed to be at [0, 0]. The measurement units are meters. The points can be listed in either clockwise or counterclockwise order around the perimeter of the robot.

- `inflation_radius: 0.3` – The radius in meters that obstacles will be inflated in the map. If your robot is having trouble passing through narrow

doorways or other tight spaces, trying reducing this value slightly. Conversely, if the robot keeps running into things, try increasing the value.

8.1.2.3 `global_costmap_params.yaml`

There are a few parameters in this file that you might experiment with depending on the power of your robot's CPU and the reliability of the network connection between the robot and your workstation:

- `global_frame: /map` – For the global cost map, we use the the map frame as the global frame.
- `robot_base_fame: /base_footprint` – This will usually be either `/base_link` or `/base_footprint`. For a TurtleBot it is `/base_footprint`.
- `update_frequency: 1.0` – The frequency in Hz of how often the global map is updated with sensor information. The higher this number, the greater the load on your computer's CPU. Especially for the global map, one can generally get away with a relatively small value of between 1.0 and 5.0.
- `publish_frequency: 0` – For a static global map, there is generally no need to continually publish it.
- `static_map: true` – This parameter and the next are always set to opposite values. The global map is usually static so we set `static_map` to `true`.
- `rolling_window: false` – The global map is generally not updated as the robot moves to we set this parameter to `false`.
- `transform_tolerance: 1.0` – Specifies the delay in seconds that is tolerated between the frame transforms in the `tf` tree. For typical wireless connections between the robot and the workstation, something on the order of 1.0 seconds works OK.

8.1.2.4 `local_costmap_params.yaml`

There are a few more parameters to be considered for the local cost map:

- `global_frame: /odom` – For the local cost map, we use the odometry frame as the global frame
- `robot_base_fame: /base_footprint` – This will usually be either `/base_link` or `/base_footprint`. For a TurtleBot it is `/base_footprint`.

- `update_frequency: 3.0` – How frequently (times per second) do we update the local map based on sensor data. For a really slow computer, you may have to reduce this value.

- `publish_frequency: 1.0` – We definitely want updates to the local map published so we set a non-zero value here. Once per second should be good enough unless your robot needs to move more quickly.

- `static_map: false` – This parameter and the next are always set to opposite values. The local map is dynamically updated with sensor data so we set `static_map` to `false`.

- `rolling_window: true` – The local map is updated within a rolling window defined by the next few parameters.

- `width: 6.0` – The x dimension in meters of the rolling map.

- `height: 6.0` – The y dimension in meters of the rolling map.

- `Resolution: 0.01` – The resolution in meters of the rolling map. This should match the resolution set in the YAML file for your map (explained in a later section).

- `transform_tolerance: 1.0` – Specifies the delay in seconds that is tolerated between the frame transforms in the `tf` tree or the mapping process will temporarily abort. On a fast computer connected directly to the robot, a value 1.0 should work OK. But on slower computers and especially over wireless connections, the tolerance may have to be increased. The tell-tale warning message you will see when the tolerance is set too low will look like the following:

  ```
  [ WARN] [1339438850.439571557]: Costmap2DROS transform
  timeout. Current time: 1339438850.4395, global_pose stamp:
  1339438850.3107, tolerance: 0.0001
  ```

  If you are running the `move_base` node on a computer other than the robot's computer, the above warning can also indicate that the clocks on the two machines are out of sync. Recall from the earlier section on Networking that you can use the `ntpdate` command to synchronize the two clocks. Refer back to that section for details or review the Network Setup pages on the ROS Wiki.

8.2 Testing `move_base` in the ArbotiX Simulator

The `move_base` node requires a map of the environment but the map can simply be a blank square if we just want to test the `move_base` action interface. We will learn how

to create and use real maps in a later section. The rbx1_nav package includes a blank map called blank_map.pgm in the maps subdirectory and its corresponding description file is blank_map.yaml. The launch file for bringing up the move_base node and the blank map is called fake_move_base_blank_map.launch in the launch subdirectory. Let's take a look at it now:

```
<launch>
  <!-- Run the map server with a blank map -->
  <node name="map_server" pkg="map_server" type="map_server" args="$(find
rbx1_nav)/maps/blank_map.yaml"/>

  <!-- Launch move_base and load all navigation parameters -->
  <include file="$(find rbx1_nav)/launch/fake_move_base.launch" />

  <!-- Run a static transform between /odom and /map -->
  <node pkg="tf" type="static_transform_publisher" name="odom_map_broadcaster"
args="0 0 0 0 0 0 /odom /map 100" />
</launch>
```

The comments in the launch file help us understand what is going on. First we run the ROS map_server node with the blank map. Note how the map is specified in terms of its .yaml file which describes the size and resolution of the map itself. Then we include the fake_move_base.launch file (described below) which runs the move_base node and loads all the required configuration parameters that work well in the fake simulator. Finally, since we are using a blank map and our simulated robot has no sensors, the robot cannot use scan data for localization. Instead, we simply set a static identity transform to tie the robot's odometry frame to the map frame which essentially assumes that the odometry is perfect.

Let's take a look at the fake_move_base.launch file now:

```
<launch>

  <node pkg="move_base" type="move_base" respawn="false" name="move_base"
output="screen">
    <rosparam file="$(find rbx1_nav)/config/fake/costmap_common_params.yaml"
command="load" ns="global_costmap" />
    <rosparam file="$(find rbx1_nav)/config/fake/costmap_common_params.yaml"
command="load" ns="local_costmap" />
    <rosparam file="$(find rbx1_nav)/config/fake/local_costmap_params.yaml"
command="load" />
    <rosparam file="$(find rbx1_nav)/config/fake/global_costmap_params.yaml"
command="load" />
    <rosparam file="$(find rbx1_nav)/config/fake/base_local_planner_params.yaml"
command="load" />
  </node>

</launch>
```

The launch file runs the move_base node together with five calls to rosparam to load the parameter files we described earlier. The reason that we load the costmap_common_params.yaml file twice is to set those common parameters in both

the global_costmap namespace and the local_costmap namespace. This is accomplished using the appropriate "ns" attribute in each line.

To try it out in simulation, first fire up the ArbotiX simulator:

```
$ roslaunch rbx1_bringup fake_turtlebot.launch
```

(Replace with your favorite fake robot.)

To launch the move_base node together with the blank map, run the command:

```
$ roslaunch rbx1_nav fake_move_base_blank_map.launch
```

You should see a series of messages similar to:

```
process[map_server-1]: started with pid [304]
process[move_base-2]: started with pid [319]
process[odom_map_broadcaster-3]: started with pid [334]
[ INFO] [1386339148.328543477]: Loading from pre-hydro parameter style
[ INFO] [1386339148.358918577]: Using plugin "static_layer"
[ INFO] [1386339148.465150309]: Requesting the map...
[ INFO] [1386339148.667043125]: Resizing costmap to 600 X 600 at
0.010000 m/pix
[ INFO] [1386339148.766803526]: Received a 600 X 600 map at 0.010000
m/pix
[ INFO] [1386339148.775537537]: Using plugin "obstacle_layer"
[ INFO] [1386339148.777476200]:     Subscribed to Topics:
[ INFO] [1386339148.792662154]: Using plugin "footprint_layer"
[ INFO] [1386339148.802180784]: Using plugin "inflation_layer"
[ INFO] [1386339149.014790405]: Loading from pre-hydro parameter style
[ INFO] [1386339149.033272052]: Using plugin "obstacle_layer"
[ INFO] [1386339149.136896985]:     Subscribed to Topics:
[ INFO] [1386339149.159491227]: Using plugin "footprint_layer"
[ INFO] [1386339149.167455772]: Using plugin "inflation_layer"
[ INFO] [1386339149.312802700]: Created local_planner
base_local_planner/TrajectoryPlannerROS
[ INFO] [1386339149.333349679]: Sim period is set to 0.33
[ INFO] [1386339150.117016385]: odom received!
```

The line highlighted in bold above indicates that our parameter files are not using the new layered costmap feature available in Hydro. Since we don't actually need the layered costmap at this point, we can use the same parameter files as we used in Groovy and move_base will run in a backward compatibility mode.

If you're not already running RViz, bring it up now with the included navigation configuration file:

```
$ rosrun rviz rviz -d `rospack find rbx1_nav`/nav.rviz
```

We are now ready to control the motion of the robot using `move_base` actions rather than simple `Twist` messages. To test it out, let's move the robot 1.0 meters directly forward. Since our robot is starting off at coordinates (0, 0, 0) in both the `/map` frame and the `/base_link` frame, we could use either frame to specify the first movement. However, since the first movement will not place the robot at exactly the goal position and orientation, subsequent goals relative to the `/base_link` frame would start accumulating error. For this reason, it is better to set our goals relative to the static `/map` frame. Referring to the `move_base` goal message syntax listed earlier, the command we want is:

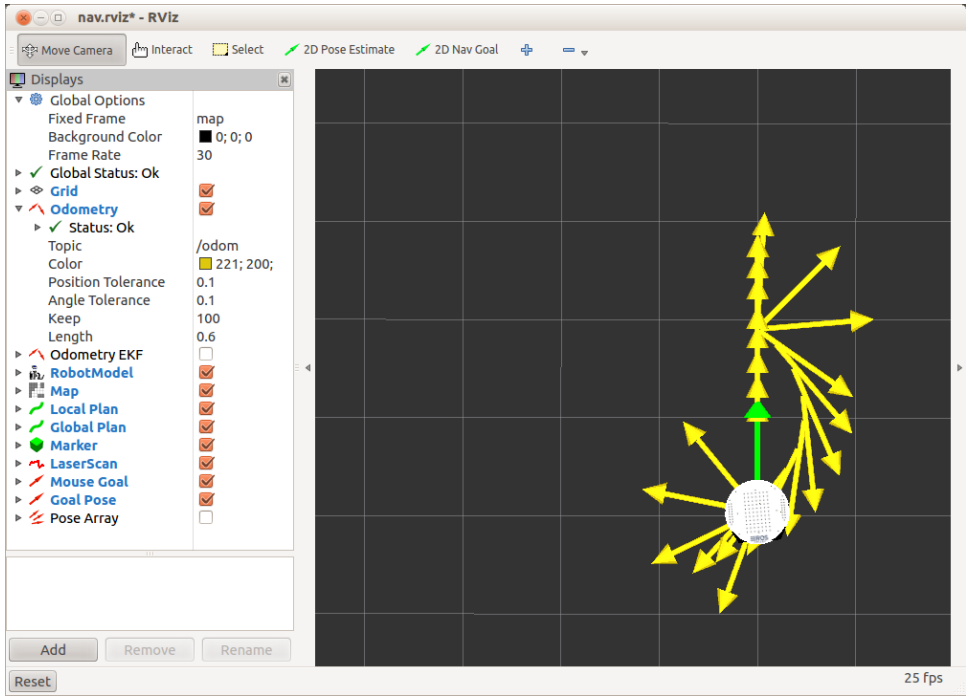
```
$ rostopic pub /move_base_simple/goal geometry_msgs/PoseStamped \
  '{ header: { frame_id: "map" }, pose: { position: { x: 1.0, y: 0, z:
  0 }, orientation: { x: 0, y: 0, z: 0, w: 1 } } }'
```

You should see the robot move about 1 meter forward in `RViz`. (The large green arrow that appears is the target pose.) Note that the orientation in the command above is specified as a quaternion and a quaternion with components (0, 0, 0, 1) specifies an identity rotation.

To move the robot back to the starting point, first type `Ctrl-C` to end the previous command. Then send the coordinates (0, 0, 0) in the `map` frame with the robot pointing upward:

```
$ rostopic pub /move_base_simple/goal geometry_msgs/PoseStamped \
  '{ header: { frame_id: "map" }, pose: { position: { x: 0, y: 0, z:
  0 }, orientation: { x: 0, y: 0, z: 0, w: 1 } } }'
```

The view in `RViz` after both commands should look something like the following:



(Note: the robot might make the return trip by turning left instead of left as shown above. It is a bit of a coin toss depending on the exact orientation the robot has at the first goal location.) As the robot moves, a thin green line (which might be difficult or impossible to see if the **Odometry** arrows are being displayed) indicates the global path planned for the robot from the starting position to the goal. A shorter red line is the locally planned trajectory that gets updated more often as the robot makes progress toward the destination.

The display of the global and local paths can be turned off and on using the check boxes beside the appropriate displays in the **Displays** panel on the left in `RViz`. You can also change the colors of the paths using the appropriate property value for each display. To view the global and local paths more clearly, turn off the displays for **Odometry**, **Goal Pose** and **Mouse Pose**, then re-run the two `move_base` commands above.

You will notice that the local trajectory in this case follows an arc that is fairly far off the straight line global path. This is due partly to the fact that there are no obstacles in the map, so the planner selects a nice smooth turning trajectory. It also reflects our choice of the two parameters `pdist_scale` (0.4) and `gdist_scale` (0.8) and the max linear speed of the robot (`max_vel_x`). To make the robot follow the global path more closely, we can use `rqt_reconfigure` to dynamically increase the `pdist_scale` parameter)or decrease `max_vel_x`) without having to restart all of the running nodes.

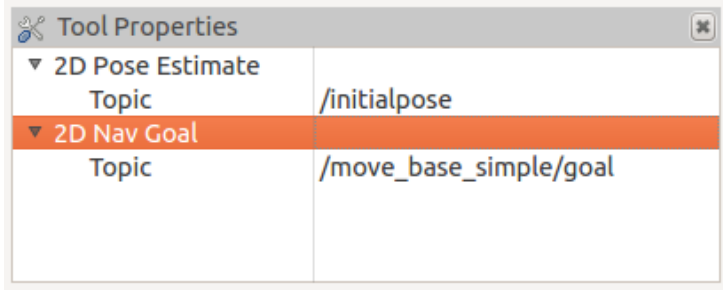
To try it out, open another terminal window and bring up `rqt_reconfigure`:

```
$ rosrun rqt_reconfigure rqt_reconfigure
```

Next, open the **`move_base`** group and select the **`TrajectoryPlannerROS`** node, then set the `pdist_scale` parameter to something higher like 2.5 and leave the `gdist_scale` at 0.8 . Then turn off the **Odometry** arrows in `RViz` (if they are still on) and run the two `move_base` commands again to see the effect. You should see that the robot now more closely follows the global planned path (green) more closely. To tighten up the trajectory even more, slow the robot down by changing its `max_vel_x` parameter from 0.5 to something like 0.2.

8.2.1 Point and Click Navigation in `RViz`

We can also specify a goal pose using the mouse in `RViz`. If you launched `RViz` with the `nav.rviz` file as described above, you should be good to go. However, just to check, if you don't see the **Tool Properties** window on the right side of the `RViz` screen, click on the **Panels** menu and select **Tool Properties**. A pop-up window should appear that looks like this:



Under the **2D Nav Goal** category, the topic should be listed as `/move_base_simple/goal`. If not, type that topic name into the field now.

If you had to make any changes, click on the **File** menu in `RViz` and choose **Save Config**. When you are finished, you can close the **Tool Properties** window by clicking the little x in the upper right corner.

With these preliminaries out of the way, we can now use the mouse to move the robot. Click the **Reset** button to clear any left over odometry arrows. Next, click on the **2D Nav Goal** button near the top of the `RViz` screen. Then click and hold the mouse somewhere on the grid where you'd like the robot to end up. If you move the mouse slightly while holding down the button, a big green arrow will appear. Rotate the arrow to indicate the goal orientation of the robot. Now release the mouse button and

move_base should guide the robot to the goal. Try setting a few different goals and observe the changes in the global and local paths. As in the previous section, you can also bring up rqt_reconfigure and change the relative values of the pdist_scale and gdist_scale parameters as well as max_vel_x to change the behavior of the local and global paths.

If you look back in the terminal window where you launched RViz, you'll see a number of [INFO] messages indicating the poses (position and orientation) that you set with the mouse. This information is useful if you ever want to write down a particular set of coordinates on the map and use them later as a goal location for the robot as we will see later.

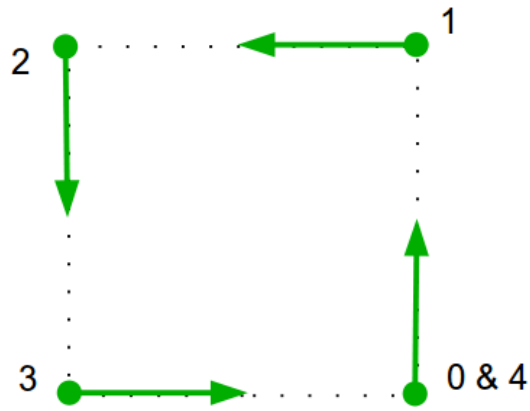
8.2.2 Navigation Display Types for RViz

As you can see from the earlier RViz screen shot, the nav.rviz configuration file we used includes a number of display types typically used for navigation. In addition to the **Odometry** and **Robot Model** displays, we also have one for the **Map**, the **Local Plan** and **Global Plan**, a **Laser Scan**, **Inflated Obstacle** cells, the **Goal Pose** (including one for goals set using the mouse) and one for **Markers**. (We will talk about the **Pose Array** later on when we discuss localization.)

For a complete list of display types used with ROS navigation see the tutorial on the ROS Wiki called Using rviz with the Navigation Stack. The tutorial includes an excellent demonstration video which is a must-see you are new to either RViz or ROS navigation. Note however that at the time of this writing, the video uses the pre-Hydro version of RViz so the look and feel will be a little different.

8.2.3 Navigating a Square using move_base

We are now ready to move our robot in a square using move_base rather than simple Twist messages. The script move_base_square.py in the nodes subdirectory does the work. The script simply cycles through four target poses, one for each corner of the square. The four target poses are shown by the arrows in the figure below:



Remember that a "pose" in ROS means both a position and orientation.

To make sure we are starting with a clean slate, terminate any launch files used in the previous section by typing `Ctrl-C` in the appropriate terminal windows. Then fire up the fake TurtleBot and `move_base` node as before:

```
$ roslaunch rbx1_bringup fake_turtlebot.launch
```

And in another terminal:

```
$ roslaunch rbx1_nav fake_move_base_blank_map.launch
```

Then make sure you have `RViz` up with the `nav.rivz` configuration file:

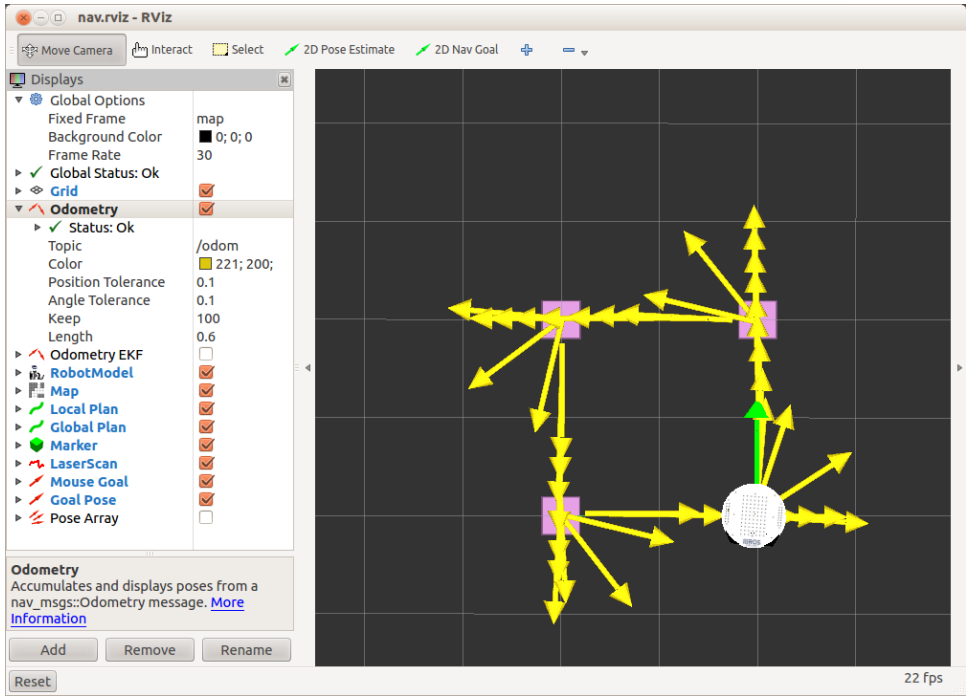
```
$ rosrun rviz rviz -d `rospack find rbx1_nav`/nav.rviz
```

If you already had `RViz` running, clear any left over **Odometry** arrows by clicking the **Reset** button.

Finally, run the `move_base_square.py` script:

```
$ rosrun rbx1_nav move_base_square.py
```

When the scripts completes, the view in `RViz` should look something like the following:



The small squares indicate the locations of the four corner points we want to hit. (The first square is hidden beneath the robot in the picture above.) As you can see, the square trajectory is not bad even though all we specified were the four corner poses. One could refine the trajectory even further by tweaking the configuration parameters for the base local planner, or setting intermediate waypoints between the corners. But the real purpose of `move_base` is not to follow a precise path, but to reach arbitrary goal locations while avoiding obstacles. Consequently, it works very well for getting a robot around the house or office without running into things as we shall see in the next section.

Let's now take a look at the code.

Link to source: move_base_square.py

```
1.  #!/usr/bin/env python
2.
3.  import rospy
4.  import actionlib
5.  from actionlib_msgs.msg import *
6.  from geometry_msgs.msg import Pose, Point, Quaternion, Twist
7.  from move_base_msgs.msg import MoveBaseAction, MoveBaseGoal
8.  from tf.transformations import quaternion_from_euler
9.  from visualization_msgs.msg import Marker
10. from math import radians, pi
```

```
11.
12.     class MoveBaseSquare():
13.         def __init__(self):
14.             rospy.init_node('nav_test', anonymous=False)
15.
16.             rospy.on_shutdown(self.shutdown)
17.
18.             # How big is the square we want the robot to navigate?
19.             square_size = rospy.get_param("~square_size", 1.0) # meters
20.
21.             # Create a list to hold the target quaternions (orientations)
22.             quaternions = list()
23.
24.             # First define the corner orientations as Euler angles
25.             euler_angles = (pi/2, pi, 3*pi/2, 0)
26.
27.             # Then convert the angles to quaternions
28.             for angle in euler_angles:
29.                 q_angle = quaternion_from_euler(0, 0, angle, axes='sxyz')
30.                 q = Quaternion(*q_angle)
31.                 quaternions.append(q)
32.
33.             # Create a list to hold the waypoint poses
34.             waypoints = list()
35.
36.             # Append each of the four waypoints to the list.  Each waypoint
37.             # is a pose consisting of a position and orientation in the map
frame.
38.             waypoints.append(Pose(Point(square_size, 0.0, 0.0),
quaternions[0]))
39.             waypoints.append(Pose(Point(square_size, square_size, 0.0),
quaternions[1]))
40.             waypoints.append(Pose(Point(0.0, square_size, 0.0),
quaternions[2]))
41.             waypoints.append(Pose(Point(0.0, 0.0, 0.0), quaternions[3]))
42.
43.             # Initialize the visualization markers for RViz
44.             self.init_markers()
45.
46.             # Set a visualization marker at each waypoint
47.             for waypoint in waypoints:
48.                 p = Point()
49.                 p = waypoint.position
50.                 self.markers.points.append(p)
51.
52.             # Publisher to manually control the robot (e.g. to stop it)
53.             self.cmd_vel_pub = rospy.Publisher('cmd_vel', Twist)
54.
55.             # Subscribe to the move_base action server
56.             self.move_base = actionlib.SimpleActionClient("move_base",
MoveBaseAction)
57.
58.             rospy.loginfo("Waiting for move_base action server...")
59.
60.             # Wait 60 seconds for the action server to become available
61.             self.move_base.wait_for_server(rospy.Duration(60))
62.
```

```
63.             rospy.loginfo("Connected to move base server")
64.             rospy.loginfo("Starting navigation test")
65.
66.             # Initialize a counter to track waypoints
67.             i = 0
68.
69.             # Cycle through the four waypoints
70.             while i < 4 and not rospy.is_shutdown():
71.                 # Update the marker display
72.                 self.marker_pub.publish(self.markers)
73.
74.                 # Initialize the waypoint goal
75.                 goal = MoveBaseGoal()
76.
77.                 # Use the map frame to define goal poses
78.                 goal.target_pose.header.frame_id = 'map'
79.
80.                 # Set the time stamp to "now"
81.                 goal.target_pose.header.stamp = rospy.Time.now()
82.
83.                 # Set the goal pose to the i-th waypoint
84.                 goal.target_pose.pose = waypoints[i]
85.
86.                 # Start the robot moving toward the goal
87.                 self.move(goal)
88.
89.                 i += 1
90.
91.         def move(self, goal):
92.             # Send the goal pose to the MoveBaseAction server
93.             self.move_base.send_goal(goal)
94.
95.             # Allow 1 minute to get there
96.             finished_within_time =
self.move_base.wait_for_result(rospy.Duration(60))
97.
98.             # If we don't get there in time, abort the goal
99.             if not finished_within_time:
100.                self.move_base.cancel_goal()
101.                rospy.loginfo("Timed out achieving goal")
102.            else:
103.                # We made it!
104.                state = self.move_base.get_state()
105.                if state == GoalStatus.SUCCEEDED:
106.                    rospy.loginfo("Goal succeeded!")
107.
108.        def init_markers(self):
109.            # Set up our waypoint markers
110.            marker_scale = 0.15
111.            marker_lifetime = 0 # 0 is forever
112.            marker_ns = 'waypoints'
113.            marker_id = 0
114.            marker_color = {'r': 1.0, 'g': 0.0, 'b': 0.0, 'a': 1.0}
115.
116.            # Define a marker publisher.
117.            self.marker_pub = rospy.Publisher('waypoint_markers', Marker)
118.
```

```
            # Initialize the marker points list.
            self.markers = Marker()
            self.markers.ns = marker_ns
            self.markers.id = marker_id
            self.markers.type = Marker.CUBE_LIST
            self.markers.action = Marker.ADD
            self.markers.lifetime = rospy.Duration(marker_lifetime)
            self.markers.scale.x = marker_scale
            self.markers.scale.y = marker_scale
            self.markers.color.r = marker_color['r']
            self.markers.color.g = marker_color['g']
            self.markers.color.b = marker_color['b']
            self.markers.color.a = marker_color['a']

            self.markers.header.frame_id = 'map'
            self.markers.header.stamp = rospy.Time.now()
            self.markers.points = list()

        def shutdown(self):
            rospy.loginfo("Stopping the robot...")
            # Cancel any active goals
            self.move_base.cancel_goal()
            rospy.sleep(2)
            # Stop the robot
            self.cmd_vel_pub.publish(Twist())
            rospy.sleep(1)

    if __name__ == '__main__':
        try:
            MoveBaseSquare()
        except rospy.ROSInterruptException:
            rospy.loginfo("Navigation test finished.")
```

Let's now examine the key lines of the script.

```
        # First define the corner orientations as Euler angles
        euler_angles = (pi/2, pi, 3*pi/2, 0)

        # Then convert the angles to quaternions
        for angle in euler_angles:
            q_angle = quaternion_from_euler(0, 0, angle, axes='sxyz')
            q = Quaternion(*q_angle)
            quaternions.append(q)
```

Here we define the target orientations at the four corners of the square, first as Euler angles which are easier to visualize on the map, and then converted to quaternions.

```
        waypoints.append(Pose(Point(square_size, 0.0, 0.0),
    quaternions[0]))
        waypoints.append(Pose(Point(square_size, square_size, 0.0),
    quaternions[1]))
        waypoints.append(Pose(Point(0.0, square_size, 0.0),
    quaternions[2]))
        waypoints.append(Pose(Point(0.0, 0.0, 0.0), quaternions[3]))
```

Next we create the four waypoint poses by combining the rotations with the coordinates of the four corners.

```
43.         # Initialize the visualization markers for RViz
44.         self.init_markers()
45.
46.         # Set a visualization marker at each waypoint
47.         for waypoint in waypoints:
48.             p = Point()
49.             p = waypoint.position
50.             self.markers.points.append(p)
```

While not covered in this volume, setting visualization markers in `RViz` is fairly straightforward and a number of existing tutorials can be found here. The script places a red square at each target corner and the function `self.init_markers()`, defined toward the end of the script, sets up the marker shapes, sizes and colors. We then append the four markers to a list to be used later.

```
56.         self.move_base = actionlib.SimpleActionClient("move_base",
    MoveBaseAction)
```

Here we define a `SimpleActionClient` which will send goals to the `move_base` action server.

```
61.         self.move_base.wait_for_server(rospy.Duration(60))
```

Before we can start sending goals, we have to wait for the `move_base` action server to become available. We give it 60 seconds before timing out.

```
69.         # Cycle through each waypoint
70.         while i < 4 and not rospy.is_shutdown():
71.             # Update the marker display
72.             self.marker_pub.publish(self.markers)
73.
74.             # Initialize the waypoint goal
75.             goal = MoveBaseGoal()
76.
77.             # Use the map frame to define goal poses
78.             goal.target_pose.header.frame_id = 'map'
79.
80.             # Set the time stamp to "now"
81.             goal.target_pose.header.stamp = rospy.Time.now()
82.
83.             # Set the goal pose to the i-th waypoint
84.             goal.target_pose.pose = waypoints[i]
85.
86.             # Start the robot moving toward the goal
87.             self.move(goal)
88.
89.             i += 1
```

We then enter our main loop, cycling through each of the four waypoints. First we publish the markers to indicate the four goal poses. (This has to be done on each cycle to keep them visible throughout the movement.) Then we initialize the `goal` variable as a `MoveBaseGoal` action type. Next we set the goal `frame_id` to the `map` frame and the time stamp to the current time. Finally, we set the goal pose to the current waypoint and send the goal to the `move_base` action server using the helper function `move()`, which we describe next.

```
91.     def move(self, goal):
92.         # Send the goal pose to the MoveBaseAction server
93.         self.move_base.send_goal(goal)
94.
95.         # Allow 1 minute to get there
96.         finished_within_time =
    self.move_base.wait_for_result(rospy.Duration(60))
97.
98.         # If we don't get there in time, abort the goal
99.         if not finished_within_time:
100.            self.move_base.cancel_goal()
101.            rospy.loginfo("Timed out achieving goal")
102.        else:
103.            # We made it!
104.            state = self.move_base.get_state()
105.            if state == GoalStatus.SUCCEEDED:
106.                rospy.loginfo("Goal succeeded!")
```

The `move()` helper function takes a goal as input, sends it to the `MoveBaseAction` server, then waits up to 60 seconds for the robot to get there. A success or failure message is then displayed on the screen. Note how much easier this approach is to code than our earlier `nav_square.py` script that used `Twist` messages directly. In particular, we no longer have to monitor the odometry data directly. Instead the `move_base` action server takes care of it for us.

8.2.4 Avoiding Simulated Obstacles

One of the more powerful features of `move_base` is the ability to avoid obstacles while still getting to the goal. Obstacles can be a static part of the current map (such as a wall) or they can appear dynamically such as when someone walks in front of the robot. The base local planner can recompute the robot's path on the fly to keep the robot from striking objects yet still allowing it to reach its destination.

To illustrate the process, will will load a new map with a pair of obstacles in the way of the robot. We will then run the `move_base_square.py` script again to see if the base local planner can find a path around the obstacles and still guide the robot to the four goal locations.

First terminate any fake robots you might still have running as well as the earlier `fake_move_base_blank_map.launch` file by typing `Ctrl-C` in the appropriate terminal window(s). Then run the following commands:

```
$ roslaunch rbx1_bringup fake_turtlebot.launch
```

followed by:

```
$ rosparam delete /move_base
```

and

```
$ roslaunch rbx1_nav fake_move_base_map_with_obstacles.launch
```

The first command clears all existing `move_base` parameters which is less drastic than killing and restarting `roscore`. And the second command brings up the `move_base` node along with a map with a pair of obstacles near the center.

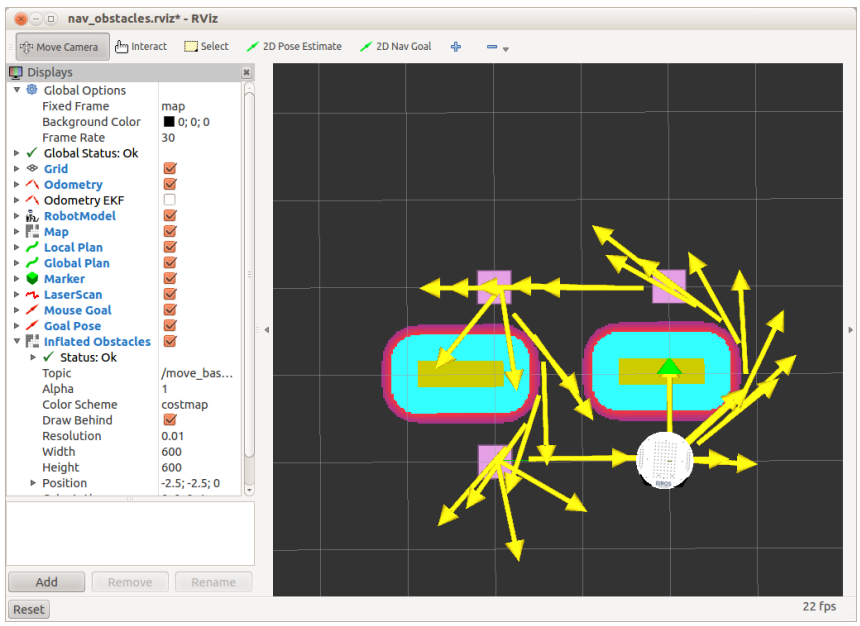
If you are already running `RViz`, close it down and bring it up again with the `nav_obstacles.rviz` config file:

```
$ rosrun rviz rviz -d `rospack find rbx1_nav`/nav_obstacles.rviz
```

When the `move_base` node is up and you see the obstacles in `RViz`, click the **Reset** button in to refresh the display, then run the `move_base_square.py` script:

```
$ rosrun rbx1_nav move_base_square.py
```

When the script completes, the view in `RViz` should look something like this:



The obstacles are represented by the horizontal yellow bars while the multi-colored oval-shaped regions around the obstacles reflect the inflation radius (0.3 meters) we have set to provide a safety buffer. As you can see, the simulated robot has no difficulty getting to the goal locations while avoiding the obstacles. Even better, on the way to the third corner of the square (lower left), the base local planner chose the shorter route between the two obstacles rather than going around the outside.

During the simulation you will notice a thin green line that indicates the global path planned for the next goal location. The shorter red line that appears to shoot out in front of the robot is the local path which can adapt more quickly to local conditions. Since we aren't simulating any sensors, the robot is actually running blind and is relying only on the static map and its (fake) odometry.

To get the robot around these tightly spaced obstacles, it was necessary to change a few navigation parameters from the values we used with the blank map. If you look at the launch file `fake_move_base_map_with_obstacles.launch` in the `rbx1_nav/launch` directory, you will see that it brings up the `move_base` node by including the additional launch file `fake_move_base_obstacles.launch`. This file is almost identical to the launch file `fake_move_base.launch` that we used in the previous section except that we have added a line that overrides a few of the navigation parameters. The launch file looks like this:

```
<launch>
  <node pkg="move_base" type="move_base" respawn="false" name="move_base"
output="screen">
    <rosparam file="$(find rbx1_nav)/config/fake/costmap_common_params.yaml"
command="load" ns="global_costmap" />
    <rosparam file="$(find rbx1_nav)/config/fake/costmap_common_params.yaml"
command="load" ns="local_costmap" />
    <rosparam file="$(find rbx1_nav)/config/fake/local_costmap_params.yaml"
command="load" />
    <rosparam file="$(find rbx1_nav)/config/fake/global_costmap_params.yaml"
command="load" />
    <rosparam file="$(find rbx1_nav)/config/fake/base_local_planner_params.yaml"
command="load" />
    <rosparam file="$(find rbx1_nav)/config/nav_obstacles_params.yaml"
command="load" />
  </node>
</launch>
```

The key line is highlighted in bold faced and it loads the parameter file `nav_obstacles_params.yaml`. That file in turn looks like this:

```
TrajectoryPlannerROS:
  max_vel_x: 0.3
  pdist_scale: 0.8
  gdist_scale: 0.4
```

As you can see, the file contains a handful of parameters that we used in the `base_local_planner_params.yaml` file but sets them to values that work better for avoiding obstacles. In particular, we have slowed down the max speed from 0.5 m/s to 0.3 m/s and we have reversed the relative weights on `pdist_scale` and `gdist_scale` so that now we give more weight to following the planned path. We could have simply created a whole new `base_local_planner_params.yaml` with these few changes, but this way we can keep the bulk of our parameters in the main file and just override them as need with special file "snippets" like the one used here.

8.2.5 Setting Manual Goals with Obstacles Present

You can also use the mouse as we have done before to set navigation goals when obstacles are present. As before, first click on the **2D Nav Goal** button near the top of the `RViz` window, then click somewhere on the map. If you try setting a goal inside one of the obstacles or even too close to one, the base local planner will abort any attempt to get there. (An abort message is also displayed in the terminal window in which you launched the `fake_move_base_map_with_obstacles.launch` file.) As usual, any time you want to clear the markers on the screen, just click the **Reset** button.

8.3 Running `move_base` on a Real Robot

If you have a ROS-compatible robot with a mobile base, you can try out the `move_base_square.py` script in the real world. If you are using an original TurtleBot (using an iRobot Create base), you can probably get away with the `move_base` configuration included in the `rbx1_nav/config/turtlebot` directory. Otherwise, you might have to change a few parameters to better match your robot. For a good introduction on tuning the navigation parameters, see the Navigation Tuning Guide on the ROS Wiki.

8.3.1 Testing `move_base` without Obstacles

To make sure we are running with a clean slate, `Ctrl-C` out of any running ROS launch files or nodes including `roscore`. Then start up `roscore` again and get your robot ready.

Make sure your robot has enough room to trace out the square (you can shorten the sides in the script if you like). When you are ready, fire up the required launch file(s) to connect to your robot. For an original TurtleBot (using an iRobot Create base). you would run:

```
$ roslaunch rbx1_bringup turtlebot_minimal_create.launch
```

(Or use your own launch file if you have created one to store your calibration parameters.)

If you are using a TurtleBot, you might also run the `odom_ekf.py` script to be able to see the TurtleBot's combined odometry frame in `RViz`. You can skip this if you are not using a TurtleBot. This launch file should be run on the TurtleBot's laptop:

```
$ roslaunch rbx1_bringup odom_ekf.launch
```

Next, launch the `move_base` node with a blank map. **NOTE**: This is a different launch file than the one we used for the simulated robot. This particular launch file loads a set of navigation parameters that should work fairly well with the original TurtleBot.

```
$ roslaunch rbx1_nav tb_move_base_blank_map.launch
```

Now bring up `RViz` on your workstation with the `nav` configuration file. You'll probably want to run this on your desktop machine rather than the TurtleBot's laptop:

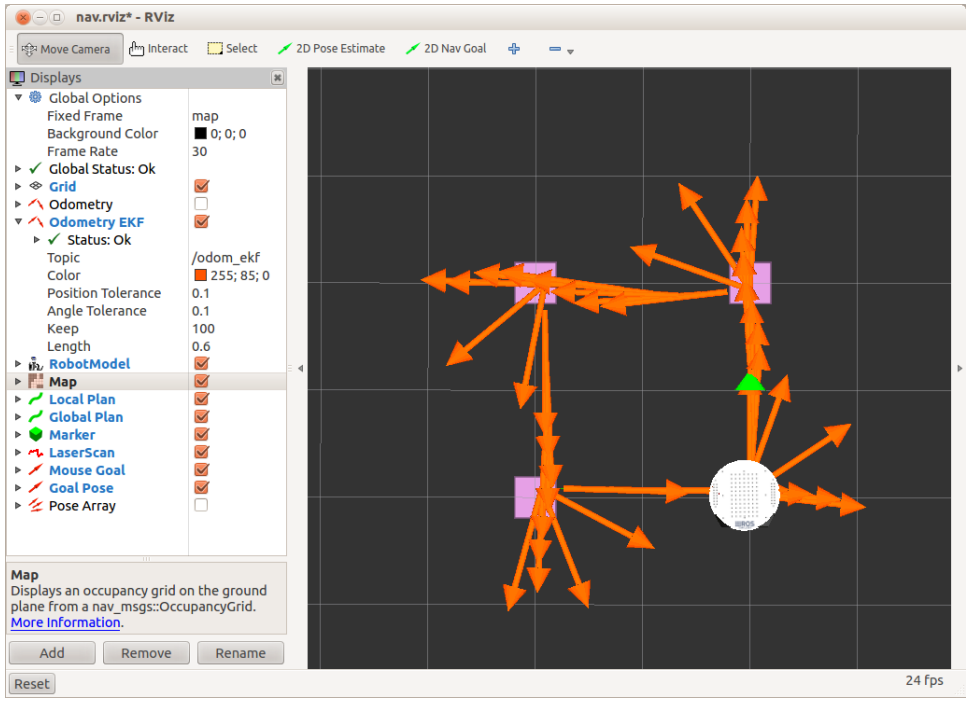
```
$ rosrun rviz rviz -d `rospack find rbx1_nav`/nav.rviz
```

Finally, run the `move_base_square.py` script. You can run this either on the robot's laptop or your desktop:

```
$ rosrun rbx1_nav move_base_square.py
```

If all goes well, your robot should move from one corner of the square to the next and then stop when it has come back to the start.

The following image shows the result when running my own TurtleBot on a low-ply carpet:



In the real world, the robot came withing 27 cm of the starting position and approximately 5 degrees out of alignment.

8.3.2 Avoiding Obstacles using a Depth Camera as a Fake Laser

Our final test will use a depth camera such as the Kinect or Xtion Pro to emulate a kind of fake laser scanner to detect obstacles so that `move_base` can plan a path around them while guiding the robot to a goal. If you look at the parameter file `costmap_common_params.yaml` in the `rbx1_nav/config/turtlebot` directory, you'll see the following pair of lines:

```
observation_sources: scan
scan: {data_type: LaserScan, topic: /scan, marking: true,
clearing: true, expected_update_rate: 0}
```

The first line tells the base local planner to expect sensor data from a source called "`scan`" and the second line indicates that the "`scan`" source is of type `LaserScan` and it publishes its data on the `/scan` topic. Setting the flags `marking` and `clearing` to `True` means that the laser data can be used to mark areas of the local cost map as occupied or free as the robot moves about.

The expected_update_rate determines how often we should expect a reading from the scan. A value of 0 allows an infinite time between readings and allows us to use the navigation stack with or without an active laser scanner. However, if you typically operate your robot using a laser scanner or the fake laser described here, it is better to set this value to something like 0.3 so that the navigation stack will stop the robot if the laser scanner stops working.

The following steps should work if you are using the original TurtleBot. If you don't already have the robot's drivers running, fire them up now:

```
$ roslaunch rbx1_bringup turtlebot_minimal_create.launch
```

(Or use your own launch file if you have created one to store your calibration parameters.)

Make sure your Kinect or Xtion is plugged into a USB port on the robot, then log into the TurtleBot's laptop and run:

```
$ roslaunch rbx1_bringup fake_laser.launch
```

The fake_laser.launch file first runs the openni_driver.launch file to bring up the drivers for the Kinect or Asus camera. It then runs a depthimage_to_laserscan node to convert the depth image into an equivalent laser scan message that is published on the /scan topic.
Next, launch the TurtleBot move_base node with a blank map. **NOTE**: This is a different launch file than the one we used for the simulated robot. This particular launch file loads a set of navigation parameters that should work fairly well with the original TurtleBot.

```
$ roslaunch rbx1_nav tb_move_base_blank_map.launch
```

If you are already running RViz, close it down and bring it up again with the nav_obstacles.rviz config file:

```
$ rosrun rviz rviz -d `rospack find rbx1_nav`/nav_obstacles.rviz
```

Assuming there is at least one object within the camera's range, you should see the laser scan appear in RViz. The configuration file (nav_obstacles.rviz) we are using for RViz includes a **Laser Scan** display which is set to the /scan topic by default. You can verify this by scrolling down the **Displays** panel on the left of RViz. The light blue areas around the laser scan are due to the **Inflated Obstacles** display which subscribes to the topic /move_base/local_costmap/costmap and reflects the inflation radius parameter we have set in the common_costmap_params.yaml file to provided a safety buffer around obstacles.

So now we are ready to test how well the robot can avoid obstacles on the way to a goal. Pick a point a few meters away from the robot and set the **2D Nav Goal** in `RViz` using the mouse as we have done previously. As the robot makes it way toward the goal, walk into its path a few feet ahead of it. As soon as you enter the view of the (fake) laser scan, the robot should veer to go around you, then continue on to the goal location.

Depending on the speed of your robot's computer, you may notice that the robot comes fairly close to your feet even with the inflation radius set to 0.5. There are two possible reasons for this. The first is the narrow field of view of the Kinect or Xtion cameras—about 57 degrees. So as the robot begins moving past an obstacle, the object quickly passes out of the field of view and the path planner figures it can start turning back toward the goal. Second, both RGB-D cameras are blind to depths within about 50cm (approx 2 feet) of the projector. This means it is possible for an obstacle to be completely invisible even if it is situated directly in front of the robot. A real laser scanner like a Hokuyo or the one found on the Neato XV-11 has a much wider field of view (240 – 360 degrees) and can sense objects as close as a few centimeters.

8.4 Map Building using the `gmapping` Package

Now that we understand how to use `move_base`, we can replace the blank map with a real map of the robot's environment. A map in ROS is simply a bitmap image representing an occupancy grid where white pixels indicate free space, black pixels represent obstacles, and grey pixels stand for "unknown". You can therefore draw a map yourself in any graphics program or use a map that someone else has created. However, if your robot is equipped with a laser scanner or depth camera, the robot can create its own map as it moves around the target area. If your robot has neither of these pieces of hardware, you can use the test map include in `rbx1_nav/maps`. In this case, you might want to skip directly to the next chapter on doing localization using an existing map.

The ROS `gmapping` package contains the `slam_gmapping` node that does the work of combining the data from laser scans and odometry into an occupancy map. The usual strategy is first to teleoperate the robot around an area while recording the laser and odometry data in a `rosbag` file. Then we run the `slam_gmapping` node against the recorded data to generate a map. The advantage of recording the data first is that you can generate any number of test maps later using the same data but with different parameters for `gmapping`. In fact, since we no longer need the robot at that point, you could do this testing anywhere you have a computer with ROS and your rosbag data.

Lets take each of these steps in turn.

8.4.1 Laser Scanner or Depth Camera?

To use the `gmapping` package, our robot needs either a laser scanner or a depth camera like the Kinect or Xtion. If you have a laser scanner, you will need the ROS laser drivers which you can install using:

```
$ sudo apt-get install ros-hydro-laser-*
```

In this case, your robot's startup files will need to launch the driver node for your particular scanner. For example, Pi Robot uses a Hokuyo laser scanner and the corresponding launch file looks like this:

```
<launch>
  <node name="hokuyo" pkg="hokuyo_node" type="hokuyo_node">
    <param name="min_ang" value="-1.7" />
    <param name="max_ang" value="1.7" />
    <param name="hokuyo_node/calibrate_time" value="true" />
    <param name="frame_id" value="/base_laser" />
  </node>
</launch>
```

You can find this `hokuyo.launch` file in the `rbx1_bringup/launch` subdirectory.

If you do not have a laser scanner but you do have a depth camera like the Kinect or Xtion, then you can use the "fake laser scanner" found in the `rbx1_bringup` package. To see how it works, take a look at the `fake_laser.launch` file as follows:

```
$ roscd rbx1_bringup/launch
$ cat fake_laser.launch
```

which should result in the following output:

```
<launch>
  <!-- Launch the OpenNI drivers -->
  <include file="$(find rbx1_bringup)/launch/openni_driver.launch" />

  <!-- Run the depthimage_to_laserscan node -->
  <node pkg="depthimage_to_laserscan" type="depthimage_to_laserscan"
name="depthimage_to_laserscan" output="screen">
    <remap from="image" to="/camera/depth/image_raw" />
    <remap from="camera_info" to="/camera/depth/camera_info" />
    <remap from="scan" to="/scan" />
</node>

</launch>
```

The `fake_laser.launch` file begins by launching the drivers for the Kinect or Asus camera by including the `rbx1_bringup/openni_driver.launch` file. It then runs a `depthimage_to_laserscan` node to convert the depth image into an equivalent laser scan message. The only catch is that the scan is rather narrow—about 57 degrees—which is the width of the field of view of the Kinect or Xtion. A typical laser scanner

has a width around 240 degrees or more. Nonetheless, the fake scan can be used to create fairly good maps.

To try it out on an original TurtleBot, log into the robot's laptop and run:

```
$ roslaunch rbx1_bringup turtlebot_minimal_create.launch
```

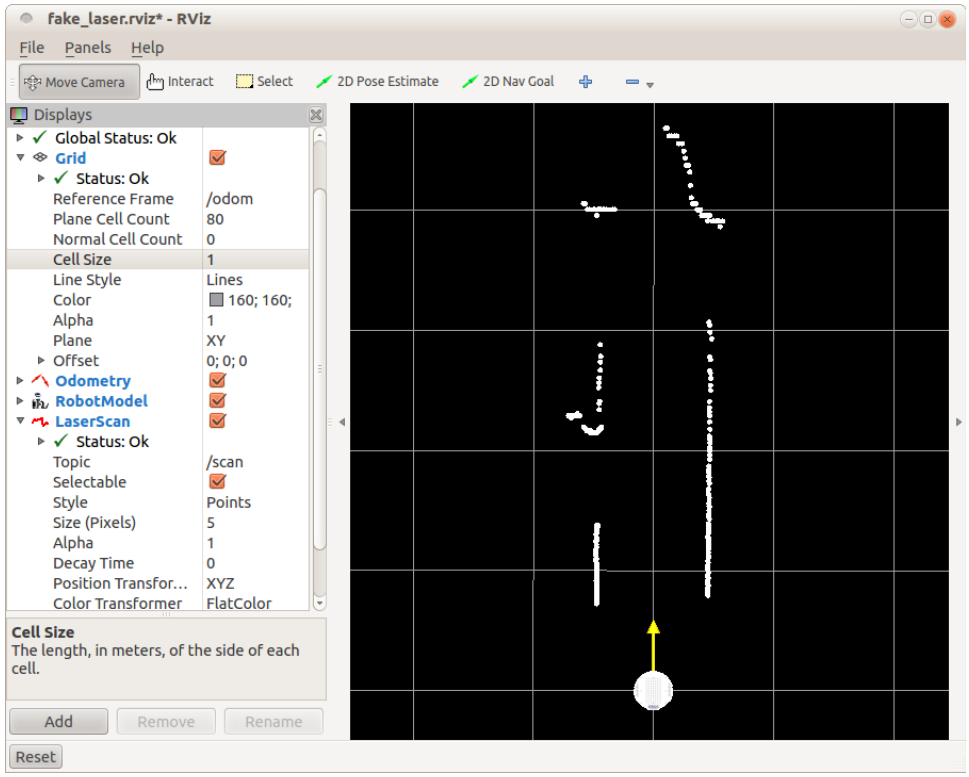
Next, open another terminal window and log into the TurtleBot's laptop (or the computer attached to your depth camera) and run the command:

```
$ roslaunch rbx1_bringup fake_laser.launch
```

Back on your desktop workstation, bring up `RViz` with the included `fake_laser` configuration file:

```
$ rosrun rviz rviz -d `rospack find rbx1_nav`/fake_laser.rviz
```

If all goes well, you should see something like the following in `RViz`:



This particular screen shot was taken with the robot part way down a hallway with open doorways on the left and far right, another open door at the end of the hallway, and an opening on the far left toward another hallway. The grid lines are 1 meter apart. Notice how we are subscribing to the `/scan` topic under the **Laser Scan** Display. If you don't see the scan points, click on the **Topic** field and check that the scan is being published and is visible on your workstation.

8.4.2 Collecting and Recording Scan Data

Whether using a laser scanner or depth camera, the strategy for collecting scan data is the same. First we fire up any launch files required to control our robot's mobile base. Then we launch the drivers for our laser scanner or fake laser together with a teleop node so we can remotely control the robot. Finally, we start recording data into a ROS bag file and proceed to drive the robot around the target area.

If you own a TurtleBot, you can simply follow the official TurtleBot gmapping Tutorial. However, if you need a few more details, you can follow along here as well.

It is probably a good idea to start your ROS environment with a clean slate. So kill off any running launch files and even `roscore`. Then restart `roscore` and continue as follows. These instructions should work with the original TurtleBot with an iRobot Create base.

Log into the TurtleBot's laptop and run:

```
$ roslaunch rbx1_bringup turtlebot_minimal_create.launch
```

Replace the above command with the appropriate launch file for your own robot if you have one.

Next, log into the TurtleBot using another terminal window and run the command:

```
$ roslaunch rbx1_bringup fake_laser.launch
```

(If you have a real laser scanner, you would run its launch file here instead of the fake laser launch file.)

Next, launch the `gmapping_demo.launch` launch file. You can launch this file on your desktop workstation or the robot's laptop:

```
$ roslaunch rbx1_nav gmapping_demo.launch
```

Then bring up `RViz` with the included gmapping configuration file:

```
$ rosrun rviz rviz -d `rospack find rbx1_nav`/gmapping.rviz
```

Next, launch a teleop node for either the keyboard or joystick depending on your hardware:

```
$ roslaunch rbx1_nav keyboard_teleop.launch
```

or

```
$ roslaunch rbx1_nav joystick_teleop.launch
```

The joystick (or the receiver if you have a wireless game controller) can be connected to either the robot's laptop or your desktop. But remember that the `joystick_teleop.launch` file needs to be launched on that same computer.

Test out your connection to the robot by trying to move it with the keyboard or joystick. As the robot moves, the scan points in `RViz` should reflect the changing surroundings near the robot. (Double-check that the **Fixed Frame** is set to the `odom` frame under **Global Options**.)

The final step is to start recording the data to a bag file. You can create the file anywhere you like, but there is a folder called `bag_files` in the `rbx1_nav` package for this purpose if you want to use it:

```
$ roscd rbx1_nav/bag_files
```

Now start the recording process:

```
$ rosbag record -O my_scan_data /scan /tf
```

where `my_scan_data` can be any filename you like. The only data we need to record is the laser scan data and the `tf` transforms. (The `tf` transform tree includes the transformation from the `/odom` frame to the `/base_link` or `/base_footprint` frame which gives us the needed odometry data.)

You are are now ready to drive the robot around the area you'd like to map. Be sure to move the robot slowly, especially when rotating. Stay relatively close to walls and furniture so that there is always something within range of the scanner. Finally, plan to drive a closed loop and continue past the starting point for several meters to ensure a good overlap between the beginning and ending scan data.

8.4.3 Creating the Map

When you are finished driving the robot, type `Ctrl-C` in the `rosbag` terminal window to stop the recording process. Then save the current map as follows:

```
$ roscd rbx1_nav/maps
```

```
$ rosrun map_server map_saver -f my_map
```

where "`my_map`" can be any name you like. This will save the generated map into the current directory under the name you specified on the command line. If you look at the contents of the `rbx1_nav/maps` directory, you will see two new files: `my_map.pgm` which is the map image and `my_map.yaml` that describes the dimensions of the map. It is this latter file that you will point to in subsequent launch files when you want to use the map for navigation.

To view the new map, you can use any image viewer program to bring up the `.pgm` file created above. For example, to use the Ubuntu `eog` viewer ("eye of Gnome") run the command:

```
$ roscd rbx1_nav/maps
$ eog my_map.pgm
```

You can zoom the map using your scroll wheel or the +/- buttons.

Here is a video demonstrating the gmapping process using Pi Robot and a Hokuyo laser scanner: http://youtu.be/7iIDdvCXIFM

8.4.4 Creating a Map from Bag Data

You can also create the map from the bag data you stored during the scanning phase above. This is a useful technique since you can try out different `gmapping` parameters on the same scan data without having to drive the robot around again.

To try it out, first terminate your robot's startup nodes (e.g. `turtlebot_minimal_create.launch`), as well as any laser processes (e.g. `fake_laser.launch`), the `gmapping_demo.launch` file (if it is still running) and any running teleop nodes.

Next, turn on simulated time by setting the `use_sim_time` parameter to `true`:

```
$ rosparam set use_sim_time true
```

Then clear the move_base parameters and re-launch the `gmapping_demo.launch` file again:

```
$ rosparam delete /move_base
$ roslaunch rbx1_nav gmapping_demo.launch
```

You can monitor the process in `RViz` using the gmapping configuration file:

```
$ rosrun rviz rviz -d `rospack find rbx1_nav`/gmapping.rviz
```

Finally, play back your recorded data:

```
$ roscd rbx1_nav/bag_files
$ rosbag play my_scan_data.bag
```

You will probably have to zoom and/or pan the display to keep the entire scan area in view.

When the rosbag file has played all the way through, you save the generated map the same way we did with the live data:

```
$ roscd rbx1_nav/maps
$ rosrun map_server map_saver -f my_map
```

where "`my_map`" can be any name you like. This will save the generated map into the current directory under the name you specified on the command line. If you look at the contents of the `rbx1_nav/maps` directory, you will see two files: `my_map.pgm` which is the map image and `my_map.yaml` that describes the dimensions of the map. It is this latter file that you will point to in subsequent launch files when you want to use the map for navigation.

To view the map created, you can use any image viewer program to bring up the `.pgm` file created above. For example, to use the Ubuntu `eog` viewer ("eye of Gnome") run the command:

```
$ roscd rbx1_nav/maps
$ eog my_map.pgm
```

You can zoom the map using your scroll wheel or the +/- buttons.

NOTE: Don't forget to reset the `use_sim_time` parameter after you are finished map building. Use the command:

```
$ rosparam set use_sim_time false
```

Now that your map is saved we will learn how to use it for localization in the next section.

For additional details about gmapping, take a look at the `gmapping_demo.launch` file in the `rbx1_nav/launch` directory. There you will see many parameters that can be tweaked if needed. This particular launch file is a copied from the `turtlebot_navigation` package and the folks at OSRG have already dialed in the settings that should work for you. To learn more about each parameter, you can check out the gmapping Wiki page.

8.4.5 Can I Extend or Modify an Existing Map?

The ROS `gmapping` package does not provide a way to start with an existing map and then modify it through additional mapping. However, you can always edit a map using your favorite graphics editor. For example, to keep your robot from entering a room, draw a black line across the doorway(s). (**NOTE**: There may be a bug in `move_base` that prevents this trick from working. You can also try setting all the pixels in forbidden region to mid-gray.) To remove a piece of furniture that has been moved, erase the corresponding pixels. In the next section we will find that localization is not terribly sensitive to the exact placement of pixels so it can handle small changes to the locations of objects. But if you have moved a large sofa from one side of a room to the other for example, the new arrangement might confuse your robot. In such a case, you could either move the corresponding pixels in your editor, or take your robot on another scanning run and rebuild the map from scratch using new scan data.

8.5 Navigation and Localization using a Map and `amcl`

If you do not have the hardware to build a map using your own robot, you can use the test map in `rbx1_nav/maps` for this chapter. Otherwise, if you created a map of your own by following the instructions in the previous section you can use it here.

ROS uses the `amcl` package to localize the robot within an existing map using the current scan data coming from the robot's laser or depth camera. But before trying `amcl` on a real robot, let's try fake localization in the ArbotiX simulator.

8.5.1 Testing `amcl` with Fake Localization

Take a look at the launch file called `fake_amcl.launch` in the `rbx1_nav/launch` directory:

```
<launch>

   <param name="use_sim_time" value="false" />
```

```
  <!-- Set the name of the map yaml file: can be overridden on the command line.
-->
  <arg name="map" default="test_map.yaml" />

  <!-- Run the map server with the desired map -->
  <node name="map_server" pkg="map_server" type="map_server" args="$(find
rbx1_nav)/maps/$(arg map)"/>

  <!-- The move_base node -->
  <include file="$(find rbx1_nav)/launch/fake_move_base.launch" />

  <!-- Run fake localization compatible with AMCL output -->
  <node pkg="fake_localization" type="fake_localization"
name="fake_localization" output="screen" />

  <!-- For fake localization we need static transforms between /odom and /map
and /map and /world -->
  <node pkg="tf" type="static_transform_publisher" name="odom_map_broadcaster"
args="0 0 0 0 0 0 /odom /map 100" />
</launch>
```

As you can see from the launch file, first we load a map into the map server, in this case the test map of a single floor house. The map name can be overridden on the command line as we will see below. Alternatively, you can edit the launch file and change the filename to your map so you do not have to type it on the command line.

Next we include the same `fake_move_base.launch` file that we used earlier. Finally, we fire up the `fake_localization` node. As you can read on the `fake_localization` ROS Wiki page, this node simply republishes the robot's odometry data in a form that is compatible with `amcl`.

To test it all out in the ArbotiX simulator, run the following commands—skip any commands that you already have running in various terminals:

```
$ roslaunch rbx1_bringup fake_turtlebot.launch
```

(Replace with your favorite fake robot if desired.)

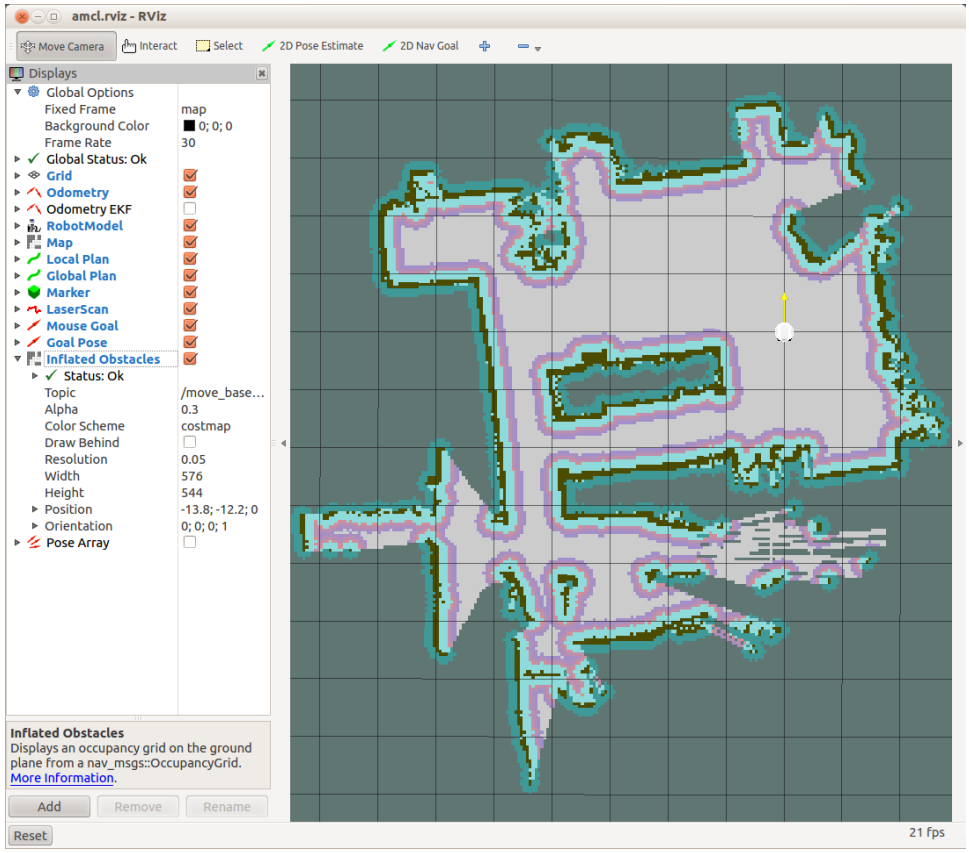
Next run the `fake_amcl.launch` file with the included test map or point it to your own map on the command line:

```
$ roslaunch rbx1_nav fake_amcl.launch map:=test_map.yaml
```

Finally, bring up `RViz` with the amcl configuration file:

```
$ rosrun rviz rviz -d `rospack find rbx1_nav`/amcl.rviz
```

Once the map appears in `RViz`, use the right mouse button or scroll wheel to zoom in or out use the left mouse button to pan (shift-click) or rotate (click). If you are using the test map, the image should look something like the following:



When you have a view you like, click on the **2D Nav Goal** button, then select a goal pose for the robot within the map. As soon as you release the mouse, `move_base` will take over to move the robot to the goal. Try a number of different goals to see how `amcl` and `move_base` work together to plan and implement a path for the robot.

8.5.2 Using `amcl` with a Real Robot

If your robot has a laser scanner or RGB-D camera like the Kinect or Xtion, you can try out `amcl` in the real world. Assuming you have already created a map called `my_map.yaml` in the directory `rbx1_nav/maps`, follow these steps to start localizing your robot.

First, terminate any fake robots you may have running as well as the `fake_amcl.launch` file if you ran it in the previous section. It's also not a bad idea to shut down and restart your `roscore` process just to be sure we are starting with a clean slate.

If you have a TurtleBot, you can run through the official TurtleBot SLAM Tutorial on the ROS Wiki. You can also follow along here and get essentially the same result.

Begin by launching your robot's startup files. For an original TurtleBot, you would run the following on the robot's laptop computer:

```
$ roslaunch rbx1_bringup turtlebot_minimal_create.launch
```

(Or use your own launch file if you have stored your calibration parameters in a different file.)

Next fire up the fake laser scanner. Log into the robot in another terminal and run:

```
$ roslaunch rbx1_bringup fake_laser.launch
```

(If you have a real laser scanner, run its launch file instead.)

Now launch the `tb_demo_amcl.launch` file with your map as an argument:

```
$ roslaunch rbx1_nav tb_demo_amcl.launch map:=my_map.yaml
```

Bring up `RViz` with the included navigation test configuration file:

```
$ rosrun rviz rviz -d `rospack find rbx1_nav`/nav_test.rviz
```

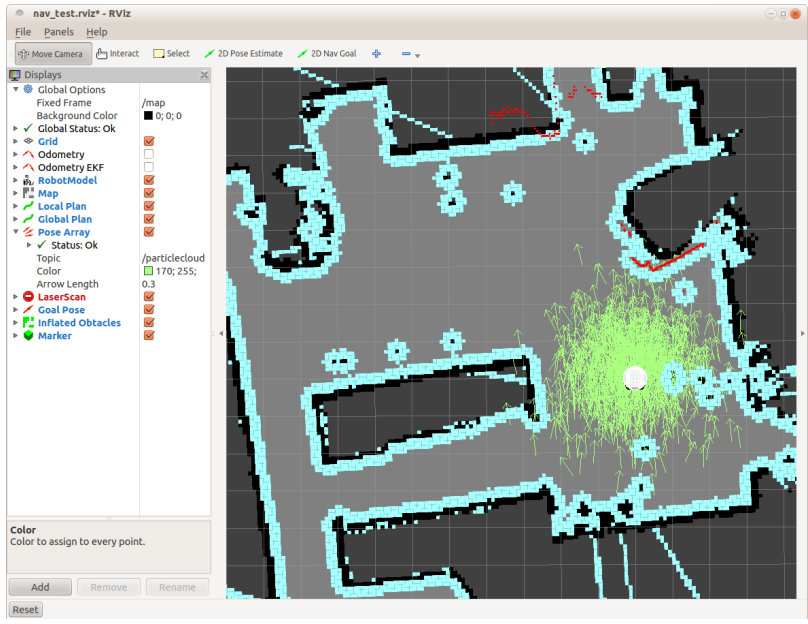
When `amcl` first starts up, you have to give it the initial pose (position and orientation) of the robot as this is something `amcl` cannot figure out on its own. To set the initial pose, first click on the **2D Pose Estimate** button in `RViz`. Then click on the point in the map where you know your robot is located. While holding down the mouse button, a large green arrow should appear. Move the mouse to orient the arrow to match the orientation of your robot, then release the mouse.

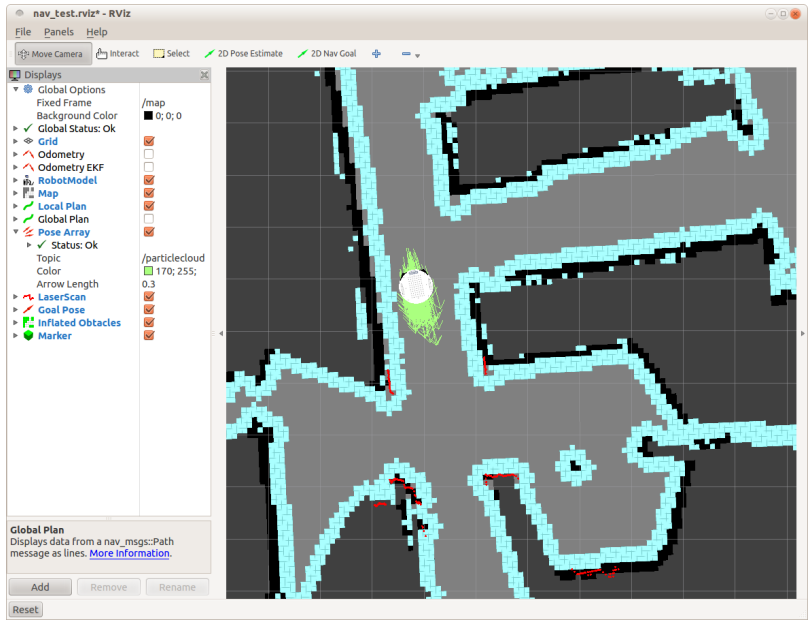
With your robot's initial position set, you should be able to use the **2D Nav Goal** button in `RViz` to point-and-click navigation goals for your robot at different locations in the map. Use your mouse scroll-wheel to zoom out or in as necessary. As the robot moves about, you should see the laser scan points line up with walls and obstacle boundaries.

Probably the first thing you will notice in `RViz` is the cloud of light green arrows surrounding the robot. These arrows represent the range of poses returned by `amcl` and are being displayed because the `nav_test` config file we are using with `RViz` includes a **Pose Array** display type. (Look just below the **GlobalPlan** display in the left-hand panel in `RViz`.) The **Pose Array** topic is set to the `/particlecloud` topic which is the default topic on which `amcl` publishes its pose estimates for the robot. (See the amcl Wiki page for details.) As the robot moves about the environment, this cloud should

shrink in size as additional scan data allows `amcl` to refine its estimate of the robot's position and orientation.

Here are a couple of screen shots from a live test. The first was taken just after the test started and the second was taken after the robot had moved around the the environment for a few minutes :





Notice how the pose array in the first image is quite spread out while in the second image the array has contracted around the robot. At this point in the test, the robot is fairly certain where it is in the map.

To test obstacle avoidance, start the robot toward a goal some distance away, then walk in front of the robot as it is moving. The base local planner should steer the robot around you and then continue on to the target location.

You can also also teleoperate the robot at any time while `amcl` is running if you launch either a keyboard or joystick teleoperation node.

8.5.3 Fully Autonomous Navigation

Using your mouse to send a robot to any location in the house is pretty cool, especially when the robot will dynamically avoid obstacles along the way, but ultimately we want to be able to specify locations programmatically and have the robot move about autonomously. For instance, if we wanted to create a "Patrol Bot" application, we might select a series of locations for the robot to visit, then set the program running and walk away.

In this section we will look at a Python script that does precisely this. It was designed as a kind of navigation endurance test to see how long Pi Robot could autonomously navigate around the house without running into trouble. The file is called `nav_test.py` in the `rbx1_nav/nodes` directory and it executes the following steps:

- Initialize a set of goal locations (poses) within the map.
- Randomly select the next location. (The locations were sampled without replacement so that each location was visited equally often.)
- Send the appropriate `MoveBaseGoal` goal to the `move_base` action server.
- Record the success or failure of the navigation to the goal, as well as the elapsed time and distance traveled.
- Pause for a configurable number of seconds at each location.
- Rinse and repeat.

Before trying it out on a real robot, let's run it in the ArbotiX simulator.

8.5.4 Running the Navigation Test in Simulation

The entire set of nodes we need for the endurance test can be launched using the `fake_nav_test.launch` file. This launch file is actually a good example of a complete ROS application: it launches the robot drivers, the `map_server`, the `move_base` node and the navigation test script.

For this test, the robot will be approaching target locations from different directions depending on the previous location. Rather than force the robot to rotate to a particular orientation at each location, we can set the `yaw_goal_tolerance` parameter to 360 degrees (6.28 radians) which means that once the robot gets to a goal location, the base local planner will be happy with whatever orientation the robot has as it arrives. This results in a more natural motion of the robot from one location to the next. (Of course, for an actual Patrol Bot, you might have reasons for wanting the robot to face a particular direction at specific locations.)

We can set this special `yaw_goal_tolerance` parameter in a separate config file and load the file to override the value we specify in the default configuration file `base_local_planner_params.yaml`. In the `rbx1_nav/config` directory you will find the file `rbx1_nav/config/nav_test_params.yaml` that looks like this:

```
TrajectoryPlannerROS:
  yaw_goal_tolerance: 6.28
```

As you can see, the file is simply a snippet of the `base_local_planner_params.yaml` file consisting of the namespace and the new yaw tolerance. This file is then loaded after the `base_local_planner_params.yaml` file in the `fake_nav_test.launch` file like this:

```
  <!-- The move_base node -->
  <node pkg="move_base" type="move_base" respawn="false" name="move_base"
output="screen">
    <rosparam file="$(find rbx1_nav)/config/fake/costmap_common_params.yaml"
command="load" ns="global_costmap" />
    <rosparam file="$(find rbx1_nav)/config/fake/costmap_common_params.yaml"
command="load" ns="local_costmap" />
    <rosparam file="$(find
rbx1_nav)/config/fake/hydro/local_costmap_params.yaml" command="load" />
    <rosparam file="$(find rbx1_nav)/config/fake/global_costmap_params.yaml"
command="load" />
    <rosparam file="$(find rbx1_nav)/config/fake/base_local_planner_params.yaml"
command="load" />
    <rosparam file="$(find rbx1_nav)/config/nav_test_params.yaml" command="load"
/>
  </node>
```

where you see that the last `rosparam` line loads the `nav_test_params.yaml` file.

With these preliminaries out of the way, let's run the test. To make sure we are starting with a clean slate, kill any running launch files as well as `roscore`. Begin with a new `roscore`:

```
$ roscore
```

Next, fire up `rqt_console`. This allows us to view the status messages being output by the `nav_test.py` script. The same messages are shown in the terminal window

used to run the the `fake_nav_test.launch` file, but `rqt_console` provides a nicer monitoring interface.

```
$ rqt_console &
```

Now run the `fake_nav_test.launch` file. This launch file brings up a fake TurtleBot, the `map_server` node with the test map loaded, the `move_base` node with all the parameters tuned for the fake TurtleBot, the `fake_localization` node, and finally, the `nav_test.py` script itself.

```
$ roslaunch rbx1_nav fake_nav_test.launch
```

The very last message in the output from the launch file should be:

```
*** Click the 2D Pose Estimate button in RViz to set the robot's
initial pose...
```

At this point, fire up `RViz` with the `amcl` configuration file:

```
$ rosrun rviz rviz -d `rospack find rbx1_nav`/amcl.rviz
```

When `RViz` is up and the robot is visible, click on the **2D Pose Estimate** button to set the initial pose of the robot. Then click on the middle of the robot and align the green pose arrow with the yellow odometry arrow. As soon as you release the mouse, the navigation test will start. Use the mouse to zoom or pan the map during the test.

The fake TurtleBot should now start moving in `RViz` from one location to another. For the fake test, the robot does not pause at each location. The simulation will run forever until you type `Ctrl-C` in the `fake_nav_test.launch` window.

As the test proceeds, bring the `rqt_console` window to the foreground and you can monitor which goal location is next, the time elapsed, distance traveled, the number of goal locations attempted and the success rate. You will probably have to widen the **Message** column with your mouse to see each message in full.

8.5.5 Understanding the Navigation Test Script

Before running the navigation test on a real robot, let's make sure we understand the code that makes it work.

Link to source: nav_test.py

```
1.  #!/usr/bin/env python
2.
3.  import rospy
4.  import actionlib
```

```
5.  from actionlib_msgs.msg import *
6.  from geometry_msgs.msg import Pose, PoseWithCovarianceStamped, Point,
    Quaternion, Twist
7.  from move_base_msgs.msg import MoveBaseAction, MoveBaseGoal
8.  from random import sample
9.  from math import pow, sqrt
10.
11. class NavTest():
12.     def __init__(self):
13.         rospy.init_node('nav_test', anonymous=True)
14.
15.         rospy.on_shutdown(self.shutdown)
16.
17.         # How long in seconds should the robot pause at each location?
18.         self.rest_time = rospy.get_param("~rest_time", 10)
19.
20.         # Are we running in the fake simulator?
21.         self.fake_test = rospy.get_param("~fake_test", False)
22.
23.         # Goal state return values
24.         goal_states = ['PENDING', 'ACTIVE', 'PREEMPTED',
25.                        'SUCCEEDED', 'ABORTED', 'REJECTED',
26.                        'PREEMPTING', 'RECALLING', 'RECALLED',
27.                        'LOST']
28.
29.         # Set up the goal locations. Poses are defined in the map frame.
30.         # An easy way to find the pose coordinates is to point-and-click
31.         # Nav Goals in RViz when running in the simulator.
32.         # Pose coordinates are then displayed in the terminal
33.         # that was used to launch RViz.
34.         locations = dict()
35.
36.         locations['hall_foyer'] = Pose(Point(0.643, 4.720, 0.000),
37.                                        Quaternion(0.000, 0.000, 0.223, 0.975))
38.         locations['hall_kitchen'] = Pose(Point(-1.994, 4.382, 0.000),
39.                                          Quaternion(0.000, 0.000, -0.670, 0.743))
40.         locations['hall_bedroom'] = Pose(Point(-3.719, 4.401, 0.000),
41.                                          Quaternion(0.000, 0.000, 0.733, 0.680))
42.         locations['living_room_1'] = Pose(Point(0.720, 2.229, 0.000),
43.                                           Quaternion(0.000, 0.000, 0.786, 0.618))
44.         locations['living_room_2'] = Pose(Point(1.471, 1.007, 0.000),
45.                                           Quaternion(0.000, 0.000, 0.480, 0.877))
46.         locations['dining_room_1'] = Pose(Point(-0.861, -0.019, 0.000),
47.                                           Quaternion(0.000, 0.000, 0.892, -0.451))
48.
49.         # Publisher to manually control the robot (e.g. to stop it)
50.         self.cmd_vel_pub = rospy.Publisher('cmd_vel', Twist)
51.
52.         # Subscribe to the move_base action server
53.         self.move_base = actionlib.SimpleActionClient("move_base",
    MoveBaseAction)
54.
55.         rospy.loginfo("Waiting for move_base action server...")
56.
57.         # Wait 60 seconds for the action server to become available
58.         self.move_base.wait_for_server(rospy.Duration(60))
59.
```

```
60.          rospy.loginfo("Connected to move base server")
61.
62.          # A variable to hold the initial pose of the robot to be set by
63.          # the user in RViz
64.          initial_pose = PoseWithCovarianceStamped()
65.
66.          # Variables to keep track of success rate, running time,
67.          # and distance traveled
68.          n_locations = len(locations)
69.          n_goals = 0
70.          n_successes = 0
71.          i = n_locations
72.          distance_traveled = 0
73.          start_time = rospy.Time.now()
74.          running_time = 0
75.          location = ""
76.          last_location = ""
77.
78.          # Get the initial pose from the user
79.          rospy.loginfo("Click on the map in RViz to set the intial
    pose...")
80.          rospy.wait_for_message('initialpose', PoseWithCovarianceStamped)
81.          self.last_location = Pose()
82.          rospy.Subscriber('initialpose', PoseWithCovarianceStamped,
    self.update_initial_pose)
83.
84.          # Make sure we have the initial pose
85.          while initial_pose.header.stamp == "":
86.              rospy.sleep(1)
87.
88.          rospy.loginfo("Starting navigation test")
89.
90.          # Begin the main loop and run through a sequence of locations
91.          while not rospy.is_shutdown():
92.              # If we've gone through the current sequence,
93.              # start with a new random sequence
94.              if i == n_locations:
95.                  i = 0
96.                  sequence = sample(locations, n_locations)
97.                  # Skip over first location if it is the same as
98.                  # the last location
99.           if sequence[0] == last_location:
100.                 i = 1
101.
102.         # Get the next location in the current sequence
103.         location = sequence[i]
104.
105.         # Keep track of the distance traveled.
106.         # Use updated initial pose if available.
107.         if initial_pose.header.stamp == "":
108.             distance = sqrt(pow(locations[location].position.x
109.                         - locations[last_location].position.x, 2) +
110.                           pow(locations[location].position.y -
111.                           locations[last_location].position.y, 2))
112.         else:
113.             rospy.loginfo("Updating current pose.")
114.             distance = sqrt(pow(locations[location].position.x
```

```
                        - initial_pose.pose.pose.position.x, 2) +
                          pow(locations[location].position.y -
                          initial_pose.pose.pose.position.y, 2))
            initial_pose.header.stamp = ""

        # Store the last location for distance calculations
        last_location = location

        # Increment the counters
        i += 1
        n_goals += 1

        # Set up the next goal location
        self.goal = MoveBaseGoal()
        self.goal.target_pose.pose = locations[location]
        self.goal.target_pose.header.frame_id = 'map'
        self.goal.target_pose.header.stamp = rospy.Time.now()

        # Let the user know where the robot is going next
        rospy.loginfo("Going to: " + str(location))

        # Start the robot toward the next location
        self.move_base.send_goal(self.goal)

        # Allow 5 minutes to get there
        finished_within_time =
self.move_base.wait_for_result(rospy.Duration(300))

        # Check for success or failure
        if not finished_within_time:
            self.move_base.cancel_goal()
            rospy.loginfo("Timed out achieving goal")
        else:
            state = self.move_base.get_state()
            if state == GoalStatus.SUCCEEDED:
                rospy.loginfo("Goal succeeded!")
                n_successes += 1
                distance_traveled += distance
            else:
              rospy.loginfo("Goal failed with error code: " +
str(goal_states[state]))

        # How long have we been running?
        running_time = rospy.Time.now() - start_time
        running_time = running_time.secs / 60.0

        # Print a summary success/failure, distance traveled and time
elapsed
        rospy.loginfo("Success so far: " + str(n_successes) + "/" +
                      str(n_goals) + " = " +
                      str(100 * n_successes/n_goals) + "%")
        rospy.loginfo("Running time: " + str(trunc(running_time, 1)) +
                      " min Distance: " + str(trunc(distance_traveled,
1)) + " m")
        rospy.sleep(self.rest_time)

def update_initial_pose(self, initial_pose):
```

```
168.     self.initial_pose = initial_pose
169.
170. def shutdown(self):
171.     rospy.loginfo("Stopping the robot...")
172.     self.move_base.cancel_goal()
173.     rospy.sleep(2)
174.     self.cmd_vel_pub.publish(Twist())
175.     rospy.sleep(1)
176.
177. def trunc(f, n):
178.     # Truncates/pads a float f to n decimal places without rounding
179.     slen = len('%.*f' % (n, f))
180.     return float(str(f)[:slen])
181.
182. if __name__ == '__main__':
183.     try:
184.         NavTest()
185.         rospy.spin()
186.     except rospy.ROSInterruptException:
187.         rospy.loginfo("AMCL navigation test finished.")
```

Here are the key lines of the script and what they do.

```
17.         # How long in seconds should the robot pause at each location?
18.         self.rest_time = rospy.get_param("~rest_time", 10)
19.
20.         # Are we running in the fake simulator?
21.         self.fake_test = rospy.get_param("~fake_test", False)
```

Set the number of seconds to pause at each location before moving to the next. If the `fake_test` parameter is `True`, the `rest_time` parameter is ignored.

```
24.         goal_states = ['PENDING', 'ACTIVE', 'PREEMPTED',
25.                        'SUCCEEDED', 'ABORTED', 'REJECTED',
26.                        'PREEMPTING', 'RECALLING', 'RECALLED',
27.                        'LOST']
```

It's nice to have a human readable form of the various `MoveBaseAction` goal states.

```
34.         locations = dict()
35.
36.         locations['hall_foyer'] = Pose(Point(0.643, 4.720, 0.000),
37.                                        Quaternion(0.000, 0.000, 0.223, 0.975))
38.         locations['hall_kitchen'] = Pose(Point(-1.994, 4.382, 0.000),
39.                                        Quaternion(0.000, 0.000, -0.670, 0.743))
40.         locations['hall_bedroom'] = Pose(Point(-3.719, 4.401, 0.000),
41.                                        Quaternion(0.000, 0.000, 0.733, 0.680))
42.         locations['living_room_1'] = Pose(Point(0.720, 2.229, 0.000),
43.                                        Quaternion(0.000, 0.000, 0.786, 0.618))
44.         locations['living_room_2'] = Pose(Point(1.471, 1.007, 0.000),
45.                                        Quaternion(0.000, 0.000, 0.480, 0.877))
46.         locations['dining_room_1'] = Pose(Point(-0.861, -0.019, 0.000),
47.                                        Quaternion(0.000, 0.000, 0.892, -0.451))
```

The goal locations (poses) are stored as a Python dictionary. The `Point` and `Quaternion` values can be found by clicking on the **2D Nav Goal** button in `RViz` and noting the values displayed in the terminal that was used to launch `RViz`. Then copy and paste those values into your script.

```
83.         rospy.Subscriber('initialpose', PoseWithCovarianceStamped,
    self.update_initial_pose)
```

Here we subscribe to the `initialpose` topic so that we can set the initial location and orientation of the robot within the map at the start of the test. When running the test on a real robot, the initial pose is set by the user in `RViz`. For a fake robot (`fake_test = True`), this variable is ignored since fake localization is already perfect.

```
91.         while not rospy.is_shutdown():
```

Run the test until the user aborts the application.

```
96.             sequence = sample(locations, n_locations)
```

We use the Python `sample()` function to generate a random sequence of goals from the set of locations.

```
127.            # Set up the next goal location
128.            self.goal = MoveBaseGoal()
129.            self.goal.target_pose.pose = locations[location]
130.            self.goal.target_pose.header.frame_id = 'map'
131.            self.goal.target_pose.header.stamp = rospy.Time.now()
132.
133.            # Let the user know where the robot is going next
134.            rospy.loginfo("Going to: " + str(location))
135.
136.            # Start the robot toward the next location
137.            self.move_base.send_goal(self.goal)
```

Finally, we set the goal location to the next location in the sequence and send it to the `move_base` action server.

8.5.6 Running the Navigation Test on a Real Robot

The procedure for running the navigation test with a real robot is nearly the same as with the simulation. However, you will of course need your own map of the area where you want the robot to run.

For this test, it is probably a good idea to run all nodes on the robot's computer except `rqt_console` and `RViz`. This will ensure that there are no timing issues between the robot and your desktop due to fluctuations in your wifi network.

The rbx1_nav directory includes a launch file called tb_nav_test.launch that should work with a TurtleBot but you will need to edit it first and point to your own map file. You can also supply the name of your map file on the command line as we did earlier and as we will illustrate below.

First terminate any simulations you might have running. If you really want to be on the safe side, shut down and restart roscore to make sure the entire parameter server is also cleared.

Next, fire up your robot's startup files. In the case of the original TurtleBot, you would run the following launch file on the TurtleBot's laptop:

```
$ roslaunch rbx1_bringup turtlebot_minimal_create.launch
```

(Use your own launch file if you have saved your calibration parameters in a different file.)

If you don't have a real laser scanner, launch the fake laser:

```
$ roslaunch rbx1_bringup fake_laser.launch
```

If it is not already running, bring up up rqt_console to monitor status messages during the test. This can be run on your desktop:

```
$ rqt_console &
```

Next, launch the tb_nav_test.launch file. Run the launch file on the robot's computer:

```
$ roslaunch rbx1_nav tb_nav_test.launch map:=my_map.yaml
```

where you need to replace my_map.yaml with the actual name of your map file. You can leave off the map argument if you have edited your tb_nav_test.launch file to include your map file there.

Finally, if you don't already have RViz running with the nav_test config file, bring it up now on your desktop computer:

```
$ rosrun rviz rviz -d `rospack find rbx1_nav`/nav_test.rviz
```

If everything goes well, you should see the following message in rqt_console and in the terminal window used to launch the tb_nav_test.launch file:

```
*** Click the 2D Pose Estimate button in RViz to set the robot's
initial pose...
```

Before the navigation test will start, you need to set the initial pose of the robot. Click on the approximate starting location of the robot in `RViz`, and when the large green arrow appears, point the arrow to match the orientation of the robot. As soon as you release the mouse button, the robot should begin the test.

NOTE: To stop the robot (and the test) at any time, simply type `Ctrl-C` in the terminal window used to start the `tb_nav_test.launch` file. If you are using a TurtleBot, you can also press the front bumper with your foot or hand and the robot will stop moving.

8.5.7 What's Next?

This concludes the chapter on Navigation. You should now have the tools you need to program your own robot to autonomously navigation your home or office. To get started, you could try a few variations on the navigation test script. For example, instead of using a set of fixed target locations, try picking locations at random, or simply add some random variation around the primary locations. You could also have the robot rotate in place at each location as a Patrol Bot might do to scan its surroundings.

9. Speech Recognition and Synthesis

Speech recognition and Linux have come a long way in the past few years, thanks mostly to the CMU Sphinx and Festival projects. We can also benefit from existing ROS packages for both speech recognition and text-to-speech. Consequently, it is quite easy to add speech control and voice feedback to your robot as we will now show.

In this chapter we will:

- Install and test the `pocketsphinx` package for speech recognition
- Learn how to create a custom vocabulary for speech recognition
- Teleoperate a real and simulated robot using voice commands
- Install and test the Festival text-to-speech system and the ROS `sound_play` package

9.1 Installing PocketSphinx for Speech Recognition

Thanks to Michael Ferguson from the University of Albany (and now CTO at Unbounded Robotics), we can use the ROS `pocketsphinx` package for speech recognition. The `pocketsphinx` package requires the installation of the Ubuntu package `gstreamer0.10-pocketsphinx` and we will also need the ROS sound drivers stack (in case you don't already have it) so so let's take care of both first. You will be prompted to install the Festival packages if you don't already have them—answer "Y" if prompted:

```
$ sudo apt-get install gstreamer0.10-pocketsphinx
$ sudo apt-get install ros-hydro-pocketsphinx
$ sudo apt-get install ros-hydro-audio-common
$ sudo apt-get install libasound2
```

The `pocketsphinx` package includes the node `recognizer.py`. This script does all the hard work of connecting to the audio input stream of your computer and matching voice commands to the words or phrases in the current vocabulary. When the recognizer node matches a word or phrase, it publishes it on the `/recognizer/output` topic. Other nodes can subscribe to this topic to find out what the user has just said.

9.2 Testing the PocketSphinx Recognizer

You will get the best speech recognition results using a headset microphone, either USB, standard audio or Bluetooth. Once you have your microphone connected to your

computer, make sure it is selected as the input audio device. (If you are using Ubuntu 12.04 or higher, go to **System Settings** and click on the **Sound** control panel.) Once the Sound Preferences window opens, click on the **Input** tab and select your microphone device from the list (if there is more than one). Speak a few words into your microphone and you should see the volume meter respond. Then click on the **Output** tab, select your desired output device, and adjust the volume slider. Finally, close the **Sound** window.

NOTE: If you disconnect a USB or Bluetooth microphone and then reconnect it later, you will likely have to select it as the input again using the procedure described above.

Michael Ferguson includes a vocabulary file suitable for RoboCup@Home competitions that you can use to the test the recognizer. Fire it up now by running:

```
$ roslaunch pocketsphinx robocup.launch
```

You should see a list of INFO messages indicating that the various parts of the recognition model are being loaded. The last few messages will look something like this:

```
INFO: ngram_search_fwdtree.c(195): Creating search tree
INFO: ngram_search_fwdtree.c(203): 0 root, 0 non-root channels,
26 single-phone words
INFO: ngram_search_fwdtree.c(325): max nonroot chan increased to
328
INFO: ngram_search_fwdtree.c(334): 77 root, 200 non-root
channels, 6 single-phone words
```

Now say a few of the RoboCup phrases such as "bring me the glass", "go to the kitchen", or "come with me". The output should look something like this:

```
[INFO] [WallTime: 1387548761.587537] bring me the glass
[INFO] [WallTime: 1387548765.296757] go to the kitchen
[INFO] [WallTime: 1387548769.417876] come with me
```

Congratulations—you can now talk to your robot! The recognized phrase is also published on the topic `/recognizer/output`. To see the result, bringup another terminal and run:

```
$ rostopic echo /recognizer/output
```

Now try the same three phrases as above and you should see:

```
data: bring me the glass
---
data: go to the kitchen
---
data: come with me
---
```

For my voice, and using a Bluetooth over-the-ear microphone, the recognizer was surprisingly fast and accurate.

To see all the phrases you can use with the demo RoboCup vocabulary, run the following commands:

```
$ roscd pocketsphinx/demo
$ more robocup.corpus
```

Now try saying a phrase that is not in the vocabulary, such as "the sky is blue". In my case, the result on the `/recognizer/output` topic was "this go is room". As you can see, the recognizer will respond with something no matter what you say. This means that care must be taken to "mute" the speech recognizer if you don't want random conversation to be interpreted as speech commands. We will see how to do this below when we learn how to map speech recognition into actions.

9.3 Creating a Vocabulary

It is easy to create a new vocabulary or ***corpus*** as it is referred to in PocketSphinx. First, create a simple text file with one word or phrase per line. Here is a corpus that could be used to drive your robot around using voice commands. We will store it in a file called `nav_commands.txt` in the `config` subdirectory of the `rbx1_speech` package. To view its contents, run the commands:

```
$ roscd rbx1_speech/config
$ more nav_commands.txt
```

You should see the following list of phrases (they will appear all in one column on your screen):

pause speech	come forward	stop
continue speech	come backward	stop now
move forward	come left	halt
move backward	come right	abort
move back	turn left	kill
move left	turn right	panic
move right	rotate left	help
go forward	rotate right	help me
go backward	faster	freeze
go back	speed up	turn off
go left	slower	shut down
go right	slow down	cancel
go straight	quarter speed	
	half speed	
	full speed	

You can also bring the file up in your favorite editor and add, delete or change some of these words or phrases before proceeding to the next step. When you enter your phrases, try not to mix upper and lower case and do not use punctuation marks. Also, if want to include a number such as 54, spell it out as "fifty four".

Before we can use this corpus with PocketSphinx, we need to compile it into special dictionary and pronunciation files. This can be done using the online CMU language model (**lm**) tool located at:

http://www.speech.cs.cmu.edu/tools/lmtool-new.html

Follow the directions to upload your `nav_commands.txt` file, click the **Compile Knowledge Base** button, then download the file labeled **COMPRESSED TARBALL** that contains all the language model files. Extract these files into the `config` subdirectory of the `rbx1_speech` package. The files will all begin with the same number, such as `3026.dic`, `3026.lm`, etc. These files define your vocabulary as a language model that PocketSphinx can understand. You can rename all these files to something more memorable using a command like the following (the 4-digit number will likely be different in your case):

```
$ roscd rbx1_speech/config
$ rename -f 's/3026/nav_commands/' *
```

Next, take a look at the `voice_nav_commands.launch` file found in the `rbx1_speech/launch` subdirectory. It looks like this:

```
<launch>
  <node name="recognizer" pkg="pocketsphinx" type="recognizer.py"
output="screen">
    <param name="lm" value="$(find rbx1_speech)/config/nav_commands.lm"/>
    <param name="dict" value="$(find rbx1_speech)/config/nav_commands.dic"/>
  </node>
</launch>
```

As you can see, the launch file runs the `recognizer.py` node from the `pocketsphinx` package and we point the `lm` and `dict` parameters to the files `nav_commands.lm` and `nav_commands.dic` created in the steps above. Note also that the parameter `output="screen"` is what allows us to see the real-time recognition results in the launch window.

Launch this file and test speech recognition by monitoring the `/recognizer/output` topic. First type `Ctrl-C` to terminate the earlier RoboCup demo if it is still running. Then run the commands:

```
$ roslaunch rbx1_speech voice_nav_commands.launch
```

And in another terminal:

```
$ rostopic echo /recognizer/output
```

Try saying a few navigation phrases such as "move forward", "slow down" and "stop". You should see your commands echoed on the `/recognizer/output` topic.

9.4 A Voice-Control Navigation Script

The `recognizer.py` node in the `pocketsphinx` package publishes recognized speech commands to the `/recognizer/output` topic. To map these commands to robot actions, we need a second node that subscribes to this topic, looks for appropriate messages, then causes the robot to execute different behaviors depending on the message received. To get us started, Michael Ferguson includes a Python script called `voice_cmd_vel.py` in the `pocketsphinx` package that maps voice commands into `Twist` messages that can be used to control a mobile robot. We will use a slightly modified version of this script called `voice_nav.py` found in the `rbx1_speech/nodes` subdirectory.

Let's now look at the `voice_nav.py` script.

Link to source: voice_nav.py

```
1  #!/usr/bin/env python
2
3  """
```

```
4     Based on the voice_cmd_vel.py script by Michael Ferguson in
5     the pocketsphinx ROS package.
6
7     See http://wiki.ros.org/pocketsphinx
8   """
9
10
11  import rospy
12  from geometry_msgs.msg import Twist
13  from std_msgs.msg import String
14  from math import copysign
15
16  class VoiceNav:
17      def __init__(self):
18          rospy.init_node('voice_nav')
19
20          rospy.on_shutdown(self.cleanup)
21
22          # Set a number of parameters affecting the robot's speed
23          self.max_speed = rospy.get_param("~max_speed", 0.4)
24          self.max_angular_speed = rospy.get_param("~max_angular_speed", 1.5)
25          self.speed = rospy.get_param("~start_speed", 0.1)
26          self.angular_speed = rospy.get_param("~start_angular_speed", 0.5)
27          self.linear_increment = rospy.get_param("~linear_increment", 0.05)
28          self.angular_increment = rospy.get_param("~angular_increment", 0.4)
29
30          # We don't have to run the script very fast
31          self.rate = rospy.get_param("~rate", 5)
32          r = rospy.Rate(self.rate)
33
34          # A flag to determine whether or not voice control is paused
35          self.paused = False
36
37          # Initialize the Twist message we will publish.
38          self.cmd_vel = Twist()
39
40          # Publish the Twist message to the cmd_vel topic
41          self.cmd_vel_pub = rospy.Publisher('cmd_vel', Twist)
42
43          # Subscribe to the /recognizer/output topic to receive voice commands.
44          rospy.Subscriber('/recognizer/output', String, self.speech_callback)
45
46          # A mapping from keywords or phrases to commands
47          self.keywords_to_command = {'stop': ['stop', 'halt', 'abort', 'kill',
'panic', 'off', 'freeze', 'shut down', 'turn off', 'help', 'help me'],
48                                      'slower': ['slow down', 'slower'],
49                                      'faster': ['speed up', 'faster'],
50                                      'forward': ['forward', 'ahead',
'straight'],
51                                      'backward': ['back', 'backward', 'back
up'],
52                                      'rotate left': ['rotate left'],
53                                      'rotate right': ['rotate right'],
54                                      'turn left': ['turn left'],
55                                      'turn right': ['turn right'],
56                                      'quarter': ['quarter speed'],
57                                      'half': ['half speed'],
```

```
                                    'full': ['full speed'],
                                    'pause': ['pause speech'],
                                    'continue': ['continue speech']}

        rospy.loginfo("Ready to receive voice commands")

        # We have to keep publishing the cmd_vel message if we want
        # the robot to keep moving.
        while not rospy.is_shutdown():
            self.cmd_vel_pub.publish(self.cmd_vel)
            r.sleep()

    def get_command(self, data):
        # Attempt to match the recognized word or phrase to the
        # keywords_to_command dictionary and return the appropriate
        # command
        for (command, keywords) in self.keywords_to_command.iteritems():
            for word in keywords:
                if data.find(word) > -1:
                    return command

    def speech_callback(self, msg):
        # Get the motion command from the recognized phrase
        command = self.get_command(msg.data)

        # Log the command to the screen
        rospy.loginfo("Command: " + str(command))

        # If the user has asked to pause/continue voice control,
        # set the flag accordingly
        if command == 'pause':
            self.paused = True
        elif command == 'continue':
            self.paused = False

        # If voice control is paused, simply return without
        # performing any action
        if self.paused:
            return

        # The list of if-then statements should be fairly
        # self-explanatory
        if command == 'forward':
            self.cmd_vel.linear.x = self.speed
            self.cmd_vel.angular.z = 0

        elif command == 'rotate left':
            self.cmd_vel.linear.x = 0
            self.cmd_vel.angular.z = self.angular_speed

        elif command == 'rotate right':
            self.cmd_vel.linear.x = 0
            self.cmd_vel.angular.z = -self.angular_speed

        elif command == 'turn left':
            if self.cmd_vel.linear.x != 0:
                self.cmd_vel.angular.z += self.angular_increment
```

```
114             else:
115                 self.cmd_vel.angular.z = self.angular_speed
116 
117         elif command == 'turn right':
118             if self.cmd_vel.linear.x != 0:
119                 self.cmd_vel.angular.z -= self.angular_increment
120             else:
121                 self.cmd_vel.angular.z = -self.angular_speed
122 
123         elif command == 'backward':
124             self.cmd_vel.linear.x = -self.speed
125             self.cmd_vel.angular.z = 0
126 
127         elif command == 'stop':
128             # Stop the robot!  Publish a Twist message consisting of all
zeros.
129             self.cmd_vel = Twist()
130 
131         elif command == 'faster':
132             self.speed += self.linear_increment
133             self.angular_speed += self.angular_increment
134             if self.cmd_vel.linear.x != 0:
135                 self.cmd_vel.linear.x += copysign(self.linear_increment,
self.cmd_vel.linear.x)
136             if self.cmd_vel.angular.z != 0:
137                 self.cmd_vel.angular.z += copysign(self.angular_increment,
self.cmd_vel.angular.z)
138 
139         elif command == 'slower':
140             self.speed -= self.linear_increment
141             self.angular_speed -= self.angular_increment
142             if self.cmd_vel.linear.x != 0:
143                 self.cmd_vel.linear.x -= copysign(self.linear_increment,
self.cmd_vel.linear.x)
144             if self.cmd_vel.angular.z != 0:
145                 self.cmd_vel.angular.z -= copysign(self.angular_increment,
self.cmd_vel.angular.z)
146 
147         elif command in ['quarter', 'half', 'full']:
148             if command == 'quarter':
149                 self.speed = copysign(self.max_speed / 4, self.speed)
150 
151             elif command == 'half':
152                 self.speed = copysign(self.max_speed / 2, self.speed)
153 
154             elif command == 'full':
155                 self.speed = copysign(self.max_speed, self.speed)
156 
157             if self.cmd_vel.linear.x != 0:
158                 self.cmd_vel.linear.x = copysign(self.speed,
self.cmd_vel.linear.x)
159 
160             if self.cmd_vel.angular.z != 0:
161                 self.cmd_vel.angular.z = copysign(self.angular_speed,
self.cmd_vel.angular.z)
162 
163         else:
```

```
164             return
165
166         self.cmd_vel.linear.x = min(self.max_speed, max(-self.max_speed,
self.cmd_vel.linear.x))
167         self.cmd_vel.angular.z = min(self.max_angular_speed, max(-
self.max_angular_speed, self.cmd_vel.angular.z))
168
169     def cleanup(self):
170         # When shutting down be sure to stop the robot!
171         twist = Twist()
172         self.cmd_vel_pub.publish(twist)
173         rospy.sleep(1)
174
175 if __name__=="__main__":
176     try:
177         VoiceNav()
178         rospy.spin()
179     except rospy.ROSInterruptException:
180         rospy.loginfo("Voice navigation terminated.")
```

The script is fairly straightforward and heavily commented so we will only describe the highlights.

```
46         # A mapping from keywords or phrases to commands
47         self.keywords_to_command = {'stop': ['stop', 'halt', 'abort', 'kill',
'panic', 'off', 'freeze', 'shut down', 'turn off', 'help', 'help me'],
48                                     'slower': ['slow down', 'slower'],
49                                     'faster': ['speed up', 'faster'],
50                                     'forward': ['forward', 'ahead',
'straight'],
51                                     'backward': ['back', 'backward', 'back
up'],
52                                     'rotate left': ['rotate left'],
53                                     'rotate right': ['rotate right'],
54                                     'turn left': ['turn left'],
55                                     'turn right': ['turn right'],
56                                     'quarter': ['quarter speed'],
57                                     'half': ['half speed'],
58                                     'full': ['full speed'],
59                                     'pause': ['pause speech'],
60                                     'continue': ['continue speech']}
61
```

The `keywords_to_command` Python dictionary allows us to map different verbal works and phrases into the same action. For example, it is really important to be able to stop the robot once it is moving. However, the word "stop" is not always recognized by the PocketSphinx recognizer. So we provide a number of alternative ways of telling the robot to stop like "halt", "abort", "help", etc. Of course, these alternatives must be included in our original PocketSphinx vocabulary (corpus).

The `voice_nav.py` node subscribes to the `/recognizer/output` topic and looks for recognized keywords as specified in the `nav_commands.txt` corpus. If a match is

found, the `keywords_to_commands` dictionary maps the matched phrase to an appropriate command word. Our callback function then maps the command word to the appropriate `Twist` action sent to the robot.

Another feature of the `voice_nav.py` script is that it will respond to the two special commands "pause speech" and "continue speech". If you are voice controlling your robot, but you would like to say something to another person without the robot interpreting your words as movement commands, just say "pause speech". When you want to continue controlling the robot, say "continue speech".

9.4.1 Testing Voice-Control in the ArbotiX Simulator

Before using voice navigation with a real robot, let's give it a try in the ArbotiX simulator. First fire up the fake TurtleBot as we have done before:

```
$ roslaunch rbx1_bringup fake_turtlebot.launch
```

Next, bring up `RViz` with the simulation config file:

```
$ rosrun rviz rviz -d `rospack find rbx1_nav`/sim.rviz
```

Let's also use `rqt_console` to more easily monitor the output of the voice navigation script. In particular, this will allow us to view the commands the script recognizes:

```
$ rqt_console &
```

Before running the voice navigation script, check your **Sound Settings** as described earlier to make sure your microphone is still set as the **Input** device. Now run the `voice_nav_commands.launch` and `turtlebot_voice_nav.launch` files:

```
$ roslaunch rbx1_speech voice_nav_commands.launch
```

and in another terminal:

```
$ roslaunch rbx1_speech turtlebot_voice_nav.launch
```

You should now be able to use voice commands to move the fake TurtleBot around in `RViz`. For example, try the commands "rotate left", "move forward", "full speed", "halt", and so on. Here is the list of voice commands again for easy reference:

pause speech	come forward	stop
continue speech	come backward	stop now
move forward	come left	halt
move backward	come right	abort
move back	turn left	kill
move left	turn right	panic
move right	rotate left	help
go forward	rotate right	help me
go backward	faster	freeze
go back	speed up	turn off
go left	slower	shut down
go right	slow down	cancel
go straight	quarter speed	
	half speed	
	full speed	

You can also try the two special commands, "pause speech" and "continue speech" to see if you can turn voice control off and on again.

9.4.2 Using Voice-Control with a Real Robot

To voice control a TurtleBot, move the robot into an open space free of obstacles, then start the `turtlebot.launch` file on the TurtleBot's laptop:

```
$ roslaunch rbx1_bringup turtlebot_minimal_create.launch
```

If it's not already running, bring up `rqt_console` to more easily monitor the output of the voice navigation script:

```
$ rqt_console &
```

Before running the voice navigation script, check your **Sound Settings** as described earlier to make sure your microphone is still set as the **Input** device.

On your workstation computer, run the `voice_nav_commands.launch` file:

```
$ roslaunch rbx1_speech voice_nav_commands.launch
```

and in another terminal run the `turtlebot_voice_nav.launch` file:

```
$ roslaunch rbx1_speech turtlebot_voice_nav.launch
```

Try a relatively safe voice command first such as "rotate right". Refer to the list of commands above for different ways you can move the robot. The `turtlebot_voice_nav.launch` file includes parameters you can set that determine

the maximum speed of the TurtleBot as well as the increments used when you say "go faster" or "slow down".

The following video presents a short demonstration of the script running on a modified TurtleBot: http://www.youtube.com/watch?v=10ysYZUX_jA

9.5 Installing and Testing Festival Text-to-Speech

Now that we can talk to our robot, it would be nice if it could talk back to us. Text-to-speech (TTS) is accomplished using the CMU Festival system together with the ROS sound_play package. If you have followed this chapter from the beginning, you have already done the following step. Otherwise, run it now. You will be prompted to install the Festival packages if you don't already have them—answer "Y" of course:

```
$ sudo apt-get install ros-hydro-audio-common
$ sudo apt-get install libasound2
```

The `sound_play` package uses the CMU Festival TTS library to generate synthetic speech. Let's test it out with the default voice as follows. First fire up the primary `sound_play` node:

```
$ rosrun sound_play soundplay_node.py
```

In another terminal, enter some text to be converted to voice:

```
$ rosrun sound_play say.py "Greetings Humans. Take me to your leader."
```

The default voice is called `kal_diphone`. To see all the English voices currently installed on your system:

```
$ ls /usr/share/festival/voices/english
```

To get a list of all basic Festival voices available, run the following command:

```
$ sudo apt-cache search --names-only festvox-*
```

To install the `festvox-don` voice (for example), run the command:

```
$ sudo apt-get install festvox-don
```

And to test out your new voice, add the voice name to the end of the command line like this:

```
$ rosrun sound_play say.py "Welcome to the future" voice_don_diphone
```

There aren't a huge number of voices to choose from, but a few additional voices can be installed as described here and demonstrated here. Here are the steps to get and use two of those voices, one male and one female:

```
$ sudo apt-get install festlex-cmu
$ cd /usr/share/festival/voices/english/
$ sudo wget -c \
  http://www.speech.cs.cmu.edu/cmu_arctic/packed/cmu_us_clb_arctic\
  -0.95-release.tar.bz2
$ sudo wget -c \
  http://www.speech.cs.cmu.edu/cmu_arctic/packed/cmu_us_bdl_arctic\
  -0.95-release.tar.bz2
$ sudo tar jxfv cmu_us_clb_arctic-0.95-release.tar.bz2
$ sudo tar jxfv cmu_us_bdl_arctic-0.95-release.tar.bz2
$ sudo rm cmu_us_clb_arctic-0.95-release.tar.bz2
$ sudo rm cmu_us_bdl_arctic-0.95-release.tar.bz2
$ sudo ln -s cmu_us_clb_arctic cmu_us_clb_arctic_clunits
$ sudo ln -s cmu_us_bdl_arctic cmu_us_bdl_arctic_clunits
```

You can test these two voices like this:

```
$ rosrun sound_play say.py "I am speaking with a female C M U voice" \
  voice_cmu_us_clb_arctic_clunits
$ rosrun sound_play say.py "I am speaking with a male C M U voice" \
  voice_cmu_us_bdl_arctic_clunits
```

NOTE: If you don't hear the phrase on the first try, try re-running the command. Also, remember that a `sound_play` node must already be running in another terminal.

You can also use `sound_play` to play wave files or a number of built-in sounds. To play the R2D2 wave file in the `rbx1_speech/sounds` directory, use the command:

```
$ rosrun sound_play play.py `rospack find rbx1_speech`/sounds/R2D2a.wav
```

Note that the `play.py` script requires the absolute path to the wave file which is why we used '`rospack find`'. You could also just type out the full path name.

In previous versions of ROS, the script `playbuiltin.py` could be used to play a number of built-in sounds. Unfortunately, this script currently does not work in Hydro. The issue has been reported here.

9.5.1 Using Text-to-Speech within a ROS Node

So far we have only used the Festival voices from the command line. To see how to use text-to-speech from within a ROS node, let's take a look at the `talkback.py` script that can be found in the `rbx1_speech/nodes` directory.

Link to source: talkback.py

```
1.  #!/usr/bin/env python
2.
3.  import rospy
4.  from std_msgs.msg import String
5.  from sound_play.libsoundplay import SoundClient
6.  import sys
7.
8.  class TalkBack:
9.      def __init__(self, script_path):
10.         rospy.init_node('talkback')
11.
12.         rospy.on_shutdown(self.cleanup)
13.
14.         # Set the default TTS voice to use
15.         self.voice = rospy.get_param("~voice", "voice_don_diphone")
16.
17.         # Set the wave file path if used
18.         self.wavepath = rospy.get_param("~wavepath", script_path +
    "/../sounds")
19.
20.         # Create the sound client object
21.         self.soundhandle = SoundClient()
22.
23.         # Wait a moment to let the client connect to the
24.         # sound_play server
25.         rospy.sleep(1)
26.
27.         # Make sure any lingering sound_play processes are stopped.
28.         self.soundhandle.stopAll()
29.
30.         # Announce that we are ready for input
31.         self.soundhandle.playWave(self.wavepath + "/R2D2a.wav")
32.         rospy.sleep(1)
33.         self.soundhandle.say("Ready", self.voice)
34.
35.         rospy.loginfo("Say one of the navigation commands...")
36.
37.         # Subscribe to the recognizer output and set the callback
    function
38.         rospy.Subscriber('/recognizer/output', String, self.talkback)
39.
40.     def talkback(self, msg):
41.         # Print the recognized words on the screen
42.         rospy.loginfo(msg.data)
43.
44.         # Speak the recognized words in the selected voice
45.         self.soundhandle.say(msg.data, self.voice)
46.
```

```
47.          # Uncomment to play one of the built-in sounds
48.          #rospy.sleep(2)
49.          #self.soundhandle.play(5)
50.
51.          # Uncomment to play a wave file
52.          #rospy.sleep(2)
53.          #self.soundhandle.playWave(self.wavepath + "/R2D2a.wav")
54.
55.      def cleanup(self):
56.          self.soundhandle.stopAll()
57.          rospy.loginfo("Shutting down talkback node...")
58.
59.  if __name__=="__main__":
60.      try:
61.          TalkBack(sys.path[0])
62.          rospy.spin()
63.      except rospy.ROSInterruptException:
64.          rospy.loginfo("AMCL navigation test finished.")
```

Let's look at the key lines of the script.

```
5.  from sound_play.libsoundplay import SoundClient
```

The script talks to the `sound_play` server using the `SoundClient` class which we import from the `sound_play` library.

```
15.          self.voice = rospy.get_param("~voice", "voice_don_diphone")
```

Set the Festival voice for text-to-speech. This can be overridden in the launch file.

```
18.          self.wavepath = rospy.get_param("~wavepath", script_path +
    "/../sounds")
```

Set the path for reading wave files. The `sound_play` client and server need the full path to wave files so we read in the script_path from the `sys.path[0]` environment variable (see the `"__main__"` block toward the end of the script.)

```
21.          self.soundhandle = SoundClient()
```

Create a handle to the `SoundClient()` object.

```
30.          self.soundhandle.playWave(self.wavepath + "/R2D2a.wav")
31.          rospy.sleep(1)
32.          self.soundhandle.say("Ready", self.voice)
```

Use the `self.soundhandle` object to play a short wave file (R2D2 sound) and speak "Ready" in the default voice.

```
40.      def talkback(self, msg):
41.          # Print the recognized words on the screen
```

```
42.         rospy.loginfo(msg.data)
43.
44.         # Speak the recognized words in the selected voice
45.         self.soundhandle.say(msg.data, self.voice)
```

This is our callback when receiving messages on the `/recognizer/output` topic. The `msg` variable holds the text spoken by the user as recognized by the PocketSphinx node. As you can see, we simply echo the recognized text back to the terminal as well as speak it using the default voice.

To learn more about the `SoundClient` object, take a look at the libsoundplay API on the ROS Wiki.

9.5.2 Testing the talkback.py script

You can test the script using the `talkback.launch` file. The launch file first brings up the PocketSphinx `recognizer` node and loads the navigation phrases. Then the `sound_play` node is launched, followed by the `talkback.py` script. So first terminate any `sound_play` node you might still have running, then run the command:

```
$ roslaunch rbx1_speech talkback.launch
```

Now try saying one of the voice navigation commands we used earlier such as "move right" and you should hear it echoed back by the TTS voice.

You should now be able to write your own script that combines speech recognition and text-to-speech. For example, see if you can figure out how to ask your robot the date and time and get back the answer from the system clock.

10. Robot Vision

One might say that we are living in a golden age for computer vision. Webcams are cheap and depth cameras like the Microsoft Kinect and Asus Xtion allow even a hobby roboticist to work with 3D vision without breaking the bank on an expensive stereo camera. But getting the pixels and depth values into your computer is just the beginning. Using that data to extract useful information about the visual world is a challenging mathematical problem. Fortunately for us, decades of research by thousands of scientists has yielded powerful vision algorithms from simple color matching to people detectors that we can use without having to start from scratch.

The overall goal of machine vision is to recognize the structure of the world behind the changing pixel values. Individual pixels are in a continual state of flux due to changes in lighting, viewing angle, object motion, occlusions and random noise. So computer vision algorithms are designed to extract more stable *features* from these changing values. Features might be corners, edges, blobs, colors, motion patches, etc. Once a set of robust features can be extracted from an image or video stream, they can be tracked or grouped together into larger patterns to support object detection and recognition.

10.1 OpenCV, OpenNI and PCL

The three pillars of computer vision in the ROS community are OpenCV, OpenNI and PCL. OpenCV is used for 2D image processing and machine learning. OpenNI provides drivers and a "Natural Interaction" library for skeleton tracking using depth cameras such as the Kinect and Xtion. And PCL, or Point Cloud Library, is the library of choice for processing 3D point clouds. In this book, we will focus on OpenCV but we will also provide a brief introduction to OpenNI and PCL. (For those readers already familiar with OpenCV and PCL, you might also be interested in Ecto, a new vision framework from Willow Garage that allows one to access both libraries through a common interface.)

In this chapter we will learn how to:

- connect to a webcam or RGB-D (depth) camera using ROS
- use the ROS `cv_bridge` utility for processing ROS image streams with OpenCV
- write ROS programs to detect faces, track keypoints using optical flow, and follow objects of a particular color
- track a user's skeleton using an RGB-D camera and OpenNI

- detect the nearest person using PCL

10.2 A Note about Camera Resolutions

Most video cameras used in robotics can be run at various resolutions, typically 160x120 (QQVGA), 320x240 (QVGA), 640x480 (VGA) and sometimes as high as 1280x960 (SXGA). It is often tempting to set the camera to the highest resolution possible under the assumption that more detail is better. However, more pixels come at the cost of more computation per frame. For example, each frame of a 640x480 video contains four times as many pixels as a 320x240 video. That means four times as many operations per frame for most kinds of basic image processing. If your CPU and graphics processor are already maxed out when processing a 320x240 video at 20 frames per second (fps), you will be lucky to get 5 fps for a 640x480 video which is really too slow for most mobile robots.

Experience shows that a resolution of 320x240 (QVGA) produces a good balance between image detail and computational load. Anything less and functions like face detection tend to fail due to the lack of detail. Anything more and you will probably suffer from insufficient frame rates and delays in other processes you may need to run at the same time. Consequently, the camera launch files used in this book set the resolution to 320x240. Of course, if you have a ripping fast PC controlling your robot, by all means use the highest resolution that still gives you good performance.

10.3 Installing and Testing the ROS Camera Drivers

If you haven't done so already, install the drivers necessary for your camera following the instructions below.

10.3.1 Installing the OpenNI Drivers

To install the ROS OpenNI drivers for the Microsoft Kinect or Asus Xtion, use the command:

```
$ sudo apt-get install ros-hydro-openni-camera
```

That's all there is to it!

10.3.2 Installing Webcam Drivers

To use a typical USB webcam, there are a number of possible driver packages available. The one we will use here is from Eric Perko's repository. Type the following commands to move into your personal ROS directory, retrieve the source code, then make the package:

```
$ sudo apt-get install git-core
$ cd ~/ros_workspace
$ git clone https://github.com/ericperko/uvc_cam.git
$ rosmake uvc_cam
```

IMPORTANT: If you were using an earlier version of the `uvc_cam` package with ROS Electric, Fuerte or Groovy, be sure to get and rebuild the latest version as follows:

```
$ roscd uvc_cam
$ git pull
$ rosmake --pre-clean
```

10.3.3 Testing your Kinect or Xtion Camera

Once you have the OpenNI driver installed, make sure you can see the video stream from the camera by using the ROS `image_view` package. For the Kinect or Xtion, first plug the camera in to any available USB port (and for the Kinect, make sure it has power through its 12V adapter or other means), then run the following launch file:

```
$ roslaunch rbx1_vision openni_node.launch
```

If the connection to the camera is successful, you should see a series of diagnostic messages that look something like this:

```
[ INFO] [1384877526.285283926]: Number devices connected: 1
[ INFO] [1384877526.285412166]: 1. device on bus 002:77 is a SensorV2
(2ae) from PrimeSense (45e) with serial id 'A00365A20145047A'
[ INFO] [1384877526.286543142]: Searching for device with index = 1
[ INFO] [1384877526.341836602]: Opened 'SensorV2' on bus 2:77 with
serial number 'A00365A20145047A'
process[camera_base_link2-20]: started with pid [17928]
process[camera_base_link3-21]: started with pid [17946]
[ INFO] [1384877527.436366816]: rgb_frame_id =
'/camera_rgb_optical_frame'
[ INFO] [1384877527.436487549]: depth_frame_id =
'/camera_depth_optical_frame'
```

NOTE: Don't worry if you see an error message that begins "Skipping XML Document..." or a few warning warning messages about the camera calibration file not being found. These messages can be ignored.

Next, use the ROS `image_view` utility to view the RGB video stream. We have set up our camera launch files so that the color video stream is published on the ROS topic `/camera/rgb/image_color`. To view the video, we therefore run:

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

A small camera display window should pop up and after a brief delay, you should see the live video stream from your camera. Move something in front of the camera to verify that the image updates appropriately. Once you are satisfied that you have a live video, close the `image_view` window or type `Ctrl-C` in the terminal from which you launched it.

You can also test the depth image from your camera using the ROS `disparity_view` node:

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

In this case, the resulting colors in the image represent depth, with red/yellow indicating points closer to the camera, blue/violet representing points further away and green for points with intermediate depth. Note that if you move an object inside the camera's minimum depth range (about 50cm), the object will turn grey indicating that the depth value is no longer valid at this range.

10.3.4 Testing your USB Webcam

For a USB camera, we need to specify the video device we want to use. If your computer has an internal camera (as many laptop's do), it will likely be on `/dev/video0` while an externally attached USB camera will probably be on `/dev/video1`. Be sure to `Ctrl-C` out of the `openni_node.launch` file from the previous section if it is still running, then run the appropriate command below depending on your webcam's video device:

```
$ roslaunch rbx1_vision uvc_cam.launch device:=/dev/video0
```

or

```
$ roslaunch rbx1_vision uvc_cam.launch device:=/dev/video1
```

depending on the camera you want use. If the connection is successful, you should see a stream of diagnostic messages describing various camera settings. Don't worry if some of these messages indicate that a parameter could not be set.

Next, use the ROS `image_view` utility to view the basic video stream. We have set up our camera launch files so that the color video stream is published on the ROS topic `/camera/rgb/image_color`. To view the video, we therefore run:

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

A small camera display window should pop up and, after a brief delay, you should see the live video stream from your camera. Move something in front of the camera to verify that the image updates appropriately.

NOTE: By default, the `uvc_cam` node publishes the image on the topic `/camera/image_raw`. Our launch file `uvc_cam.launch` remaps the topic to `/camera/rgb/image_color` which is the topic used by the openni driver for depth cameras. This way we can use the same code below for either type of camera.

10.4 Installing OpenCV on Ubuntu Linux

The easiest way to install OpenCV under Ubuntu Linux is to get the Debian packages. At the time of this writing, the latest version is 2.4 which can be installed with the following command:

```
$ sudo apt-get install ros-hydro-opencv2 ros-hydro-vision-opencv
```

If you need a feature found only in the latest version, you can compile the library from source. Instructions can be found here:

http://docs.opencv.org/doc/tutorials/introduction/linux_install/linux_install.html

To check your installation, try the following commands:

```
$ python
>>> from cv2 import cv
>>> quit()
```

Assuming you do not get an import error, you should be ready to go and you can skip to the next section. If instead you get an error of the form:

```
ImportError: No module named cv2
```

then either OpenCV is not install properly or your Python path is not set correctly. The OpenCV Python library is stored in the file `cv2.so`. To verify that it is installed, run the command:

```
$ locate cv2.so | grep python
```

You should get an output similar to:

```
/opt/ros/hydro/lib/python2.7/dist-packages/cv2.so
```

(If you have other ROS distributions installed on the same computer, you will likely see their copy of `cv2.so` listed as well.)

10.5 ROS to OpenCV: The `cv_bridge` Package

With the camera drivers up and running, we now need a way to process ROS video streams using OpenCV. ROS provides the `cv_bridge` utility to convert between ROS and OpenCV and image formats. The Python script `cv_bridge.demo.py` in the `rbx1_vision/nodes` directory demonstrates how to use `cv_bridge`. Before we look at the code, you can try it out as follows.

If you have a Kinect or Xtion, make sure you first run the OpenNI driver if it is not already running:

```
$ roslaunch rbx1_vision openni_node.launch
```

or, for a webcam:

```
$ roslaunch rbx1_vision uvc_cam.launch device:=/dev/video0
```

(Change the video device is necessary.)

Now run the `cv_bridge_demo.py` node:

```
$ rosrun rbx1_vision cv_bridge_demo.py
```

After a brief delay, you should see two image windows appear. The window on the top shows the live video after been converted to greyscale then sent through OpenCV blur and edge filters. The window on the bottom shows a greyscale depth image where white pixels are further away and dark grey pixels are closer to the camera. (This window will remain blank if you are using an ordinary webcam.) To exit the demo, either type the letter "q" with the mouse over one of the windows or `Ctrl-C` in the terminal where you launched the demo script.

Let's now look at the code to see how it works.

Link to source: `cv_bridge_demo.py`

```
1.  #!/usr/bin/env python
2.
3.  import rospy
4.  import sys
5.  import cv2
6.  import cv2.cv as cv
7.  from sensor_msgs.msg import Image, CameraInfo
8.  from cv_bridge import CvBridge, CvBridgeError
9.  import numpy as np
10.
11.  class cvBridgeDemo():
12.      def __init__(self):
13.          self.node_name = "cv_bridge_demo"
```

```
14.
15.             rospy.init_node(self.node_name)
16.
17.             # What we do during shutdown
18.             rospy.on_shutdown(self.cleanup)
19.
20.             # Create the OpenCV display window for the RGB image
21.             self.cv_window_name = self.node_name
22.             cv.NamedWindow(self.cv_window_name, cv.CV_WINDOW_NORMAL)
23.             cv.MoveWindow(self.cv_window_name, 25, 75)
24.
25.             # And one for the depth image
26.             cv.NamedWindow("Depth Image", cv.CV_WINDOW_NORMAL)
27.             cv.MoveWindow("Depth Image", 25, 350)
28.
29.             # Create the cv_bridge object
30.             self.bridge = CvBridge()
31.
32.             # Subscribe to the camera image and depth topics and set
33.             # the appropriate callbacks
34.             self.image_sub = rospy.Subscriber("/camera/rgb/image_color",
35.                                               Image, self.image_callback
36.             self.depth_sub = rospy.Subscriber("/camera/depth/image_raw",
37.                                               Image, self.depth_callback)
38.
39.             rospy.loginfo("Waiting for image topics...")
40.
41.         def image_callback(self, ros_image):
42.             # Use cv_bridge() to convert the ROS image to OpenCV format
43.             try:
44.                 frame = self.bridge.imgmsg_to_cv(ros_image, "bgr8")
45.             except CvBridgeError, e:
46.                 print e
47.
48.             # Convert the image to a Numpy array since most cv2 functions
49.             # require Numpy arrays.
50.             frame = np.array(frame, dtype=np.uint8)
51.
52.             # Process the frame using the process_image() function
53.             display_image = self.process_image(frame)
54.
55.             # Display the image.
56.             cv2.imshow(self.node_name, display_image)
57.
58.             # Process any keyboard commands
59.             self.keystroke = cv.WaitKey(5)
60.             if 32 <= self.keystroke and self.keystroke < 128:
61.                 cc = chr(self.keystroke).lower()
62.                 if cc == 'q':
63.                     # The user has press the q key, so exit
64.                     rospy.signal_shutdown("User hit q key to quit.")
65.
66.         def depth_callback(self, ros_image):
67.             # Use cv_bridge() to convert the ROS image to OpenCV format
68.             try:
69.                 # The depth image is a single-channel float32 image
70.                 depth_image = self.bridge.imgmsg_to_cv(ros_image, "32FC1")
```

```
71.             except CvBridgeError, e:
72.                 print e
73.
74.             # Convert the depth image to a Numpy array since most cv2
functions
75.             # require Numpy arrays.
76.             depth_array = np.array(depth_image, dtype=np.float32)
77.
78.             # Normalize the depth image to fall between 0 and 1
79.             cv2.normalize(depth_array, depth_array, 0, 1, cv2.NORM_MINMAX)
80.
81.             # Process the depth image
82.             depth_display_image = self.process_depth_image(depth_array)
83.
84.             # Display the result
85.             cv2.imshow("Depth Image", depth_display_image)
86.
87.         def process_image(self, frame):
88.             # Convert to greyscale
89.             grey = cv2.cvtColor(frame, cv.CV_BGR2GRAY)
90.
91.             # Blur the image
92.             grey = cv2.blur(grey, (7, 7))
93.
94.             # Compute edges using the Canny edge filter
95.             edges = cv2.Canny(grey, 15.0, 30.0)
96.
97.             return edges
98.
99.         def process_depth_image(self, frame):
100.            # Just return the raw image for this demo
101.            return frame
102.
103.        def cleanup(self):
104.            print "Shutting down vision node."
105.            cv2.destroyAllWindows()
106.
107.    def main(args):
108.        try:
109.            cvBridgeDemo()
110.            rospy.spin()
111.        except KeyboardInterrupt:
112.            print "Shutting down vision node."
113.            cv.DestroyAllWindows()
114.
115.    if __name__ == '__main__':
116.        main(sys.argv)
```

Let's take a look at the key lines in this script.

```
5.  import cv2
6.  import cv2.cv as cv
7.  from sensor_msgs.msg import Image, CameraInfo
8.  from cv_bridge import CvBridge, CvBridgeError
9.  import numpy as np
```

All of our OpenCV scripts will import the `cv2` library as well as the older pre-cv2 functions found in `cv2.cv`. We will also usually need the ROS message types Image and `CameraInfo` from the `sensor_msgs` package. For a node that needs to convert ROS image format to OpenCV, we need the `CvBridge` and `CvBridgeError` classes from the ROS `cv_bridge` package. Finally, OpenCV does most of its image processing using Numpy arrays, so we almost always need access to the Python `numpy` module.

```
20.         # Create the OpenCV display window for the RGB image
21.         self.cv_window_name = self.node_name
22.         cv.NamedWindow(self.cv_window_name, cv.CV_WINDOW_NORMAL)
23.         cv.MoveWindow(self.cv_window_name, 25, 75)
24.
25.         # And one for the depth image
26.         cv.NamedWindow("Depth Image", cv.CV_WINDOW_NORMAL)
27.         cv.MoveWindow("Depth Image", 25, 350)
```

If you are already familiar with OpenCV, you'll recognize these statements that create named display windows for monitoring video streams; in this case, one window for the regular RGB video and one for the depth image if we are using an RGB-D camera. We also move the windows so that the depth window lies below the RGB window.

```
30.         self.bridge = CvBridge()
```

This is how we create the `CvBridge` object to be used later to convert ROS images to OpenCV format.

```
34.         self.image_sub = rospy.Subscriber("/camera/rgb/image_color",
35.                                           Image, self.image_callback)
36.         self.depth_sub = rospy.Subscriber("/camera/depth/image_raw",
37.                                           Image, self.depth_callback)
```

These are the two key subscribers, one for the RGB image stream and one for the depth image. Normally we would not hard code the topic names so that they can be remapped in the appropriate launch file but for the demo we'll use the default topic names used by the `openni` node. As with all subscribers, we assign the callback functions for doing the actual work on the images.

```
41.     def image_callback(self, ros_image):
42.         # Use cv_bridge() to convert the ROS image to OpenCV format
43.         try:
44.             frame = self.bridge.imgmsg_to_cv(ros_image, "bgr8")
45.         except CvBridgeError, e:
46.             print e
```

This is the start of our callback function for the RGB image. ROS provides the image as the first argument which we have called `ros_image`. The `try-except` block then uses the `imgmsg_to_cv` function to convert the image to OpenCV format assuming a blue/green/red 8-bit conversion.

```
50.         frame = np.array(frame, dtype=np.uint8)
51.
52.         # Process the frame using the process_image() function
53.         display_image = self.process_image(frame)
54.
55.         # Display the image.
56.         cv2.imshow(self.node_name, display_image)
```

Most OpenCV functions require the image to be converted to a Numpy array so we make the conversion here. Then we send the image array to the `process_image()` function which we will describe below. The result is a new image called `display_image` which is displayed in our previously-created display window using the OpenCV `imshow()` function.

```
58.         # Process any keyboard commands
59.         self.keystroke = cv.WaitKey(5)
60.         if 32 <= self.keystroke and self.keystroke < 128:
61.             cc = chr(self.keystroke).lower()
62.             if cc == 'q':
63.                 # The user has press the q key, so exit
64.                 rospy.signal_shutdown("User hit q key to quit.")
```

Finally, we look for keyboard input from the user. In this case, we only look for a key press of the letter "`q`" which signals that we want to terminate the script.

```
64.     def depth_callback(self, ros_image):
65.         # Use cv_bridge() to convert the ROS image to OpenCV format
66.         try:
67.             # The depth image is a single-channel float32 image
68.             depth_image = self.bridge.imgmsg_to_cv(ros_image, "32FC1")
69.         except CvBridgeError, e:
70.             print e
```

The depth callback begins similar to the image callback described earlier. The first difference we notice is that we use a conversion to a single-channel 32-bit float (32FC1) instead of the 3-channel 8 bit color conversion used for the RGB image.

```
76.         depth_array = np.array(depth_image, dtype=np.float32)
77.
78.         # Normalize the depth image to fall between 0 and 1
79.         cv2.normalize(depth_array, depth_array, 0, 1, cv2.NORM_MINMAX)
```

After converting the image to a Numpy array, we normalize to the interval [0, 1] since OpenCV's `imshow()` function can display greyscale images when the pixel values lie between 0 and 1.

```
82.         depth_display_image = self.process_depth_image(depth_array)
83.
84.         # Display the result
85.         cv2.imshow("Depth Image", depth_display_image)
```

The depth array is sent to the `process_depth_image()` function for further processing (if desired) and the result is displayed in the depth image window created earlier.

```
87.     def process_image(self, frame):
88.         # Convert to greyscale
89.         grey = cv2.cvtColor(frame, cv.CV_BGR2GRAY)
90.
91.         # Blur the image
92.         grey = cv2.blur(grey, (7, 7))
93.
94.         # Compute edges using the Canny edge filter
95.         edges = cv2.Canny(grey, 15.0, 30.0)
96.
97.         return edges
```

Recall that the image callback function in turn called the `process_image()` function in case we want to manipulate the image before displaying it back to the user. For the purposes of this demonstration, here we convert the image to greyscale, blur it using a Gaussian filter with a variance of 7 pixels in the x and y dimensions, then compute the Canny edges over the result. The edge image is returned to the callback where it is displayed to the user.

```
99.     def process_depth_image(self, frame):
100.        # Just return the raw image for this demo
101.        return frame
```

In this demo, we do nothing with the depth image and just return the original frame. Feel free to add your own set of OpenCV filters here.

10.6 The `ros2opencv2.py` Utility

Many of our ROS vision nodes will share a common set of functions such as converting from ROS to OpenCV image format using `cv_bridge`, drawing text on the screen, allowing the user to selection regions with the mouse and so on. We will therefore begin by programming a script that takes care of these common tasks and can be included in other nodes.

The file `ros2opencv2.py` found in the `rbx1_vision/src/rbx1_vision` subdirectory performs the following tasks:

- Subscribes to the raw image topic that is published by the camera driver. The format of this image is defined by the ROS `sensor_msgs/Image` message type.
- Creates an instance of the ROS cv_bridge utility that converts the ROS image format into OpenCV format.

- Creates an OpenCV display window for monitoring the image.
- Creates a callback for processing mouse clicks by the user; e.g., selecting a region to track.
- Creates a `process_image()` function that will do all the work of processing the image and returning the results.
- Creates a `process_depth_image()` function for processing a depth image.
- Publishes a region of interest (ROI) on the `/roi` topic that contains the pixels or keypoints that are returned by `process_image()`.
- Stores the currently detected target's location in the global variable `self.detect_box`.
- Stores the currently tracked target's location in the global variable `self.track_box`.

Other ROS nodes that we develop below will override the functions `process_image()` and/or `process_depth_image()` and detect specific features such as faces or colors as well as track keypoints found within the detected region.

Most of the code in this script is fairly straightforward and simply expands on the `cv_bridge_demo.py` script we saw earlier so we won't run through it line-by-line. (The script is heavily commented and should give you a good idea what each part does. You can also view it online here:

Link to source: ros2opencv2.py

But before moving on to more complex vision processing, let's test the `ros2opencv2.py` node on its own. With your camera's driver up and running, move to another terminal and run the `ros2opencv2.launch` file as follows:

```
$ roslaunch rbx1_vision ros2opencv2.launch
```

After a brief pause, you should see the OpenCV display window appear. By default, you will also see the processing speed (CPS = cycles per second) and the image resolution. (Cycles per second is the reciprocal of the time it take to process a single frame. It is therefore an estimate of the *fastest* frame rate we could process.) You can resize the window by dragging one of the corners. If you click on a point in the image, a small yellow circle or dot will appear on the image in that location. If you drag across the image, a selection rectangle will be drawn in yellow until you release the mouse at which point it turns green and remains on the image. The green box represents your region of interest (ROI). To see the coordinates of the ROI, open another terminal and view the `/roi` topic:

```
$ rostopic echo /roi
```

Try drawing different rectangles on the image window and you will see the values of the ROI fields change accordingly. The meaning of these fields are:

- `x_offset`: x-coordinate of the upper left corner of the region
- `y_offset`: y-coordinate of the upper left corner of the region
- `height`: height of the region in pixels
- `width` : width of the region in pixels
- `do_rectify`: boolean value (True or `False`). Usually `False` which means that the ROI is defined with respect to the whole image. However, if `True` then the ROI is defined within a sub-window of the image.

Note that the offset coordinates are relative to the upper left corner of the image window which has coordinates (0, 0). Positive x values increase to the right while y values increase downward. The largest x value is `width-1` and the largest y value is `height-1`.

10.7 Processing Recorded Video

The `rbx1_vision` package also contains a node called `video2ros.py` for converting recorded video files into a ROS video stream so that you can use it instead of a live camera. To test the node, **terminate any camera drivers** you may have running in another terminal. Also terminate the `ros2opencv2.py` node if it is still running. Then run the following commands:

```
$ rosrun rbx1_vision cv_bridge_demo.py
```

```
[INFO] [WallTime: 1362334257.368930] Waiting for image topics...
```

```
$ roslaunch rbx1_vision video2ros.launch input:=`rospack find \
  rbx1_vision`/videos/hide2.mp4
```

(The test video comes courtesy of the Honda/UCSD video database.)

You should see two active video display windows. (The depth video window will remain blank.) The display window called "Video Playback" allows you to control the recorded video: click anywhere on the window to bring it to the foreground, then hit the **Space Bar** to pause/continue the video and hit the "`r`" key to restart the video from the

beginning. The other window displays the output from our `cv_bridge_demo.py` node which, as you will recall, computes the edge map of the input.

The `video2ros.py` script is heavily commented and fairly self-explanatory. You can find the source online at the following link:

Link to source: `video2ros.py`

Now that our basic vision nodes are working, we are ready to try out a number of OpenCV's vision processing functions.

10.8 OpenCV: The Open Source Computer Vision Library

OpenCV was developed in 1999 by Intel to test CPU intensive applications and the code was released to the public in 2000. In 2008, primary development was taken over by Willow Garage. OpenCV is not as easy to use as some GUI-based vision packages such as RoboRealm for Windows. However, the functions available in OpenCV represent many state-of-the-art vision algorithms as well as methods for machine learning such as Support Vector Machines, artificial neural networks and Random Trees.

OpenCV can be run as a standalone library on Linux, Windows, MacOS X and Android. For those new to OpenCV, please note that we will only touch on a small fraction of OpenCV's capabilities. For a complete introduction to the library and all its features, please refer to Learning OpenCV by Gary Bradski and Adrian Kaehler. You can also refer to the complete online manual including a number of introductory tutorials.

10.8.1 Face Detection

OpenCV makes it relatively easy to detect faces in an image or video stream. And since this is a popular request for those interested in robot vision, it is a good place to start.

OpenCV's face detector uses a Cascade Classifier with Haar-like features. You can learn more about cascade classifiers and Haar features at the provided link. For now, all we need to understand is that the OpenCV cascade classifier can be initialized with different XML files that define the object we want to detect. We will use two of these files to detect a face when seen directly from the front. Another file will allow us to detect a face when seen from the side (profile view). These files were created by training machine learning algorithms on hundreds or even thousands of images that either contain a face or do not. The learning algorithm is then able to extract the features that characterize faces and the results are stored in XML format. (Additional cascade files have been trained for detecting eyes and even whole people.)

A few of these XML files have been copied from the OpenCV source tree to the `rbx1_vision/data/haar_detectors` directory and we will use them in our scripts below.

Our ROS face detector node is located in the file `face_detector.py` in the `rbx1_vision/src/rbx1_vision` directory. Before we look at the code, let's give it a try.

To run the detector, first launch the appropriate video driver. For the Kinect or Xtion:

```
$ roslaunch rbx1_vision openni_node.launch
```

Or for a webcam:

```
$ roslaunch rbx1_vision uvc_cam.launch device:=/dev/video0
```

(Change the video device if necessary.)

Now launch the face detector node:

```
$ roslaunch rbx1_vision face_detector.launch
```

If you position your face within the frame of the camera, you should see a green box around your face when the cascade detector finds it. When the detector loses your face, the box disappears and you will see the message "LOST FACE!" on the screen. Try turning your face from side to side and up and down. Also try moving your hand in front of your face. The "Hit Rate" number also displayed on the screen is the number of frames in which your face was detected divided by the total number of frames so far.

As you can see, the detector is pretty good, but it has limitations–the detector loses your face when you turn your head too far from either a frontal or side view. Nonetheless, since people (and robots) tend to interact while facing each other, the detector can get the job done is many situations.

Let's now take a look the code.

Link to source: face_detector.py

```
1.  #!/usr/bin/env python
2.
3.  import rospy
4.  import cv2
5.  import cv2.cv as cv
6.  from rbx1_vision.ros2opencv2 import ROS2OpenCV2
7.
8.  class FaceDetector(ROS2OpenCV2):
9.      def __init__(self, node_name):
10.         super(FaceDetector, self).__init__(node_name)
11.
12.         # Get the paths to the cascade XML files for the Haar detectors.
13.         # These are set in the launch file.
14.         cascade_1 = rospy.get_param("~cascade_1", "")
```

```
15.             cascade_2 = rospy.get_param("~cascade_2", "")
16.             cascade_3 = rospy.get_param("~cascade_3", "")
17.
18.             # Initialize the Haar detectors using the cascade files
19.             self.cascade_1 = cv2.CascadeClassifier(cascade_1)
20.             self.cascade_2 = cv2.CascadeClassifier(cascade_2)
21.             self.cascade_3 = cv2.CascadeClassifier(cascade_3)
22.
23.             # Set cascade parameters that tend to work well for faces.
24.             # Can be overridden in launch file
25.             self.haar_minSize = rospy.get_param("~haar_minSize", (20, 20))
26.             self.haar_maxSize = rospy.get_param("~haar_maxSize", (150, 150))
27.             self.haar_scaleFactor = rospy.get_param("~haar_scaleFactor", 1.3)
28.             self.haar_minNeighbors = rospy.get_param("~haar_minNeighbors", 1)
29.             self.haar_flags = rospy.get_param("~haar_flags",
cv.CV_HAAR_DO_CANNY_PRUNING)
30.
31.             # Store all parameters together for passing to the detector
32.             self.haar_params = dict(minSize = self.haar_minSize,
33.                                     maxSize = self.haar_maxSize,
34.                                     scaleFactor = self.haar_scaleFactor,
35.                                     minNeighbors = self.haar_minNeighbors,
36.                                     flags = self.haar_flags)
37.
38.             # Do we should text on the display?
39.             self.show_text = rospy.get_param("~show_text", True)
40.
41.             # Initialize the detection box
42.             self.detect_box = None
43.
44.             # Track the number of hits and misses
45.             self.hits = 0
46.             self.misses = 0
47.             self.hit_rate = 0
48.
49.         def process_image(self, cv_image):
50.             # Create a greyscale version of the image
51.             grey = cv2.cvtColor(cv_image, cv2.COLOR_BGR2GRAY)
52.
53.             # Equalize the histogram to reduce lighting effects
54.             grey = cv2.equalizeHist(grey)
55.
56.             # Attempt to detect a face
57.             self.detect_box = self.detect_face(grey)
58.
59.             # Did we find one?
60.             if self.detect_box is not None:
61.                 self.hits += 1
62.             else:
63.                 self.misses += 1
64.
65.             # Keep tabs on the hit rate so far
66.             self.hit_rate = float(self.hits) / (self.hits + self.misses)
67.
68.             return cv_image
69.
70.         def detect_face(self, input_image):
```

```
            # First check one of the frontal templates
            if self.cascade_1:
                faces = self.cascade_1.detectMultiScale(input_image,
**self.haar_params)

            # If that fails, check the profile template
            if len(faces) == 0 and self.cascade_3:
                faces =
self.cascade_3.detectMultiScale(input_image,**self.haar_params)

            # If that also fails, check a the other frontal template
            if len(faces) == 0 and self.cascade_2:
                faces = self.cascade_2.detectMultiScale(input_image,
**self.haar_params)

            # The faces variable holds a list of face boxes.
            # If one or more faces are detected, return the first one.
            if len(faces) > 0:
                face_box = faces[0]
            else:
                # If no faces were detected, print the "LOST FACE" message on
the screen
                if self.show_text:
                    font_face = cv2.FONT_HERSHEY_SIMPLEX
                    font_scale = 0.5
                    cv2.putText(self.marker_image, "LOST FACE!",
                                (int(self.frame_size[0] * 0.65),
int(self.frame_size[1] * 0.9)),
                                font_face, font_scale, cv.RGB(255, 50, 50))
                face_box = None

            # Display the hit rate so far
            if self.show_text:
                font_face = cv2.FONT_HERSHEY_SIMPLEX
                font_scale = 0.5
                cv2.putText(self.marker_image, "Hit Rate: " +
                     str(trunc(self.hit_rate, 2)),
                     (20, int(self.frame_size[1] * 0.9)),
                     font_face, font_scale, cv.RGB(255, 255, 0))

        return face_box

    def trunc(f, n):
        '''Truncates/pads a float f to n decimal places without rounding'''
        slen = len('%.*f' % (n, f))
        return float(str(f)[:slen])

    if __name__ == '__main__':
        try:
            node_name = "face_detector"
            FaceDetector(node_name)
            rospy.spin()
        except KeyboardInterrupt:
            print "Shutting down face detector node."
            cv2.destroyAllWindows()
```

Let's take a look at the key lines of the script.

```
6.  from rbx1_vision.ros2opencv2 import ROS2OpenCV2
7.
8.  class FaceDetector(ROS2OpenCV2):
9.      def __init__(self, node_name):
10.         super(FaceDetector, self).__init__(node_name)
```

First we must import the `ROS2OpenCV2` class from the `ros2opencv2.py` script that we developed earlier. The face detector node is then defined as a class that extends the `ROS2OpenCV2` class. In this way it inherits all the housekeeping functions and variables from the `ros2opencv2.py` script such as user selections with the mouse, displaying a box around the ROI and so on. Whenever we extend a class, we have to initialize the parent class as well which is done with Python's `super()` function as shown in the last line above.

```
14.         cascade_1 = rospy.get_param("~cascade_1", "")
15.         cascade_2 = rospy.get_param("~cascade_2", "")
16.         cascade_3 = rospy.get_param("~cascade_3", "")
```

These three parameters store the path names to the XML files we want to use for the Haar cascade detector. The paths are specified in the launch file `rbx1_vision/launch/face_detector.launch`. The XML files themselves are not included in the OpenCV or ROS Hydro Debian packages so they were copied from the OpenCV source to the `ros-by-example` repository and can be found in the directory `rbx1_vision/data/haar_detectors`.

```
19.         self.cascade_1 = cv2.CascadeClassifier(cascade_1)
20.         self.cascade_2 = cv2.CascadeClassifier(cascade_2)
21.         self.cascade_3 = cv2.CascadeClassifier(cascade_3)
```

These three lines create the OpenCV cascade classifiers based on the three XML files, two for frontal face views and one for side profiles.

```
25.         self.haar_minSize = rospy.get_param("~haar_minSize", (20, 20))
26.         self.haar_maxSize = rospy.get_param("~haar_maxSize", (150, 150))
27.         self.haar_scaleFactor = rospy.get_param("~haar_scaleFactor", 1.3)
28.         self.haar_minNeighbors = rospy.get_param("~haar_minNeighbors", 1)
29.         self.haar_flags = rospy.get_param("~haar_flags",
    cv.CV_HAAR_DO_CANNY_PRUNING)
30.
```

The cascade classifiers require a number a parameters that determine their speed and the probability of correctly detecting a target. In particular, the `minSize` and `maxSize` parameters (specified in x and y pixel dimensions) set the smallest and largest target (faces in our case) that will be accepted. The `scaleFactor` parameter acts as a multiplier to change the image size as the detector runs from one scale to the next. The smaller this number (it must be > 1.0), the finer the scale pyramid used to scan for faces

but the longer it will take on each frame. You can read up on the other parameters in the OpenCV documentation.

```
32.         self.haar_params = dict(minSize = self.haar_minSize,
33.                                 maxSize = self.haar_maxSize,
34.                                 scaleFactor = self.haar_scaleFactor,
35.                                 minNeighbors = self.haar_minNeighbors,
36.                                 flags = self.haar_flags)
```

Here we stick all the parameters into a Python dictionary variable for easy reference later in the script.

```
49.     def process_image(self, cv_image):
50.         # Create a greyscale version of the image
51.         grey = cv2.cvtColor(cv_image, cv2.COLOR_BGR2GRAY)
52.
53.         # Equalize the histogram to reduce lighting effects
54.         grey = cv2.equalizeHist(grey)
```

Since the `FaceDetector` class extends the `ROS2OpenCV2` class, the `process_image()` function overrides the one defined in `ros2opencv2.py`. In this case, we begin by converting the image to greyscale. Many feature detection algorithms run on a greyscale version of the image and this includes the Haar cascade detector. We then equalize the histogram of the greyscale image. Histogram equalization is a standard technique for reducing the effects of changes in overall lighting.

```
56.         self.detect_box = self.detect_face(grey)
57.
58.         # Did we find one?
59.         if self.detect_box is not None:
60.             self.hits += 1
61.         else:
62.             self.misses += 1
63.
64.         # Keep tabs on the hit rate so far
65.         self.hit_rate = float(self.hits) / (self.hits + self.misses)
```

Here we send the pre-processed image to the `detect_face()` function which we will describe below. If a face is detected, the bounding box is returned to the variable `self.detect_box` which in turn is drawn on the image by the `ROS2OpenCV2` base class.

If a face is detected in this image frame, we increment the number of hits by 1, otherwise we add to the number of misses. The running hit rate is then updated accordingly.

```
70.     def detect_face(self, input_image):
71.         # First check one of the frontal templates
72.         if self.cascade_1:
73.             faces = self.cascade_1.detectMultiScale(input_image,
    **self.haar_params)
```

Here we start the heart of the script—the `detect_face()` function. We run the input image through the cascade detector using the first XML template. The `detectMultiScale()` function searches the image across multiple scales and returns any faces as a list of OpenCV rectangles in the form (x, y, w, h) where (x, y) are the coordinates of the upper left corner of the box and (w, h) are the width and height in pixels.

```
76.           if len(faces) == 0 and self.cascade_3:
77.               faces =
self.cascade_3.detectMultiScale(input_image,**self.haar_params)
78.
79.           # If that also fails, check a the other frontal template
80.           if len(faces) == 0 and self.cascade_2:
81.               faces = self.cascade_2.detectMultiScale(input_image,
**self.haar_params)
```

If a face is not detected using the first cascade, we try a second detector and then a third if necessary.

```
85.           if len(faces) > 0:
86.               face_box = faces[0]
```

If one or more faces are found, the `faces` variable will hold a list of face boxes (`cvRect`). We will only track the first face found so we set the `face_box` variable to `faces[0]`. If no faces are found in this frame, we set `face_box` to `None`. Either way, the result is returned to the calling function `process_image()`.

That completes the script. The `process_image()` and `detect_face()` functions are applied to every frame of the video. The result is that the detect box tracks your face as long as it can be found in the current frame. Don't forget that you can monitor the `/roi` topic to see the coordinates of the tracked face:

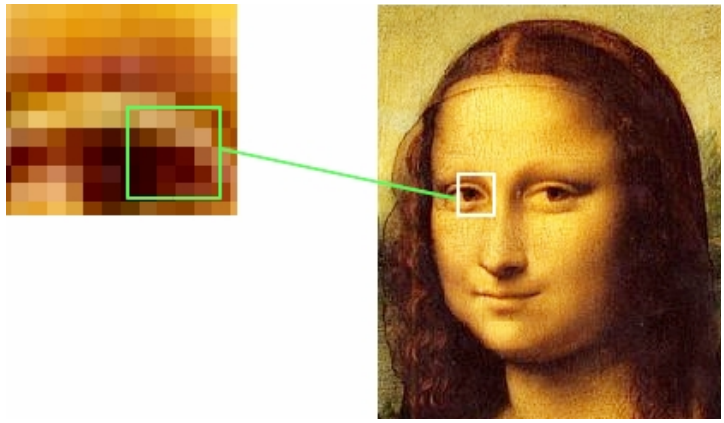
```
$ rostopic echo /roi
```

Tracking an object by running a detector over and over across the entire image is computationally expensive and can easily lose track of the object on any given frame. In the next two sections, we will show how to quickly detect and track a set of keypoints for any given region of an image. We will then combine the tracker with the face detector to make a better face tracker.

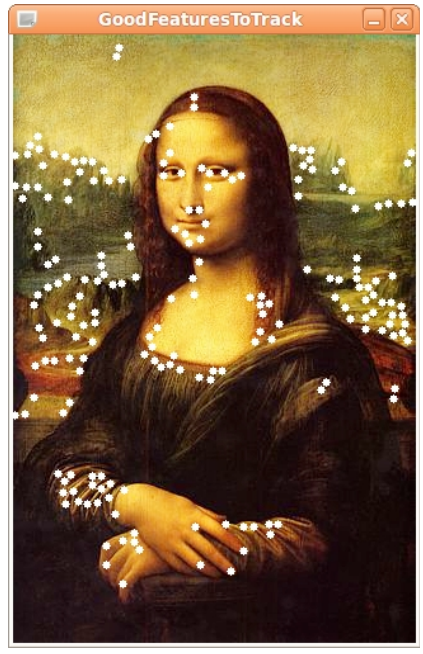
10.8.2 Keypoint Detection using `GoodFeaturesToTrack`

The Haar face detector scans an image for a specific type of object. A different strategy involves looking for smaller image features that are fairly easy to track from one frame to the next. These features are called **keypoints** or **interest points**. Keypoints tend to

be regions where there are large changes in intensity in more than one direction. Consider for example the images shown below:



The image on the left shows the pixels from the left eye region from the image on the right. The square on the left indicates the region where the intensity changes the most in all directions. The center of such a region is a keypoint of the image and it is likely to be re-detected in the same location of the face regardless of its orientation or scale.



OpenCV 2.4 includes a number of keypoint detectors including goodFeaturesToTrack(), cornerHarris() and SURF(). We will use `goodFeaturesToTrack()` for our programming example. The image on the right illustrates the keypoints returned by `goodFeaturesToTrack()`.

As you can see, keypoints are concentrated in areas where the intensity gradients are largest. Conversely, areas of the painting which are fairly homogeneous have few or no keypoints. (To reproduce this image or find the keypoints in other images, take a look at the Python program called `script_good_features.py` in the `rbx1_vision/scripts` directory.)

We are now ready to detect keypoints on a live video stream. Our ROS node is called `good_features.py` and it can be found in the `rbx1_vision/src/rbx1_vision` subdirectory. The corresponding launch file is `good_features.launch` in the launch

subdirectory. The launch file contains a number of parameters that affect the keypoints returned by `goodFeaturesToTrack()`:

- `maxCorners`: Puts an upper limit on how many keypoints are returned.
- `qualityLevel`: Reflects how strong a corner-like feature must be before it counts as a keypoint. Setting lower values returns more points.
- `minDistance`: The minimum number of pixels between keypoints.
- `blockSize`: The size of the neighborhood around a pixel used to compute whether or not there is a corner there.
- `useHarrisDetector`: Whether or not to use the original Harris corner detector or a minimum eigenvalue criterion.
- `k`: A free parameter for the Harris corner detector.

The `good_features.launch` file sets reasonable defaults for these parameters but try experimenting with them to see their effects.

To run the detector, first make sure the driver for your camera is up and running as described previously. Terminate the face detector launch file if you still have it running from the previous section, then run the command:

```
$ roslaunch rbx1_vision good_features.launch
```

When the video appears, draw a rectangle with your mouse around some object in the image. A rectangle should indicate the selected region and you should see a number of green dots indicating the keypoints found in that region by `goodFeaturesToTrack()`. Try drawing a box around other areas of the image and see if you can guess where the keypoints will appear. Note that we are not yet tracking these keypoints—we are simply computing them for whatever patch of the scene lies within your select box.

You will probably notice that some of the keypoints jiggle around a bit even on a stationary part of the scene. This is due to noise in the video. Strong keypoints are less susceptible to noise as you can see by drawing your test box around the corner of a table or other high contrast point.

Let's now look at the code.

Link to source: good_features.py

```
1.      #!/usr/bin/env python
2.
3.      import rospy
```

```
4.      import cv2
5.      import cv2.cv as cv
6.      from rbx1_vision.ros2opencv2 import ROS2OpenCV2
7.      import numpy as np
8.
9.      class GoodFeatures(ROS2OpenCV2):
10.         def __init__(self, node_name):
11.             super(GoodFeatures, self).__init__(node_name)
12.
13.             # Do we show text on the display?
14.             self.show_text = rospy.get_param("~show_text", True)
15.
16.             # How big should the feature points be (in pixels)?
17.             self.feature_size = rospy.get_param("~feature_size", 1)
18.
19.             # Good features parameters
20.             self.gf_maxCorners = rospy.get_param("~gf_maxCorners", 200)
21.             self.gf_qualityLevel = rospy.get_param("~gf_qualityLevel", 0.05)
22.             self.gf_minDistance = rospy.get_param("~gf_minDistance", 7)
23.             self.gf_blockSize = rospy.get_param("~gf_blockSize", 10)
24.             self.gf_useHarrisDetector =
rospy.get_param("~gf_useHarrisDetector", True)
25.             self.gf_k = rospy.get_param("~gf_k", 0.04)
26.
27.             # Store all parameters together for passing to the detector
28.             self.gf_params = dict(maxCorners = self.gf_maxCorners,
29.                                   qualityLevel = self.gf_qualityLevel,
30.                                   minDistance = self.gf_minDistance,
31.                                   blockSize = self.gf_blockSize,
32.                                   useHarrisDetector = self.gf_useHarrisDetector,
33.                                   k = self.gf_k)
34.
35.             # Initialize key variables
36.             self.keypoints = list()
37.             self.detect_box = None
38.             self.mask = None
39.
40.         def process_image(self, cv_image):
41.             # If the user has not selected a region, just return the image
42.             if not self.detect_box:
43.                 return cv_image
44.
45.             # Create a greyscale version of the image
46.             grey = cv2.cvtColor(cv_image, cv2.COLOR_BGR2GRAY)
47.
48.             # Equalize the histogram to reduce lighting effects
49.             grey = cv2.equalizeHist(grey)
50.
51.             # Get the good feature keypoints in the selected region
52.             keypoints = self.get_keypoints(grey, self.detect_box)
53.
54.             # If we have points, display them
55.             if keypoints is not None and len(keypoints) > 0:
56.                 for x, y in keypoints:
57.                     cv2.circle(self.marker_image, (x, y), self.feature_size,
(0, 255, 0, 0), cv.CV_FILLED, 8, 0)
58.
```

```
            # Process any special keyboard commands
            if 32 <= self.keystroke and self.keystroke < 128:
                cc = chr(self.keystroke).lower()
                if cc == 'c':
                    # Clear the current keypoints
                    keypoints = list()
                    self.detect_box = None

            return cv_image

        def get_keypoints(self, input_image, detect_box):
            # Initialize the mask with all black pixels
            self.mask = np.zeros_like(input_image)

            # Get the coordinates and dimensions of the detect_box
            try:
                x, y, w, h = detect_box
            except:
                return None

            # Set the selected rectangle within the mask to white
            self.mask[y:y+h, x:x+w] = 255

            # Compute the good feature keypoints within the selected region
            keypoints = list()
            kp = cv2.goodFeaturesToTrack(input_image, mask = self.mask,
**self.gf_params)
            if kp is not None and len(kp) > 0:
                for x, y in np.float32(kp).reshape(-1, 2):
                    keypoints.append((x, y))

            return keypoints

    if __name__ == '__main__':
        try:
            node_name = "good_features"
            GoodFeatures(node_name)
            rospy.spin()
        except KeyboardInterrupt:
            print "Shutting down the Good Features node."
            cv.DestroyAllWindows()
```

Overall, we see that the script has the same structure as the `face_detector.py` node. We initialize a `GoodFeatures` class that extends the `ROS2OpenCV2` class. We then define a `process_image()` function that does most the work. Let's examine the more important lines of the script:

```
        self.gf_maxCorners = rospy.get_param("~gf_maxCorners", 200)
        self.gf_qualityLevel = rospy.get_param("~gf_qualityLevel", 0.02)
        self.gf_minDistance = rospy.get_param("~gf_minDistance", 7)
        self.gf_blockSize = rospy.get_param("~gf_blockSize", 10)
        self.gf_useHarrisDetector =
    rospy.get_param("~gf_useHarrisDetector", True)
        self.gf_k = rospy.get_param("~gf_k", 0.04)
```

As with the Haar detector, the Good Features detector takes a number of parameters to fine tune its behavior. Probably the two most important parameters above are `qualityLevel` and `minDistance`. Smaller values for qualityLevel will result in a great number of feature points, but a number of them will be due to noise and not very consistent from one image frame to the next. Setting a higher value that is too high will yield only a few keypoints at the very strongest corners. Something on the order of 0.02 or so seems to strike a good balance for videos of natural scenes.

The `minDistance` parameter specifies the smallest distance in pixels we will allow between keypoints. The larger this value, the further apart keypoints must be resulting in fewer of them.

```
40.     def process_image(self, cv_image):
41.         # If the user has not selected a region, just return the image
42.         if not self.detect_box:
43.             return cv_image
44.
45.         # Create a greyscale version of the image
46.         grey = cv2.cvtColor(cv_image, cv2.COLOR_BGR2GRAY)
47.
48.         # Equalize the histogram to reduce lighting effects
49.         grey = cv2.equalizeHist(grey)
50.
51.         # Get the good feature keypoints in the selected region
52.         keypoints = self.get_keypoints(grey, self.detect_box)
```

As with the face detector node, we define a `process_image()` function which first converts the image to greyscale and equalizes the histogram. The resulting image is passed to the `get_keypoints()` function which does all the work of finding the Good Features.

```
69.     def get_keypoints(self, input_image, detect_box):
70.         # Initialize the mask with all black pixels
71.         self.mask = np.zeros_like(input_image)
72.
73.         # Get the coordinates and dimensions of the detect_box
74.         try:
75.             x, y, w, h = detect_box
76.         except:
77.             return None
78.
79.         # Set the selected rectangle within the mask to white
80.         self.mask[y:y+h, x:x+w] = 255
81.
82.         # Compute the good feature keypoints within the selected region
83.         keypoints = list()
84.         kp = cv2.goodFeaturesToTrack(input_image, mask = self.mask,
    **self.gf_params)
85.         if kp is not None and len(kp) > 0:
86.             for x, y in np.float32(kp).reshape(-1, 2):
87.                 keypoints.append((x, y))
88.
89.         return keypoints
```

The `get_keypoints()` function implements OpenCV's GoodFeaturesToTrack detector. Since we only want the keypoints within the box selected by the user (`detect_box`) we mask the image with the box by starting with a mask of all zeros (black) and then filling in the detect box with all white pixels (255). The output from the `cv2.goodFeaturesToTrack()` function is a vector of keypoint coordinates. So we use a little numpy reshaping to turn it into a Python list of (x,y) pairs. The resulting list is returned to the `process_image()` function where the points are drawn on the image.

10.8.3 Tracking Keypoints using Optical Flow

Now that we can detect keypoints in an image, we can use them to track the underlying object from one video frame to the next by using OpenCV's Lucas-Kanade optical flow function `calcOpticalFlowPyrLK()`. A detailed explanation of the Lucas-Kanade method can be found on Wikipedia. The basic idea is as follows.

We begin with the current image frame and the set of keypoints we have already extracted. Each keypoint has a location (x and y coordinates) and a neighborhood of surrounding image pixels. In the next image frame, the Lucas-Kanade algorithm uses a least squares method to solve for a small constant-velocity transformation that maps a given neighborhood of pixels from the first frame to the next. If the least squares error for a given neighborhood does not exceed some threshold, we assume it is the same neighborhood as in the first frame and we assign it the same keypoint to that location; otherwise the keypoint is discarded. Note that we are *not* extracting new keypoints in subsequent frames. Instead, `calcOpticalFlowPyrLK()` calculates new positions for the original keypoints. In this manner, we can extract the keypoints in the first frame, then follow them from frame to frame as the underlying object or camera moves over time.

After a number of frames, tracking will degrade for two reasons: keypoints will be dropped when the tracking error is too high between frames, and nearby keypoints will take the place of those in the original set as the algorithms makes errors in predictions. We will look at ways to overcome these limitations in later sections.

Our new node, `lk_tracker.py`, can be found in the `rbx1_vision/src/rbx1_vision` directory and combines our earlier keypoint detector (using `goodFeaturesToTrack()`) with this optical flow tracking. Terminate the good features launch file if you still have it running, then run:

```
$ roslaunch rbx1_vision lk_tracker.launch
```

When the video window appears, draw a rectangle with your mouse around an object of interest. As in the previous section, keypoints will appear over the image as green dots. Now try moving the object or the camera and the keypoints should follow the object. In particular, try drawing a box around your face. You should see keypoints attach to varies parts of your face. Now move your head around and the keypoints should move with it. Notice how much faster the CPS value is compared to running the Haar face detector. On my machine, tracking keypoints using the LK method is fully twice as fast as using the Haar face detector. It is also much more reliable in that it continues to track the face keypoints under a greater range of movement. You can best see this by turning on "night mode" (hit the "n" key when the video window is in the foreground.)

Remember that the base class `ROS2OpenCV2` publishes the bounding box around the tracked points on the `/roi` topic. So if you run the command:

```
$ rostopic echo /roi
```

while using the `lk_tracker` node, you should see the ROI move as the points move. This means that if you have another node that needs to follow the location of the tracked points, it only has to subscribe to the `/roi` topic to follow them in real time.

Now let's look at the code:

Link to source: lk_tracker.py

```
1.      #!/usr/bin/env python
2.
3.      import rospy
4.      import cv2
5.      import cv2.cv as cv
6.      import numpy as np
7.      from rbx1_vision.good_features import GoodFeatures
8.
9.      class LKTracker(GoodFeatures):
10.         def __init__(self, node_name)
11.             super(LKTracker, self).__init__(node_name)
12.
13.             self.show_text = rospy.get_param("~show_text", True)
14.             self.feature_size = rospy.get_param("~feature_size", 1)
15.
16.             # LK parameters
17.             self.lk_winSize = rospy.get_param("~lk_winSize", (10, 10))
18.             self.lk_maxLevel = rospy.get_param("~lk_maxLevel", 2)
19.             self.lk_criteria = rospy.get_param("~lk_criteria",
(cv2.TERM_CRITERIA_EPS | cv2.TERM_CRITERIA_COUNT, 20, 0.01))
20.             self.lk_derivLambda = rospy.get_param("~lk_derivLambda", 0.1)
21.
22.             self.lk_params = dict( winSize  = self.lk_winSize,
23.                       maxLevel = self.lk_maxLevel,
24.                       criteria = self.lk_criteria,
25.                       derivLambda = self.lk_derivLambda )
26.
```

```
27.             self.detect_interval = 1
28.             self.keypoints = list()
29.
30.             self.detect_box = None
31.             self.track_box = None
32.             self.mask = None
33.             self.grey = None
34.             self.prev_grey = None
35.
36.         def process_image(self, cv_image):
37.             # If we don't yet have a detection box (drawn by the user
38.             # with the mouse), keep waiting
39.             if self.detect_box is None:
40.                 return cv_image
41.
42.             # Create a greyscale version of the image
43.             self.grey = cv2.cvtColor(cv_image, cv2.COLOR_BGR2GRAY)
44.
45.             # Equalize the grey histogram to minimize lighting effects
46.             self.grey = cv2.equalizeHist(self.grey)
47.
48.             # If we haven't yet started tracking, set the track box to the
49.             # detect box and extract the keypoints within it
50.             if self.track_box is None or not
self.is_rect_nonzero(self.track_box):
51.                 self.track_box = self.detect_box
52.                 self.keypoints = list()
53.                 self.keypoints = self.get_keypoints(self.grey,
self.track_box)
54.
55.             else:
56.                 if self.prev_grey is None:
57.                     self.prev_grey = self.grey
58.
59.                 # Now that have keypoints, track them to the next frame
60.                 # using optical flow
61.                 self.track_box = self.track_keypoints(self.grey,
self.prev_grey)
62.
63.             # Process any special keyboard commands for this module
64.             if 32 <= self.keystroke and self.keystroke < 128:
65.                 cc = chr(self.keystroke).lower()
66.                 if cc == 'c':
67.                     # Clear the current keypoints
68.                     self.keypoints = list()
69.                     self.track_box = None
70.                     self.detect_box = None
71.                     self.classifier_initialized = True
72.
73.             self.prev_grey = self.grey
74.
75.             return cv_image
76.
77.         def track_keypoints(self, grey, prev_grey):
78.             try:
79.                 # We are tracking points between the previous frame and the
80.                 # current frame
```

```
                img0, img1 = prev_grey, grey

                  # Reshape the current keypoints into a numpy array required
                  # by calcOpticalFlowPyrLK()
                  p0 = np.float32([p for p in self.keypoints]).reshape(-1, 1,
2)

                  # Calculate the optical flow from the previous frame
                  # tp the current frame
                  p1, st, err = cv2.calcOpticalFlowPyrLK(img0, img1, p0,
None, **self.lk_params)

                  # Do the reverse calculation: from the current frame
                  # to the previous frame
                  p0r, st, err = cv2.calcOpticalFlowPyrLK(img1, img0, p1,
None, **self.   lk_params)

                  # Compute the distance between corresponding points
                  # in the two flows
                  d = abs(p0-p0r).reshape(-1, 2).max(-1)

                  # If the distance between pairs of points is < 1 pixel, set
                  # a value in the "good" array to True, otherwise False
                  good = d < 1

                  # Initialize a list to hold new keypoints
                  new_keypoints = list()

                  # Cycle through all current and new keypoints and only keep
                  # those that satisfy the "good" condition above
                  for (x, y), good_flag in zip(p1.reshape(-1, 2), good):
                      if not good_flag:
                          continue
                      new_keypoints.append((x, y))

                      # Draw the keypoint on the image
                      cv2.circle(self.marker_image, (x, y),
self.feature_size, (0, 255, 0, 0), cv.CV_FILLED, 8, 0)

                # Set the global keypoint list to the new list
                self.keypoints = new_keypoints

                # If we have >6 points, find the best ellipse around them
                if len(self.keypoints) > 6:
                    self.keypoints_matrix = cv.CreateMat(1,
len(self.keypoints), cv.CV_32SC2)
                i = 0
                for p in self.keypoints:
                    cv.Set2D(self.keypoints_matrix, 0, i, (int(p[0]),
int(p[1])))
                     i = i + 1
                track_box = cv.FitEllipse2(self.keypoints_matrix)
            else:
                # Otherwise, find the best fitting rectangle
                track_box = cv2.boundingRect(self.keypoints_matrix)
        except:
            track_box = None
```

```
132.
133.         return track_box
134.
135.
136.     if __name__ == '__main__':
137.         try:
138.             node_name = "lk_tracker"
139.             LKTracker(node_name)
140.             rospy.spin()
141.         except KeyboardInterrupt:
142.             print "Shutting down LK Tracking node."
143.             cv.DestroyAllWindows()
```

Let's look at the key lines of the script.

```
7.  from rbx1_vision.good_features import GoodFeatures
8.
9.  class LKTracker(GoodFeatures):
10.     def __init__(self, node_name):
11.         super(LKTracker, self).__init__(node_name)
```

The overall structure of the script is once again similar to the `face_detector.py` node. However, this time we import `good_features.py` and define the `LKTracker` class as an extension of the `GoodFeatures` class rather than the `ROS2OpenCV2`. Why? Because the keypoints we will track are precisely those we obtained from the `GoodFeatures` class in the previous section. And since the `GoodFeatures` class itself extends the `ROS2Opencv2` class, we are covered.

```
36.     def process_image(self, cv_image):
```

...

```
53.             self.keypoints = self.get_keypoints(self.grey,
    self.track_box)
```

...

```
61.             self.track_box = self.track_keypoints(self.grey,
    self.prev_grey)
```

The `process_image()` function is very similar to the one we used in the `good_features.py` script. The key lines are `53` and `61` above. In line `53` we are using the `get_keypoints()` function from the `GoodFeatures` class to get the initial keypoints. And in line `61`, we track those keypoints using the new `track_keypoints()` function which we will now describe.

```
77.     def track_keypoints(self, grey, prev_grey):
78.         try:
79.             # We are tracking points between the previous frame and the
80.             # current frame
81.             img0, img1 = prev_grey, grey
82.
83.             # Reshape the current keypoints into a numpy array required
```

```
84.             # by calcOpticalFlowPyrLK()
85.             p0 = np.float32([p for p in self.keypoints]).reshape(-1, 1,
    2)
```

To track the keypoints, we begin by storing the previous greyscale image and the current greyscale image in a couple of variables. We then store the current keypoints using a numpy array format required by the `calcOpticalFlowPyrLK()` function.

```
89.         p1, st, err = cv2.calcOpticalFlowPyrLK(img0, img1, p0, None,
    **self.lk_params)
```

In this line we use the OpenCV `calcOpticalFlowPyrLK()` function to predict the next set of keypoints from the current keypoints and the two greyscale images.

```
93.         p0r, st, err = cv2.calcOpticalFlowPyrLK(img1, img0, p1, None,
    **self.   lk_params)
```

And in this line, we make the *reverse* calculation: here we predict the previous points from the future points we just computed. This allows us to do a consistency check since we can compare the actual previous points (the keypoints we started with) with these reverse-predicted points.

```
97.         d = abs(p0-p0r).reshape(-1, 2).max(-1)
```

Next we compute the distances between pairs of reverse-predicted points (p0r) and our original keypoints (p0). The result, `d`, is an array of these distances. (Python can seemly awfully compact sometimes.)

```
101.         good = d < 1
```

And here we define a new array (`good`) that is a set of `True` or `False` values depending on whether or not the distance between a pair of points is less than 1 pixel.

```
108.         for (x, y), good_flag in zip(p1.reshape(-1, 2), good):
109.             if not good_flag:
110.                 continue
111.             new_keypoints.append((x, y))
```

Finally, we drop any keypoints that are more than 1 pixel away from their reverse-predicted counterpart.

```
117.                 self.keypoints = new_keypoints
```

The result becomes our new global set of keypoints that we send through the next tracking cycle.

10.8.4 Building a Better Face Tracker

We now have the ingredients we need to improve on our original face detector. Recall that the `face_detector.py` node attempts to detect a face over and over on every frame. This is not only CPU-intensive, but it can also fail to detect a face at all fairly often. A better strategy is to first detect the face, then use `goodFeaturesToTrack()` to extract keypoints from the face region and then use `calcOpticalFlowPyrLK()` to track those features from frame to frame. In this way, detection is only done once to originally acquire the face region.

Our processing pipeline looks like this:

`detect_face() → get_keypoints() → track_keypoints()`

In terms of the nodes we have developed so far, the pipeline becomes:

`face_detector.py() → good_features.py() → lk_tracker.py()`

Our new node, `face_tracker.py` implements this pipeline. To try it out, make sure you have launched the driver to your camera, then run:

```
$ roslaunch rbx1_vision face_tracker.launch
```

If you move your face into view of the camera, the Haar face detector should find it. After the initial detection, the keypoints are computed over the face region and then tracked through subsequent frames using optical flow. To clear the current keypoints and force a re-detection of the face, hit the "`c`" key when the video window is in the foreground.

Let's now look at the code.

Link to source: face_tracker.py

```
1.      #!/usr/bin/env python
2.
3.      import rospy
4.      import cv2
5.      import cv2.cv as cv
6.      import numpy as np
7.
8.      from rbx1_vision.face_detector import FaceDetector
9.      from rbx1_vision.lk_tracker import LKTracker
10.
11.     class FaceTracker(FaceDetector, LKTracker):
12.         def __init__(self, node_name):
13.             super(FaceTracker, self).__init__(node_name)
14.
15.             self.n_faces = rospy.get_param("~n_faces", 1)
16.             self.show_text = rospy.get_param("~show_text", True)
17.             self.feature_size = rospy.get_param("~feature_size", 1)
18.
```

```
            self.keypoints = list()
            self.detect_box = None
            self.track_box = None

            self.grey = None
            self.prev_grey = None

        def process_image(self, cv_image):
            # Create a greyscale version of the image
            self.grey = cv2.cvtColor(cv_image, cv2.COLOR_BGR2GRAY)

            # Equalize the grey histogram to minimize lighting effects
            self.grey = cv2.equalizeHist(self.grey)

            # STEP 1: Detect the face if we haven't already
            if self.detect_box is None:
                self.detect_box = self.detect_face(self.grey)

            else:
                # Step 2: If we aren't yet tracking keypoints, get them now
                if self.track_box is None or not
self.is_rect_nonzero(self.track_box):
                    self.track_box = self.detect_box
                    self.keypoints = self.get_keypoints(self.grey,
self.track_box)

                # Step 3: If we have keypoints, track them using optical flow
                if len(self.keypoints) > 0:
                    # Store a copy of the current grey image used for LK
tracking
                    if self.prev_grey is None:
                        self.prev_grey = self.grey

                    self.track_box = self.track_keypoints(self.grey,
self.prev_grey)
                else:
                    # We have lost all keypoints so re-detect the face
                    self.detect_box = None

            # Process any special keyboard commands for this module
            if 32 <= self.keystroke and self.keystroke < 128:
                cc = chr(self.keystroke).lower()
                if cc == 'c':
                    self.keypoints = list()
                    self.track_box = None
                    self.detect_box = None

            # Set store a copy of the current image used for LK tracking
            self.prev_grey = self.grey

            return cv_image

    if __name__ == '__main__':
        try:
            node_name = "face_tracker"
            FaceTracker(node_name)
            rospy.spin()
```

```
72.         except KeyboardInterrupt:
73.             print "Shutting down face tracker node."
74.             cv.DestroyAllWindows()
```

The `face_tracker` node essentially combines two nodes we have already developed: the `face_detector` node and the `lk_tracker` node. The `lk_tracker` node in turn depends on the `good_features` node. The following breakdown explains how we combine these Python classes.

```
8.  from rbx1_vision.face_detector import FaceDector
9.  from rbx1_vision.lk_tracker import LKTracker
10.
11. class FaceTracker(FaceDetector, LKTracker):
12.     def __init__(self, node_name):
13.         super(FaceTracker, self).__init__(node_name)
```

To use the `face_detector` and `lk_tracker` code we developed earlier, we first have to import their classes, `FaceDetector` and `LKTracker`. We then define our new `FaceTracker` class as extending both classes. In Python this is called multiple inheritance. As before, we then use the `super()` function to initialize our new class which also takes care of initializing the parent classes.

```
26.     def process_image(self, cv_image):
27.         # Create a greyscale version of the image
28.         self.grey = cv2.cvtColor(cv_image, cv2.COLOR_BGR2GRAY)
29.
30.         # Equalize the grey histogram to minimize lighting effects
31.         self.grey = cv2.equalizeHist(self.grey)
```

As we did with our other nodes, we begin the `process_image()` function by converting the image to greyscale and equalizing the histogram to minimize lighting effects.

```
33.         # STEP 1: Detect the face if we haven't already
34.         if self.detect_box is None:
35.             self.detect_box = self.detect_face(self.grey)
```

Step 1 is to detect the face is have not already. The `detect_face()` function comes from the `FaceDetector` class we imported.

```
38.         # STEP 2: If we aren't yet tracking keypoints, get them now
39.         if self.track_box is None or not
    self.is_rect_nonzero(self.track_box):
40.             self.track_box = self.detect_box
41.             self.keypoints = self.get_keypoints(self.grey, self.track_box)
```

Once we have detected a face, Step 2 is to get the keypoints from the face region using the `get_keypoints()` function we imported from the `LKTracker` class, which in turn actually gets the function from the `GoodFeatures` class that *it* imports.

```
43.         # STEP 3: If we have keypoints, track them using optical flow
44.         if len(self.keypoints) > 0:
45.             # Store a copy of the current grey image used for LK tracking
46.             if self.prev_grey is None:
47.                 self.prev_grey = self.grey
48.
49.             self.track_box = self.track_keypoints(self.grey,
    self.prev_grey)
```

Once we have the keypoints, Step 3 starts tracking them using the `track_keypoints()` function that we imported from the `LKTracker` class.

```
1.  else:
2.      # We have lost all keypoints so re-detect the face
3.      self.detect_box = None
```

If during tracking the number of keypoints dwindles to zero, we set the detect box to None so that we can re-detect the face in Step 1.

In the end, you can see that the overall script is essentially just a combination of our earlier nodes.

10.8.5 Dynamically Adding and Dropping Keypoints

If you play with the face tracker for a little bit, you will notice that the keypoints can drift onto other objects besides your face. You will also notice that the number of keypoints shrinks over time as the optical flow tracker drops them due to a low tracking score.

We can easily add new keypoints and drop bad ones during the tracking process. To add keypoints, we run `goodFeaturesToTrack()` every once in awhile over the region we are tracking. To drop keypoints, we can run a simple statistical clustering test on the collection of keypoints and remove the outliers.

The node `face_tracker2.py` incorporates these improvements. You can try it using the following command:

```
$ roslaunch rbx1_vision face_tracker2.launch
```

You should now see improved tracking of your face as keypoints are added and dropped to reflect the movements of your head. To actually see the points that are added and dropped, as well as the region around the face from which new keypoints are drawn, hit the "`d`" key over the image window. The expanded keypoint region is shown by a yellow box. Added keypoints will flash briefly in light blue and dropped points will flash briefly in red before they disappear. Hit the "`d`" key again to turn off the display. You can also hit the "`c`" key at any time to clear the current keypoints and force a re-detection of the face.

The code for `face_tracker2.py` is nearly the same as the first `face_tracker.py` script so we won't describe it in detail again. The full source can be found here:

Link to source: `face_tracker2.py`

The two new functions are `add_keypoints()` and `drop_keypoints()` which should be fairly self-explanatory from the comments in the code. However, it is worth briefly describing the new parameters that control when points are dropped and added. These can be found in the launch file `face_tracker2.launch` in the `rbx1_vision/launch` directory. Let's look at those parameters now. The default values are in parentheses:

- `use_depth_for_tracking:(False)` If you are using a depth camera, setting this value to `True` will drop keypoints that fall too far away from the face plane.
- `min_keypoints:(20)` The minimum number of keypoints before we will add new ones.
- `abs_min_keypoints:(6)` The absolute minimum number of keypoints before we consider the face lost and try to re-detect it.
- `add_keypoint_distance:(10)` A new keypoint must be at least this distance (in pixels) from any existing keypoint.
- `std_err_xy:(2.5)` The standard error (in pixels) for determining whether or not a keypoint is an outlier.
- `pct_err_z:(0.42)` The depth threshold (as a percent) that determines when we drop a keypoint for falling too far off the face plane.
- `max_mse:(10000)` The maximum total mean squared error in the current feature cluster before we start over and re-detect the face.
- `expand_roi:(1.02)` When looking for new keypoints, the expansion factor to grow the ROI on each cycle.
- `add_keypoints_interval:(1)` How often do we attempt to add new keypoints. A value of 1 means every frame, 2 every other frame and so on.
- `drop_keypoints_interval:(1)` How often do we attempt to drop keypoints. A value of 1 means every frame, 2 every other frame and so on.

Most of the defaults should work fairly well but of course feel free to try different values.

10.8.6 Color Blob Tracking (CamShift)

So far we have not made any use of color information to track the object of interest. OpenCV includes the CamShift filter which allows us to track a selected region of the image based on the color histogram of that region. For an excellent explanation of how the CamShift filter works, be sure to check out Robin Hewitt's article on How OpenCV's Face Tracker Works. In short, the CamShift filter scans successive frames of the video stream and assigns each pixel a probability of belonging to the original color histogram. The collection of pixels with the highest probability of "belonging" becomes the new target region to be tracked.

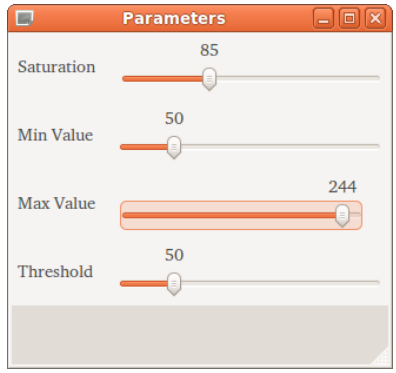
Before we look at the code, you can try it out as follows. Tracking works best with brightly colored objects. So first find an object such as a plastic ball that is fairly uniform in color and stands out from any background colors.

Next be sure to run the appropriate camera driver for the camera you are using. Then launch the CamShift node with the following command:

```
$ roslaunch rbx1_vision camshift.launch
```

When the video window appears, hold the target object in front of the camera and draw a rectangle around it. The CamShift node will immediately begin to follow the object as best it can based on the color histogram computed from the selected region. Notice how the tracked region actually fills out the object even if you only select a smaller piece of it. This is because the CamShift algorithm is adaptively matching a *range* of colors in the selected region, not just a single RGB value.

Two other windows will be present on the screen. The first is a group of slider controls that look like this:



These controls determine the selectivity of the CamShift filter. For brightly colored objects like a green tennis ball, the default values should work fairly well. However, for natural colors like faces, you may have to turn down the **Saturation** and **Min Value** settings and tweak the **Threshold** parameter slightly. Once you find a group of settings that work for your camera, you can set them in the `camshift.launch` file.

The second window shows the "back projection" of the histogram probabilities back onto the image. The result is a greyscale image where white pixels represent a high probability of belonging to the histogram and grey or black pixels represent lower probabilities. It is useful to have the back projection window visible when adjusting the slider controls: the goal is to have mostly white pixels over the target and black elsewhere.

The following video demonstrates the CamShift filter in action:

http://www.youtube.com/watch?v=rDTun7A6HO8&feature=plcp

Let's now look at the code.

Link to source: camshift.py

```
1.  #!/usr/bin/env python
2.
3.  import rospy
4.  import cv2
5.  from cv2 import cv as cv
6.  from rbx1_vision.ros2opencv2 import ROS2OpenCV2
7.  from std_msgs.msg import String
8.  from sensor_msgs.msg import Image
9.  import numpy as np
10.
11. class CamShiftNode(ROS2OpenCV2):
12.     def __init__(self, node_name):
13.         ROS2OpenCV2.__init__(self, node_name)
14.
15.         self.node_name = node_name
16.
17.         # The minimum saturation of the tracked color in HSV space,
18.         # as well as the min and max value (the V in HSV) and a
19.         # threshold on the backprojection probability image.
20.         self.smin = rospy.get_param("~smin", 85)
21.         self.vmin = rospy.get_param("~vmin", 50)
22.         self.vmax = rospy.get_param("~vmax", 254)
23.         self.threshold = rospy.get_param("~threshold", 50)
24.
25.         # Create a number of windows for displaying the histogram,
26.         # parameters controls, and backprojection image
27.         cv.NamedWindow("Histogram", cv.CV_WINDOW_NORMAL)
28.         cv.MoveWindow("Histogram", 700, 50)
29.         cv.NamedWindow("Parameters", 0)
30.         cv.MoveWindow("Parameters", 700, 325)
31.         cv.NamedWindow("Backproject", 0)
32.         cv.MoveWindow("Backproject", 700, 600)
```

```
33.
34.             # Create the slider controls for saturation, value and threshold
35.             cv.CreateTrackbar("Saturation", "Parameters", self.smin, 255,
self.set_smin)
36.             cv.CreateTrackbar("Min Value", "Parameters", self.vmin, 255,
self.set_vmin)
37.             cv.CreateTrackbar("Max Value", "Parameters", self.vmax, 255,
self.set_vmax)
38.             cv.CreateTrackbar("Threshold", "Parameters", self.threshold, 255,
self.set_threshold)
39.
40.             # Initialize a number of variables
41.             self.hist = None
42.             self.track_window = None
43.             self.show_backproj = False
44.
45.         # These are the callbacks for the slider controls
46.         def set_smin(self, pos):
47.             self.smin = pos
48.
49.         def set_vmin(self, pos):
50.             self.vmin = pos
51.
52.         def set_vmax(self, pos):
53.            self.vmax = pos
54.
55.         def set_threshold(self, pos):
56.             self.threshold = pos
57.
58.         # The main processing function computes the histogram and
backprojection
59.         def process_image(self, cv_image):
60.             # First blue the image
61.             frame = cv2.blur(cv_image, (5, 5))
62.
63.             # Convert from RGB to HSV spave
64.             hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
65.
66.             # Create a mask using the current saturation and value parameters
67.             mask = cv2.inRange(hsv, np.array((0., self.smin, self.vmin)),
np.array((180., 255., self.vmax)))
68.
69.             # If the user is making a selection with the mouse,
70.             # calculate a new histogram to track
71.             if self.selection is not None:
72.                 x0, y0, w, h = self.selection
73.                 x1 = x0 + w
74.                 y1 = y0 + h
75.                 self.track_window = (x0, y0, x1, y1)
76.                 hsv_roi = hsv[y0:y1, x0:x1]
77.                 mask_roi = mask[y0:y1, x0:x1]
78.                 self.hist = cv2.calcHist( [hsv_roi], [0], mask_roi, [16], [0,
180] )
79.                 cv2.normalize(self.hist, self.hist, 0, 255, cv2.NORM_MINMAX);
80.                 self.hist = self.hist.reshape(-1)
81.                 self.show_hist()
82.
```

```
            if self.detect_box is not None:
                self.selection = None

            # If we have a histogram, tracking it with CamShift
            if self.hist is not None:
                # Compute the backprojection from the histogram
                backproject = cv2.calcBackProject([hsv], [0], self.hist, [0,
180], 1)

                # Mask the backprojection with the mask created earlier
                backproject &= mask

                # Threshold the backprojection
                ret, backproject = cv2.threshold(backproject, self.threshold,
255, cv.CV_THRESH_TOZERO)

                x, y, w, h = self.track_window
                if self.track_window is None or w <= 0 or h <=0:
                    self.track_window = 0, 0, self.frame_width - 1,
self.frame_height - 1

            # Set the criteria for the CamShift algorithm
            term_crit = ( cv2.TERM_CRITERIA_EPS | cv2.TERM_CRITERIA_COUNT,
10, 1 )

            # Run the CamShift algorithm
            self.track_box, self.track_window = cv2.CamShift(backproject,
self.track_window, term_crit)

            # Display the resulting backprojection
            cv2.imshow("Backproject", backproject)

            return cv_image

        def show_hist(self):
            bin_count = self.hist.shape[0]
            bin_w = 24
            img = np.zeros((256, bin_count*bin_w, 3), np.uint8)
            for i in xrange(bin_count):
                h = int(self.hist[i])
                cv2.rectangle(img, (i*bin_w+2, 255), ((i+1)*bin_w-2, 255-h),
(int(180.0*i/bin_count), 255, 255), -1)
            img = cv2.cvtColor(img, cv2.COLOR_HSV2BGR)
            cv2.imshow('Histogram', img)


        def hue_histogram_as_image(self, hist):
            """ Returns a nice representation of a hue histogram """
            histimg_hsv = cv.CreateImage((320, 200), 8, 3)

            mybins = cv.CloneMatND(hist.bins)
            cv.Log(mybins, mybins)
            (_, hi, _, _) = cv.MinMaxLoc(mybins)
             cv.ConvertScale(mybins, mybins, 255. / hi)

            w,h = cv.GetSize(histimg_hsv)
            hdims = cv.GetDims(mybins)[0]
```

```
134.             for x in range(w):
135.                 xh = (180 * x) / (w - 1)  # hue sweeps from 0-180 across the
image
136.                 val = int(mybins[int(hdims * x / w)] * h / 255)
137.                 cv2.rectangle(histimg_hsv, (x, 0), (x, h-val), (xh,255,64),
-1)
138.                 cv2.rectangle(histimg_hsv, (x, h-val), (x, h), (xh,255,255),
-1)
139.
140.             histimg = cv2.cvtColor(histimg_hsv, cv.CV_HSV2BGR)
141.
142.             return histimg
143.
144.
145.     if __name__ == '__main__':
146.         try:
147.             node_name = "camshift"
148.             CamShiftNode(node_name)
149.             try:
150.                 rospy.init_node(node_name)
151.             except:
152.                 pass
153.             rospy.spin()
154.         except KeyboardInterrupt:
155.             print "Shutting down vision node."
156.             cv.DestroyAllWindows()
```

Let's look at the key lines in the script.

```
20.             self .smin = rospy.get_param("~smin", 85)
21.             self .vmin = rospy.get_param("~vmin", 50)
22.             self .vmax = rospy.get_param("~vmax", 254)
23.             self .threshold = rospy.get_param("~threshold", 50)
```

These are the parameters that control the color sensitivity of the CamShift algorithm. The algorithm will not work at all without setting these to the right values for your camera. The defaults should be close but you will want to play with the slider controls (which appear after launching the program) to find good values for your setup. Once you are satisfied with your numbers, you can enter them in the launch file to override the defaults.

The `smin` value controls the minimum *saturation* in the HSV image (Hue, Saturation, Value). It is a measure of the "richness" of a color. The `vmin` and `vmax` parameters determine the minimum and maximum *value* (brightness) a color needs to have. Finally, the `threshold` parameter is applied after the backprojection is computed to filter out low-probability pixels from the result.

```
35. cv.CreateTrackbar("Saturation", "Parameters", self.smin, 255,
    self.set_smin)
36. cv.CreateTrackbar("Min Value", "Parameters", self.vmin, 255,
    self.set_vmin)
```

```
37. cv.CreateTrackbar("Max Value", "Parameters", self.vmax, 255,
    self.set_vmax)
38. cv.CreateTrackbar("Threshold", "Parameters", self.threshold, 255,
    self.set_threshold)
```

Here we use OpenCV's trackbar function to create slider controls on the "Parameters" window. The last three arguments of the `CreateTrackbar()` function specify the minimum, maximum and default values for each trackbar.

```
59.     def process_image(self, cv_image):
60.         # First blue the image
61.         frame = cv2.blur(cv_image, (5, 5))
62.
63.         # Convert from RGB to HSV spave
64.         hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
```

The main processing loop begins by blurring the image, then converting it from blue-green-red (BGR) to hue-saturation-value (HSV). Blurring helps wash out some of the color noise in typical video images. Working in HSV space is a common practice when processing color images. In particular, the hue dimension maps nicely into what we (i.e. humans) consider different colors such as red, orange, yellow, green, blue, etc. The saturation dimension maps into how "rich" versus "washed out" a color appears to us, and the value dimension maps into how bright the color appears.

```
67.         mask = cv2.inRange(hsv, np.array((0., self.smin, self.vmin)),
    np.array((180., 255., self.vmax)))
```

The OpenCV `inRange()` function turns our saturation and value limits into a mask so that we process only those pixels that fall within our color parameters. Note that we are not filtering on hue—so we are still accepting any color at this point. Instead, we are selecting only those colors that have a fairly high saturation and value.

```
71.         if self.selection is not None:
72.             x0, y0, w, h = self.selection
73.             x1 = x0 + w
74.             y1 = y0 + h
75.             self.track_window = (x0, y0, x1, y1)
76.             hsv_roi = hsv[y0:y1, x0:x1]
77.             mask_roi = mask[y0:y1, x0:x1]
```

In this block we take a selection from the user (made with the mouse) and turn it into a region of interest for computing a color histogram and a mask.

```
78.             self.hist = cv2.calcHist( [hsv_roi], [0], mask_roi, [16], [0,
    180] )
79.             cv2.normalize(self.hist, self.hist, 0, 255, cv2.NORM_MINMAX);
80.             self.hist = self.hist.reshape(-1)
81.             self.show_hist()
```

Here we use the OpenCV `calcHist()` function to calculate a histogram over the hues in the selected region. Note that the region is also masked with `mask_roi`. We then normalize the histogram so that the maximum value is 255. This allows us to display the result as an 8-bit color image which is done with the last two lines using the helper function `show_hist()` defined later in the script.

```
89.             backproject = cv2.calcBackProject([hsv], [0], self.hist, [0,
    180], 1)
90.
91.             # Mask the backprojection with the mask created earlier
92.             backproject &= mask
93.
94.             # Threshold the backprojection
95.             ret, backproject = cv2.threshold(backproject, self.threshold,
    255, cv.CV_THRESH_TOZERO)
```

Once we have a histogram to track, we use the OpenCV `calcBackProject()` function to assign a probability to each pixel in the image of belonging to the histogram. We then mask the probabilities with our earlier mask and threshold it to eliminate low probability pixels.

```
102.         term_crit = ( cv2.TERM_CRITERIA_EPS | cv2.TERM_CRITERIA_COUNT,
    10, 1 )
103.
104.         # Run the CamShift algorithm
105.         self.track_box, self.track_window = cv2.CamShift(backproject,
    self.track_window, term_crit)
```

With the masked and thresholded backprojection in hand, we can finally run the `CamShift` algorithm itself which converts the probabilities into a new location of the track window.

The whole process is repeated on every frame (without any further selection from the user of course) and the track window follows the pixels that have the highest probability of belonging to the original histogram. If you find your target object is not being followed very well, start with a brightly colored ball or other uniformly colored object. You will also probably have to tweak the saturation and value slider controls to get the result you want.

10.9 OpenNI and Skeleton Tracking

Perhaps the earliest and best known robotics application using a depth camera is skeleton tracking. The ROS openni_tracker package can use the depth data from a Kinect or Asus Xtion to track the joint positions of a person standing in front of the camera. Using this data, one can program a robot to follow gesture commands signaled by the user. One example of how to do this using ROS and Python can be found in the pi_tracker package.

While we won't go into a lot of detail on the use of skeleton tracking, let's at least take a look at the basics.

10.9.1 Checking your OpenNI installation for Hydro

At the time of this writing, the OpenNI skeleton tracker was broken on some installations of ROS Hydro. To test your installation, plug in your depth camera (and apply power if you are using a Kinect) and run the following command:

```
$ rosrun openni_tracker openni_tracker
```

If you see an error similar to the following:

```
[ERROR] [1387637146.298760838]: Find user generator failed: This
operation is invalid!
```

Then you need to apply the fixed described below. Otherwise, stand in front of the camera and assume the "Psi pose" (see the openni_tracker Wiki page page for a picture) and after a short delay you should see messages similar to the following:

```
[ INFO] [1387675508.053121239]: New User 1
[ INFO] [1387675593.890846641]: Pose Psi detected for user 1
[ INFO] [1387675593.919569971]: Calibration started for user 1
[ INFO] [1387675594.629105834]: Calibration complete, start
tracking user 1
```

If this works for you, your installation is working OK and you can skip the steps below.

If you got the error listed above, try the following steps to fix your NiTE installation.

1. Download the NiTE v1.5.2.23 binary package from one of the following links depending on whether you are using a 32-bit or 64-bit installation of Ubuntu:

   a. **32-bit**: http://www.openni.org/wp-content/uploads/2013/10/NITE-Bin-Linux-x86-v1.5.2.23.tar.zip

   b. **64-bit**: http://www.openni.org/wp-content/uploads/2013/10/NITE-Bin-Linux-x64-v1.5.2.23.tar.zip

2. Unzip and extract the archive to a location of your choice (e.g. ~/tmp)

3. The unzipped archive is actually another archive in bz2 format so unzip and extract the archive to the same location as in Step 2.

4. The resulting folder should be called NITE-Bin-Dev-Linux-x64-v1.5.2.23 (64-bit) or NITE-Bin-Linux-x86-v1.5.2.23 (32-bit)/ Move into this folder and then

run the uninstall.sh script followed by the install.sh scripts. For the 64-bit version this would look like the following:

```
$ cd ~/tmp/NITE-Bin-Dev-Linux-x64-v1.5.2.23
$ sudo ./uninstall.sh
$ sudo ./install.sh
```

That should do it. You can now try the `openni_tracker` command again as described above.

10.9.2 Viewing Skeletons in `RViz`

The ROS openni_tracker package connects to a PrimeSense device such as a Kinect or Asus Xtion and broadcasts a ROS frame transform for each skeleton joint detected in front of the camera. The `tf` transforms are defined relative to the `openni_depth_frame` which is embedded inside the camera behind the depth sensor.

To view the skeleton frames in `RViz`, perform the following steps. First, plug in your Kinect or Asus camera and in the case of the Kinect, make sure it has power as well. Be sure to terminate any `openni` launch files you might already have running. Then run the `openni_tracker` command:

```
$ rosrun openni_tracker openni_tracker
```

Bring up `RViz` with the included `skeleton_frames.rviz` config file:

```
$ rosrun rviz rviz -d `rospack find rbx1_vision`/skeleton_frames.rviz
```

Keep your eye on `RViz` and stand back from the camera at least a 5 or 6 feet while assuming the "Psi pose". (See the openni_tracker page for a picture.) Once the tracker locks onto you, you should see your skeleton `tf` frames appear in `RViz`. At this point you can move around as you like in front of the camera and the skeleton in `RViz` should mimic your actions.

10.9.3 Accessing Skeleton Frames in your Programs

Since the `openni_tracker` node makes the skeleton joints available as ROS `tf` frames, we can use a `tf TransformListener` to find the current position of a given joint. An example of how this works can be found in the skeleton_markers package. You can install it into your personal ROS catkin directory, using the following commands:

```
$ cd ~/catkin_ws/src
$ git clone https://github.com/pirobot/skeleton_markers.git
$ cd skeleton_markers
$ git checkout hydro-devel
$ cd ~/catkin_ws
$ catkin_make
$ source devel/setup.bash
```

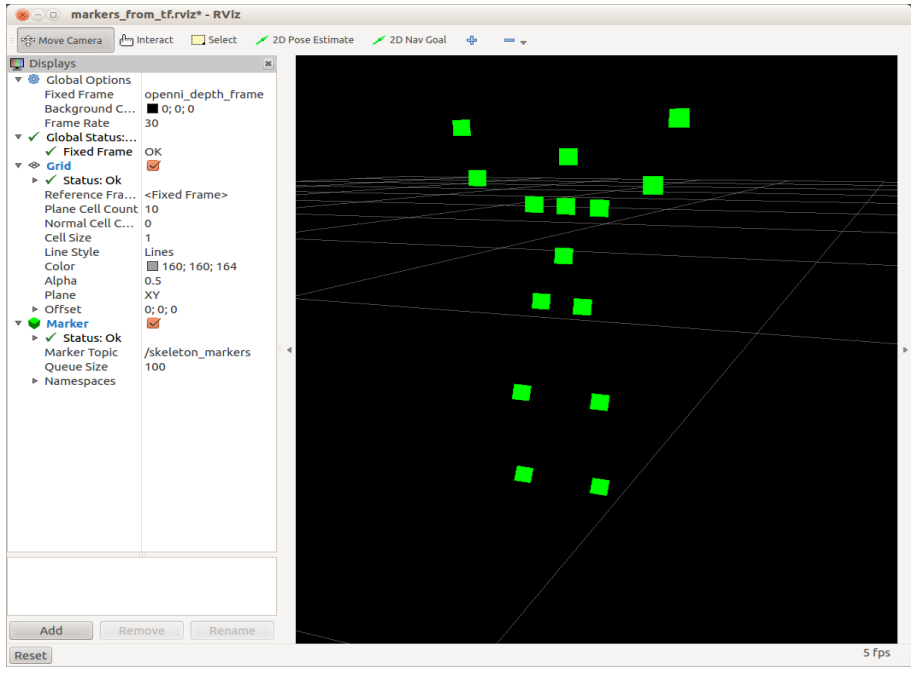
Let's try it out before looking at the code. First terminate any instances of `openni_tracker` and `RViz` you may have launched in the previous section. Next, run the following two commands:

```
$ roslaunch skeleton_markers markers_from_tf.launch
```

and

```
$ rosrun rviz rviz -d `rospack find \
  skeleton_markers`/markers_from_tf.rviz
```

Now assume the "Psi Pose" in front of the camera while keeping an eye on `RViz` until calibration is complete and tracking begins. Once the tracker locks onto you, you should see the green skeleton markers appear in `RViz`. At this point you can move around as you like in front of the camera and the skeleton in `RViz` should follow your actions as shown in the image below:



Before we look at the code, let's be sure we understand the launch file `markers_from_tf.launch` listed below:

```
<launch>
  <node pkg="openni_tracker" name="openni_tracker" type="openni_tracker"
output="screen">
    <param name="fixed_frame" value="openni_depth_frame" />
  </node>

  <node pkg="skeleton_markers" name="markers_from_tf" type="markers_from_tf.py"
output="screen">
    <rosparam file="$(find skeleton_markers)/params/marker_params.yaml"
command="load" />

  </node>
</launch>
```

First we run the `openni_tracker` node with the fixed frame set to the camera's depth frame. (This is the default so strictly speaking, specifying the parameter in the launch file is not necessary.) Then we fire up our `markers_from_tf.py` script and load the parameter file `marker_params.yaml` from the `params` directory. This file defines parameters describing the appearance of the markers as well as the list of skeleton frames we want to track.

Let's now look at the `markers_from_tf.py` script. Our overall strategy will be to use the `tf` library to find the transformation between each skeleton frame and the fixed depth frame. All we need for our purposes is the coordinates of each frame's origin relative to the depth frame. This allows us to place a visualization marker at that location to represent the position of the corresponding skeleton joint in space.

Link to source: markers_from_tf.py

```
1  #!/usr/bin/env python
2
3  import rospy
4
5  from visualization_msgs.msg import Marker
6  from geometry_msgs.msg import Point
7  import tf
8
9  class SkeletonMarkers():
10     def __init__(self):
11         rospy.init_node('markers_from_tf')
12
13         rospy.loginfo("Initializing Skeleton Markers Node...")
14
15         rate = rospy.get_param('~rate', 20)
16         r = rospy.Rate(rate)
17
18         # There is usually no need to change the fixed frame from the default
19         self.fixed_frame = rospy.get_param('~fixed_frame',
'openni_depth_frame')
20
```

```
21          # Get the list of skeleton frames we want to track
22          self.skeleton_frames = rospy.get_param('~skeleton_frames', '')
23
24          # Initialize the tf listener
25          tf_listener = tf.TransformListener()
26
27          # Define a marker publisher
28          marker_pub = rospy.Publisher('skeleton_markers', Marker)
29
30          # Intialize the markers
31          self.initialize_markers()
32
33          # Make sure we see the openni_depth_frame
34          tf_listener.waitForTransform(self.fixed_frame, self.fixed_frame,
rospy.Time(), rospy.Duration(60.0))
35
36          # A flag to track when we have detected a skeleton
37          skeleton_detected = False
38
39          # Begin the main loop
40          while not rospy.is_shutdown():
41              # Set the markers header
42              self.markers.header.stamp = rospy.Time.now()
43
44              # Clear the markers point list
45              self.markers.points = list()
46
47              # Check to see if a skeleton is detected
48              while not skeleton_detected:
49                  # Assume we can at least see the head frame
50                  frames = [f for f in tf_listener.getFrameStrings() if
f.startswith('head_')]
51
52                  try:
53                      # If the head frame is visible, pluck off the
54                      # user index from the name
55                      head_frame = frames[0]
56                      user_index = head_frame.replace('head_', '')
57
58                      # Make sure we have a transform between the head
59                      # and the fixed frame
60                      try:
61                          (trans, rot)  =
tf_listener.lookupTransform(self.fixed_frame, head_frame, rospy.Time(0))
62                          skeleton_detected = True
63
64                      except (tf.Exception, tf.ConnectivityException,
tf.LookupException):
65                          skeleton_detected = False
66                          rospy.loginfo("User index: " + str(user_index))
67                          r.sleep()
68                  except:
69                      skeleton_detected = False
70
71              # Loop through the skeleton frames
72              for frame in self.skeleton_frames:
73                  # Append the user_index to the frame name
```

```
                    skel_frame = frame + "_" + str(user_index)

                    # We only need the origin of each skeleton frame
                    # relative to the fixed frame
                    position = Point()

                    # Get the transformation from the fixed frame
                    # to the skeleton frame
                    try:
                        (trans, rot)  =
tf_listener.lookupTransform(self.fixed_frame, skel_frame, rospy.Time(0))
                        position.x = trans[0]
                        position.y = trans[1]
                        position.z = trans[2]

                        # Set a marker at the origin of this frame
                        self.markers.points.append(position)
                    except:
                        pass

                # Publish the set of markers
                marker_pub.publish(self.markers)

                r.sleep()

    def initialize_markers(self):
        # Set various parameters
        scale = rospy.get_param('~scale', 0.07)
        lifetime = rospy.get_param('~lifetime', 0) # 0 is forever
        ns = rospy.get_param('~ns', 'skeleton_markers')
        id = rospy.get_param('~id', 0)
        color = rospy.get_param('~color', {'r': 0.0, 'g': 1.0, 'b': 0.0, 'a':
1.0})

        # Initialize the marker points list
        self.markers = Marker()
        self.markers.header.frame_id = self.fixed_frame
        self.markers.ns = ns
        self.markers.id = id
        self.markers.type = Marker.POINTS
        self.markers.action = Marker.ADD
        self.markers.lifetime = rospy.Duration(lifetime)
        self.markers.scale.x = scale
        self.markers.scale.y = scale
        self.markers.color.r = color['r']
        self.markers.color.g = color['g']
        self.markers.color.b = color['b']
        self.markers.color.a = color['a']

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

```

Let's examine the key lines:

```
18          # There is usually no need to change the fixed frame from the default
19          self.fixed_frame = rospy.get_param('~fixed_frame',
'openni_depth_frame')
20
21          # Get the list of skeleton frames we want to track
22          self.skeleton_frames = rospy.get_param('~skeleton_frames', '')
```

Here we set the fixed frame to be the default used by the `openni_tracker` node. We also read in the list of skeleton frames we want to track as specified in the parameter file `marker_params.yaml`. The parameter file looks like this:

```
# The fixed reference frame
fixed_frame: 'openni_depth_frame'

# Update rate
rate: 20

# Height and width of markers in meters
scale: 0.07

# Duration of markers in RViz; 0 is forever
lifetime: 0

# Marker namespace
ns: 'skeleton_markers'

# Marker id
id: 0

# Marker color
color: { 'r': 0.0, 'g': 1.0, 'b': 0.0, 'a': 1.0 }

skeleton_frames: [
head,
neck,
torso,
left_shoulder,
left_elbow,
left_hand,
left_hip,
left_knee,
left_foot,
right_shoulder,
right_elbow,
right_hand,
right_hip,
right_knee,
right_foot
]
```

The parameter file is used to define the update rate, the appearance of the markers to display in `RViz` and the list of skeleton frames we want to track.

Returning to the `markers_from_tf.py` script:

```
24          # Initialize the tf listener
25          tf_listener = tf.TransformListener()
26
27          # Define a marker publisher
28          marker_pub = rospy.Publisher('skeleton_markers', Marker)
```

Here we create a `TransformListener` from the ROS `tf` library and set up a publisher for the visualization markers.

```
31          self.initialize_markers()
```

This line calls a function (defined later in the script) to initialize the markers. We won't discuss markers in this volume but you can probably follow the initializing function fairly easily. You can also look at the Markers Tutorial on the ROS Wiki, although the examples are in C++.

```
34          tf_listener.waitForTransform(self.fixed_frame, self.fixed_frame,
rospy.Time(), rospy.Duration(60.0))
```

Before we start looking for skeleton frames, we make sure we can at least see the camera's fixed frame and we allow 60 seconds before we time out.

```
37          skeleton_detected = False
```

A flag to indicate whether or not a skeleton is visible.

```
39          # Begin the main loop
40          while not rospy.is_shutdown():
41              # Set the markers header
42              self.markers.header.stamp = rospy.Time.now()
43
44              # Clear the markers point list
40              self.markers.points = list()
```

Now we enter the main loop of the script. First we time stamp the marker list and clear any marker coordinates.

```
48              while not skeleton_detected:
49                  # Assume we can at least see the head frame
50                  frames = [f for f in tf_listener.getFrameStrings() if
f.startswith('head_')]
```

Next we use the `tf` listener to get a list of all available frames and check to see if we can see the head frame.

```
52                  try:
53                      # If the head frame is visible, pluck off the
54                      # user index from the name
55                      head_frame = frames[0]
```

```
56                      user_index = head_frame.replace('head_', '')
57
58                      # Make sure we have a transform between the head
59                      # and the fixed frame
60                      try:
61                          (trans, rot)  =
tf_listener.lookupTransform(self.fixed_frame, head_frame, rospy.Time(0))
62                          skeleton_detected = True
63
64                      except (tf.Exception, tf.ConnectivityException,
tf.LookupException):
65                          skeleton_detected = False
66                          rospy.loginfo("User index: " + str(user_index))
67                          r.sleep()
68                  except:
69                      skeleton_detected = False
```

If we have a head frame, the outside `try-except` block will succeed and we look up the transform between the head frame and the fixed frame. If the lookup succeeds, we can be confident we have detected a skeleton and we set the flag to `True` so that we can break out of the outer `while` loop.

```
72              for frame in self.skeleton_frames:
73                  # Append the user_index to the frame name
74                  skel_frame = frame + "_" + str(user_index)
75
76                  # We only need the origin of each skeleton frame
77                  # relative to the fixed frame
78                  position = Point()
79
80                  # Get the transformation from the fixed frame
81                  # to the skeleton frame
82                  try:
83                      (trans, rot)  =
tf_listener.lookupTransform(self.fixed_frame, skel_frame, rospy.Time(0))
84                      position.x = trans[0]
85                      position.y = trans[1]
86                      position.z = trans[2]
87
88                      # Set a marker at the origin of this frame
89                      self.markers.points.append(position)
90                  except:
91                      pass
```

Now we can loop through all skeleton frames and attempt to look up the transform between each frame and the fixed frame. If the lookup is successful, the translation and rotation are returned separately. We only care about the translation component (the location of the frame's origin relative to the fixed frame) so we store x, y and z components in the `position` variable we initialized earlier as type `Point`. This point is then appended to the list of markers.

```
94              marker_pub.publish(self.markers)
95
```

```
77              r.sleep()
```

Once we have a marker for each frame, we publish the whole set of markers and then wait for one rate cycle before starting a new loop.

10.10 PCL Nodelets and 3D Point Clouds

The Point Cloud Library, or PCL, is an expansive project including many powerful algorithms for processing point clouds. It is especially useful for robots equipped with an RGB-D camera like the Kinect at Xtion Pro or even a more traditional stereo camera. While the details of using PCL are beyond the scope of this volume, we can at least touch on some of the basics.

At the time of this writing, the primary API for PCL is still C++. If you are already an experienced C++ programmer, you can start right away with the excellent tutorials on the PCL website. For Python enthusiasts, a preliminary set of Python bindings is now available. In the meantime, the pcl_ros package provides a number of nodelets for processing point clouds using PCL without having to write any code at all. For this reason, we will take a brief look at a couple of functions we can perform on point clouds using `pcl_ros`. (You can also find a complete list of `pcl_ros` nodelets in the pcl_ros tutorials on the ROS Wiki.)

10.10.1 The `PassThrough` Filter

The first PCL nodelet we will look at is the PassThrough filter. This filter is handy if you want to restrict your attention to the part of the image that falls within a certain depth range. We will use this filter in our Person Follower application in the next chapter since we will want the robot to follow only those objects that are located within a certain distance.

Before trying out the filter, let's look at the launch file `passthrough.launch` in the `rbx1_vision/launch` directory:

```
<launch>

  <!-- Start the nodelet manager -->
  <node pkg="nodelet" type="nodelet" name="pcl_filter_manager" args="manager"
output="screen" />

  <!-- Run a passthrough filter on the z axis -->
  <node pkg="nodelet" type="nodelet" name="passthrough" args="load
pcl/PassThrough pcl_filter_manager" output="screen">
    <remap from="~input" to="/camera/depth/points" />
    <remap from="~output" to="/passthrough" />
    <rosparam>
      filter_field_name: z
      filter_limit_min: 1.0
      filter_limit_max: 1.25
      filter_limit_negative: False
```

```
    </rosparam>
  </node>
</launch>
```

The launch file first loads the `pcl_filter_manager` nodelet and then the `passthrough` nodelet. The `passthrough` nodelet takes a number of parameters described as follows:

- `filter_field_name`: Typically this will be either `x`, `y` or `z` to indicate the optical axis that should be filtered. By "filtered" we mean that only points within the min and max limits (defined next) will be kept. Remember that the z-axis points out from the camera and is what we normally mean by depth.
- `filter_limit_min`: The minimum value (in meters) we will accept.
- `filter_limit_max`: The maximum value (in meters) we will accept.
- `filter_limit_negative`: If set to `True`, then only keep what lies *outside* the filter limits.

In the sample launch file, we have set the min and max limits to 1.0 and 1.25 meters, respectively. This means that only points that fall between about 3 feet and 4 feet from the camera will be retained.

Note that the launch file sets the input point cloud topic to `/camera/depth/points` and the output topic to `/passthrough`. This output topic name is needed when you want to view the results in `RViz`.

To see the result, start by launching the `openni_node` (if it isn't already):

```
$ roslaunch rbx1_vision openni_node.launch
```

In another terminal, launch the passthrough filter:

```
$ roslaunch rbx1_vision passthrough.launch
```

Next, bring up `RViz` with the provided PCL config file:

```
$ rosrun rviz rviz -d `rospack find rbx1_vision`/pcl.rviz
```

When `RViz` is up, look in the **Displays** panel on the left. By default, the **Original PointCloud** should be checked and you should see a color-coded point cloud in the main display. Use your mouse to rotate the cloud and view it from different angles. You can also zoom in and out of the point cloud using your mouse scroll wheel.

To see the result of the `Passthrough` filter, un-check the **Original PointCloud** display and check the **PassThrough** display. The **PassThrough** display topic should be set to `/passthrough`. If not, click on the **Topic** field and select it from the list.

Since we have set the distance limits to a fairly narrow window, do not be alarmed if you do not see any points in `RViz` when selecting the **PassThrough** display. However, stand in front of the camera and move toward and away from it and you should see the image of yourself appear and disappear as you pass through the filter limits.

To try different values of the minimum and maximum limits as well as the `filter_limit_negative` parameter, first bring up `rqt_reconfigure`:

```
$ rosrun rqt_reconfigure rqt_reconfigure
```

then select the `passthrough` node. If you check the checkbox beside `filter_limit_negative`, you should now find a "hole" in between the distance limits. Now everything *outside* your limits should be visible but as you move through the hole, you will disappear.

By the way, you can change how depth is displayed for a given point cloud topic by changing the **Color Transformer** type listed under **PassThrough** display options. For example, selecting *RGB8* instead of *Axis* will display the actual pixel color values while the *Axis* setting uses a color-coded depth map. The recommended setting for the **Style** setting is *Points*. You can try other options here but you will likely find that they slow down your computer significantly unless you have a very powerful machine.

10.10.2 Combining More than One `PassThrough` Filter

We can combine more than one `PassThrough` filter in a single launch file. In particular, we can put limits on all three dimensions and create a box such that only points inside the box are visible. The result is a kind of 3D focus of attention. If we set the `filter_limit_negative` parameter to `True` for each filter, only points outside the box will be visible.

As an example, take a look at the launch file `passthrough2.launch` in the directory `rbx1_vision/launch`. The launch file includes three instances of the `passthrough` nodelet, one for each optical dimension x, y and z. Note that each nodelet must have a unique name so we have called them `passthrough_x`, `passthrough_y` and `passthrough_z`. We then set min/max distance limits on each filter to create a box of the desired size. Note how the input topic of a subsequent filter is set to be the output topic of the one before it. The final result of all three filters is still published on the `/passthrough` topic so you can use the same `RViz` config file to see it. Terminate the previous `passthrough.launch` file and run:

```
$ roslaunch rbx1_vision passthrough2.launch
```

Now move around in front of the camera and you should be able to determine the boundaries of the passthrough box. You can also play with the settings in `rqt_reconfigure` to create filter boxes of different size. (**Note**: `rqt_reconfigure` does not automatically refresh the list of nodes when a new node is launched. So you will have to exit `rqt_reconfigure` and bring it up again.)

10.10.3 The `VoxelGrid` Filter

The second filter we will look at is the VoxelGrid filter. Point clouds from high resolution cameras contain a very large number of points and can take a lot of CPU power to process. To reduce the load and ensure faster frame rates, the `VoxelGrid` filter does a downsampling of the input cloud using parameters specified in the launch file. The result is a new point cloud with many fewer points. Let's take a look at the launch file `voxel.launch` found in the `rbx1_vision/launch` directory:

```
<launch>

  <!-- Start the nodelet manager -->
  <node pkg="nodelet" type="nodelet" name="pcl_manager" args="manager"
output="screen" />

  <!-- Run a VoxelGrid filter to clean NaNs and downsample the data -->
  <node pkg="nodelet" type="nodelet" name="voxel_grid" args="load pcl/VoxelGrid
pcl_manager" output="screen">
    <remap from="~input" to="/camera/depth/points" />
    <remap from="~output" to="/voxel_grid" />
    <rosparam>
      filter_field_name: z
      filter_limit_min: 0.01
      filter_limit_max: 3.5
      filter_limit_negative: False
      leaf_size: 0.05
    </rosparam>
  </node>
</launch>
```

Notice that the first four parameters are identical to the `PassThrough` filter and the behave the same way. The fourth parameter, `leaf_size`, determines the coarseness of the sampling grid. A value of 0.05 means that the original point cloud is sampled every 5cm to produce the output cloud.

If you are still running a `passthrough2` launch file from the previous section, terminate it now. If you don't still have the openni node running, bring it up now:

```
$ roslaunch rbx1_vision openni_node.launch
```

If you don't already have `RViz` running with the `pcl.rviz` config file, run it now:

```
$ rosrun rviz rviz -d `rospack find rbx1_vision`/pcl.rviz
```

Finally, fire up the `VoxelGrid` filter:

```
$ roslaunch rbx1_vision voxel.launch
```

To view the result of the `VoxelGrid` filter, un-check both the **Original PointCloud** and **PassThrough** displays in `RViz` and check the box beside **VoxelGrid**. You should see a fairly sparse cloud. This is especially apparent if you zoom in on the cloud using your mouse scroll wheel. You also use `rqt_reconfigure` to change the `leaf_size` on the fly and observe the result in `RViz`.

11. Combining Vision and Base Control

Now that we have a handle on basic vision processing and motion control in ROS, we are ready to put them together into a full ROS application. In this chapter we will look at two such applications: object tracking and person following.

11.1 A Note about Camera Coordinate Axes

ROS assumes that the optical coordinate frame attached to a given camera is oriented with the z-axis pointing outward from the camera and perpendicular to the image plane. Within the image plane, the x-axis points horizontally to the right and the y-axis points vertically downward.

Note that this frame differs from the one typically attached to a mobile base which, as we saw earlier, has the z-axis pointing upward, the x-axis pointing forward, and the y-axis point to the left. While we won't have to worry too much about this difference in this volume, it is important to keep in mind that when we talk about "depth" in the context of an RGB-D camera such as the Kinect, we are talking about the camera's z-axis which points outward from the camera.

11.2 Object Tracker

In the chapter on Robot Vision, we learned how to use OpenCV to track faces, keypoints and colors. In all cases, the result is a region of interest (ROI) that follows the object and is published on the ROS topic `/roi`. If the camera is mounted on a mobile robot, we can use the `x_offset` coordinate of the `/roi` to keep the object centered in the field of view by rotating the robot to compensate for the offset. In this way, the robot will rotate left or right to track the object as it moves. (In the next section we will add depth to the equation so that the robot can also move forward and backward and keep the object at a fixed distance as it moves away or comes closer.)

Our tracker node can be found in the `object_tracker.py` script located in the directory `rbx1_apps/nodes`. Before we review the code, let's test the tracking as follows.

11.2.1 Testing the Object Tracker with `rqt_plot`

This test can be run without a robot and requires only a camera attached to your computer. Start by launching the driver for either a depth camera or a webcam:

```
$ roslaunch rbx1_vision openni_node.launch
```

or

```
$ roslaunch rbx1_vision uvc_cam.launch device:=/dev/video0
```

(Change the video device if necessary.)

Next, bring up the `face_tracker2` node we developed earlier:

```
$ roslaunch rbx1_vision face_tracker2.launch
```

Now bring up the `object_tracker` node:

```
$ roslaunch rbx1_apps object_tracker.launch
```

Finally, run `rqt_plot` to view the angular component of the `/cmd_vel` topic:

```
$ rqt_plot /cmd_vel/angular/z
```

Now look at the camera and when the face tracker detects your face, move your head left and right. As you do so, you should see the angular velocity in `rqt_plot` vary between roughly -1.5 and 1.5 radians per second. The further away from center you move your face, the larger the velocity value on `rqt_plot`. If these values were being sent to a robot, the robot would rotate accordingly.

11.2.2 Testing the Object Tracker with a Simulated Robot

Instead of `rqt_plot`, we can use the ArbotiX simulator to test the functionality of the object tracker code. The procedure is the same as the preceding section except that instead of running `rqt_plot` as the last step, bring up the simulator and `RViz` as we have done before:

```
$ roslaunch rbx1_bringup fake_turtlebot.launch
```

```
$ rosrun rviz rviz -d `rospack find rbx1_nav`/sim.rviz
```

You should see the fake TurtleBot in `RViz`. If you now move your head left or right in front of the camera, the TurtleBot should rotate in the corresponding direction. The further left/right you move, the faster it should rotate. Note that the robot will continue to rotate as long as your face is off-center since the simulation cannot line up with your real face.

11.2.3 Understanding the Object Tracker Code

Before running the tracking application on a real robot, let's take a look at the code.

Link to source: object_tracker.py

```
1  #!/usr/bin/env python
2
3  import rospy
4  from sensor_msgs.msg import RegionOfInterest, CameraInfo
5  from geometry_msgs.msg import Twist
6  import thread
7
8  class ObjectTracker():
9      def __init__(self):
10         rospy.init_node("object_tracker")
11
12         # Set the shutdown function (stop the robot)
13         rospy.on_shutdown(self.shutdown)
14
15         # How often should we update the robot's motion?
16         self.rate = rospy.get_param("~rate", 10)
17         r = rospy.Rate(self.rate)
18
19         # The maximum rotation speed in radians per second
20         self.max_rotation_speed = rospy.get_param("~max_rotation_speed", 2.0)
21
22         # The minimum rotation speed in radians per second
23         self.min_rotation_speed = rospy.get_param("~min_rotation_speed", 0.5)
24
25         # Sensitivity to target displacements.  Setting this too high
26         # can lead to oscillations of the robot.
27         self.gain = rospy.get_param("~gain", 2.0)
28
29         # The x threshold (% of image width) indicates how far off-center
30         # the ROI needs to be in the x-direction before we react
31         self.x_threshold = rospy.get_param("~x_threshold", 0.1)
32
33         # Publisher to control the robot's movement
34         self.cmd_vel_pub = rospy.Publisher('cmd_vel', Twist)
35
36         # Intialize the movement command
37         self.move_cmd = Twist()
38
39         # Get a lock for updating the self.move_cmd values
40         self.lock = thread.allocate_lock()
41
42         # We will get the image width and height from the camera_info topic
43         self.image_width = 0
44         self.image_height = 0
45
46         # Set flag to indicate when the ROI stops updating
47         self.target_visible = False
48
49         # Wait for the camera_info topic to become available
50         rospy.loginfo("Waiting for camera_info topic...")
51         rospy.wait_for_message('camera_info', CameraInfo)
```

```
52
53          # Subscribe the camera_info topic to get the image width and height
54          rospy.Subscriber('camera_info', CameraInfo, self.get_camera_info)
55
56          # Wait until we actually have the camera data
57          while self.image_width == 0 or self.image_height == 0:
58              rospy.sleep(1)
59
60          # Subscribe to the ROI topic and set the callback to update the
robot's motion
61          rospy.Subscriber('roi', RegionOfInterest, self.set_cmd_vel)
62
63          # Wait until we have an ROI to follow
64          rospy.wait_for_message('roi', RegionOfInterest)
65
66          rospy.loginfo("ROI messages detected. Starting tracker...")
67
68          # Begin the tracking loop
69          while not rospy.is_shutdown():
70              # Acquire a lock while we're setting the robot speeds
71              self.lock.acquire()
72
73              try:
74                  # If the target is not visible, stop the robot
75                  if not self.target_visible:
76                      self.move_cmd = Twist()
77                  else:
78                      # Reset the flag to False by default
79                      self.target_visible = False
80
81                  # Send the Twist command to the robot
82                  self.cmd_vel_pub.publish(self.move_cmd)
83
84              finally:
85                  # Release the lock
86                  self.lock.release()
87
88              # Sleep for 1/self.rate seconds
89              r.sleep()
90
91      def set_cmd_vel(self, msg):
92          # Acquire a lock while we're setting the robot speeds
93          self.lock.acquire()
94
95          try:
96              # If the ROI has a width or height of 0, we have lost the target
97              if msg.width == 0 or msg.height == 0:
98                  self.target_visible = False
99                  return
100
101             # If the ROI stops updating this next statement will not happen
102             self.target_visible = True
103
104             # Compute the displacement of the ROI from the center of the image
105             target_offset_x = msg.x_offset + msg.width / 2 -
self.image_width / 2
106
```

```
107             try:
108                 percent_offset_x = float(target_offset_x) /
(float(self.image_width) / 2.0)
109             except:
110                 percent_offset_x = 0
111
112             # Rotate the robot only if the displacement of the target exceeds
the threshold
113             if abs(percent_offset_x) > self.x_threshold:
114                 # Set the rotation speed proportional to the displacement of
the target
115                 try:
116                     speed = self.gain * percent_offset_x
117                     if speed < 0:
118                         direction = -1
119                     else:
120                         direction = 1
121                     self.move_cmd.angular.z = -direction *
max(self.min_rotation_speed,
122                                                     min(self.max_rotation_speed,
abs(speed)))
123                 except:
124                     self.move_cmd = Twist()
125             else:
126                 # Otherwise stop the robot
127                 self.move_cmd = Twist()
128
129         finally:
130             # Release the lock
131             self.lock.release()
132
133     def get_camera_info(self, msg):
134         self.image_width = msg.width
135         self.image_height = msg.height
136
137     def shutdown(self):
138         rospy.loginfo("Stopping the robot...")
139         self.cmd_vel_pub.publish(Twist())
140         rospy.sleep(1)
141
142 if __name__ == '__main__':
143     try:
144         ObjectTracker()
145         rospy.spin()
146     except rospy.ROSInterruptException:
147         rospy.loginfo("Object tracking node terminated.")
```

By this point in the book, the script will probably be fairly self-explanatory. However, let's run through the key lines. Keep in mind the overall goal of the tracker application: we want to monitor the `/roi` topic for any change in the position of the target to either the left of right of the camera's center of view. We will then rotate the robot in the appropriate direction to compensate.

```
19          # The maximum rotation speed in radians per second
20          self.max_rotation_speed = rospy.get_param("~max_rotation_speed", 2.0)
21
22          # The minimum rotation speed in radians per second
23          self.min_rotation_speed = rospy.get_param("~min_rotation_speed", 0.5)
```

When controlling a mobile robot, it is always a good idea to set a maximum speed. Setting a minimum speed as well can ensure that the robot does not struggle against its own weight and friction when trying to move too slowly.

```
27          self.gain = rospy.get_param("~gain", 2.0)
```

Most feedback loops require a gain parameter to control how fast we want the system to respond to displacements of the target from the neutral point. In our case, the gain parameter will determine how quickly the robot reacts to movements of the target away from the center of view.

```
31          self.x_threshold = rospy.get_param("~x_threshold", 0.05)
```

We don't want the robot to be draining its battery chasing every tiny movement of the target. So we set a threshold on the horizontal displacement of the target to which the robot will respond. In this case, the threshold is specified as a percentage (0.05 = 5%) of the image width.

```
39          # Get a lock for updating the self.move_cmd values
40          self.lock = thread.allocate_lock()
```

Callback functions assigned to ROS subscribers run in a separate thread from the main program. Since we will be modifying the robot's rotation speed both in the ROI callback function and in the main program loop, we need to use a lock to make the overall script thread safe. We will see how to implement the lock below.

```
47          self.target_visible = False
```

If the target is lost (e.g. goes out of the field of view), we want the robot to stop. So we will use a flag to indicate when the target is visible or not.

```
54          rospy.Subscriber('camera_info', CameraInfo, self.get_camera_info)
```

Rather than hard code the video resolution into the program, we can get it dynamically from the appropriate `camera_info` topic. The actual name of this topic is mapped in the launch file `object_tracker.launch`. In the case of a Kinect or Xtion camera driven by the OpenNI node, the topic name is usually `/camera/rgb/camera_info`. The callback function `self.get_camera_info` (defined later in the script) simply sets the global variables `self.image_width` and `self.image_height` from the `camera_info` messages.

```
61          rospy.Subscriber('roi', RegionOfInterest, self.set_cmd_vel)
```

Here we subscribe to the `/roi` topic and set the callback function to `set_cmd_vel()` which will set the `Twist` command to send to the robot when the position of the target changes.

```
69          while not rospy.is_shutdown():
70              # Acquire a lock while we're setting the robot speeds
71              self.lock.acquire()
72
73              try:
74                  # If the target is not visible, stop the robot
75                  if not self.target_visible:
76                      self.move_cmd = Twist()
77                  else:
78                      # Reset the flag to False by default
79                      self.target_visible = False
80
81                  # Send the Twist command to the robot
82                  self.cmd_vel_pub.publish(self.move_cmd)
83
84              finally:
85                  # Release the lock
86                  self.lock.release()
87
88              # Sleep for 1/self.rate seconds
89              r.sleep()
```

This is our main control loop. First we acquire a lock to protect the two global variables `self.move_cmd` and `self.target_visible` since both variables can also be modified in the `set_cmd_vel()` callback function. Then we test to see if the target is still visible. If not we stop the robot by setting the movement command to the empty `Twist` message. Otherwise, we set the `target_visible` flag back to `False` (the safe default) and then publish the current movement command as set in the `set_cmd_vel()` callback described next. Finally, we release the lock and sleep for one cycle.

```
91      def set_cmd_vel(self, msg):
92          # Acquire a lock while we're setting the robot speeds
93          self.lock.acquire()
94
95          try:
96              # If the ROI has a width or height of 0, we have lost the target
97              if msg.width == 0 or msg.height == 0:
98                  self.target_visible = False
99                  return
100
101             # If the ROI stops updating this next statement will not happen
102             self.target_visible = True
```

The `set_cmd_vel()` callback gets triggered whenever there is a new message on the `/roi` topic. The first thing to check is that neither the width nor height of the ROI is zero. If so, the target has probably been lost so we return immediately to the main loop which will stop the robot from rotating. Otherwise, we set the `target_visible` flag to `True` so that the robot will react to the target's position.

```
89          target_x_offset = msg.x_offset + (msg.width / 2.0) - (self.image_width
/ 2.0)
90
91          try:
92              percent_offset = float(target_x_offset) / (float(self.image_width)
/ 2.0)
93          except:
94              percent_offset = 0
```

Before we can determine how the robot should move, we compute the displacement of the target from the center of the camera image. Recall that the `x_offset` field in an ROI message specifies the x-coordinate of the upper left corner of the region, so to find the center of the region we add half the width. Then to find the displacement from the center of the image we subtract half the width of the image. With the displacement computed in pixels, we then get the displacement as a fraction of the image width. The `try-catch` block ensures we trap any attempt to divide by zero which can happen if the `camera_info` topic hiccups and sends us an `image_width` value of 0.

```
97          if abs(percent_x_offset) > self.x_threshold:
98              # Set the rotation speed proportional to the displacement of the
target
99              try:
100             speed = self.gain * percent_x_offset
101             if speed < 0:
102                 direction = -1
103             else:
104                 direction = 1
105             self.move_cmd.angular.z = -direction *
max(self.min_rotation_speed, min(self.max_rotation_speed, abs(speed)))
106         except:
107             self.move_cmd = Twist()
108     else:
109         # Otherwise stop the robot
110         self.move_cmd = Twist()
```

Finally we compute the rotation speed of the robot to be proportional to the displacement of the target where the multiplier is the parameter `self.gain`. If the target offset does not exceed the `x_threshold` parameter, we set the movement command to the empty `Twist` message to stop the robot (or keep it stopped).

11.2.4 Object Tracking on a Real Robot

We're now ready to try the object tracker on a real robot. Since the robot will only be rotating left and right, you shouldn't have to worry about it running into anything, but move it into a clear area anyway.

If you are using a TurtleBot with its fixed camera location, you might have to get on your hands and knees to do face tracking. Alternatively, you can run the CamShift tracker and hold a colored object in from the robot to test tracking. To get it all rolling, follow these steps.

First terminate any launch files you might have running from the previous section. Then start up your robot's launch files. For the original TurtleBot we would run:

```
$ roslaunch rbx1_bringup turtlebot_minimal_create.launch
```

and

```
$ roslaunch rbx1_vision openni_node.launch
```

Next launch either the CamShift tracker or face tracker:

```
$ roslaunch rbx1_vision camshift.launch
```

or

```
$ roslaunch rbx1_vision face_tracker2.launch
```

Finally, bring up the object tracker node:

```
$ roslaunch rbx1_apps object_tracker.launch
```

If you launched the Camshift tracker, move a brightly colored object in front of the camera, select it with the mouse, and adjust the parameters so you get good isolation of the target in the backprojection window. Now move the object left or right and the robot should then rotate to keep the object centered in the image frame.

If you are running the face tracker instead, move in front of your robot's camera so your face is in the field of view, wait for your face to be detected, then move to the left or right. The robot should rotate to keep your face centered in the image frame.

Try adjusting the parameters in the `object_tracker.launch` file to get the response sensitivity you desire.

11.3 Object Follower

Our next script combines the object tracker with depth information so that the robot can also move forward and backward to keep the tracked object at a fixed distance. In this way, the robot can actually follow the target as it moves about. The script will track any target published on the `/roi` topic so we can use our earlier vision nodes such as the face tracker, CamShift or LK tracker nodes to provide the target.

The new script is called `object_follower.py` located in the `rbx1_apps/nodes` directory. In addition to the `/roi` topic, we now subscribe to the depth image published on the topic `/camera/depth/image_raw`. This allows us to compute an average distance over the region of interest and should reflect how far away the target object is from the camera. We can then move the robot forward or backward to keep the target at a given distance. (Remember that if your camera is displaced back from the front of the robot that you will want to account for that offset in the goal distance you set for following.)

11.3.1 Adding Depth to the Object Tracker

The object follower program is similar to the object tracker script but this time we will add depth information. There are two ways we can get depth values from an RGBD camera using ROS; we can subscribe either to the depth *image* published by the openni camera node and use OpenCV or we can subscribe to the depth *point cloud* and use PCL. For the object follower node, we will use the depth image and OpenCV. For our "person follower" script described in the next section, we will show how to use a point cloud and PCL nodelets.

The openni camera driver publishes the depth image on the topic `/camera/depth/image_raw` using the message type `sensor_msgs/Image`. Each "pixel" in the image stores the depth value at that point in *millimeters*. We will therefore need to divide these values by 1000 to get the results in meters. Our callback function will use `cv_bridge` to convert the depth image into a Numpy array that we can use to compute the average depth over the ROI.

Rather than list out the entire script, let's focus on what is new relative to the object tracker node from the previous section.

```
self.depth_subscriber = rospy.Subscriber("depth_image", Image,
self.convert_depth_image, queue_size=1)
```

Here we subscribe to the depth image topic and assign the callback function `convert_depth_image()`. We use the generic topic name "depth_image" so that we can remap it in the launch file. When using the openni camera driver to connect to the depth camera, the topic we want is `/camera/depth/image_raw`. If you look at the launch file `object_follower.launch` found in the `rbx1_apps/launch` directory, you will see that we do the appropriate remapping as follows:

```
    <remap from="camera_info" to="/camera/rgb/camera_info" />
    <remap from="depth_image" to="/camera/depth/image_raw" />
```

Our callback function `convert_depth_image()` then looks like this:

```
def convert_depth_image(self, ros_image):
    # Use cv_bridge() to convert the ROS image to OpenCV format
    try:
        # The depth image is a single-channel float32 image
        depth_image = self.cv_bridge.imgmsg_to_cv(ros_image, "32FC1")
    except CvBridgeError, e:
        print e

    # Convert the depth image to a Numpy array
    self.depth_array = np.array(depth_image, dtype=np.float32)
```

Here we use `CvBridge` to convert the depth image to a Numpy array and store it in the variable `self.depth_array`. This will allow us to access depth values for each x-y coordinate of the image that falls within the target ROI.

Recall from the object tracker script that we use the callback function `set_cmd_vel()` to map messages on the `/roi` topic to `Twist` commands to move the robot. In that case we computed the left-right offset of the target so that we could rotate the robot to keep the target in the center of the camera view. We now add the following code to get the average depth to the ROI.

```
# Acquire a lock while we're setting the robot speeds
self.lock.acquire()

try:

    ( ... some code omitted that is the same as object_tracker.py ...)

    # Initialize a few depth variables
    n_z = sum_z = mean_z = 0

    # Shrink the ROI slightly to avoid the target boundaries
    scaled_width = int(self.roi.width * self.scale_roi)
    scaled_height = int(self.roi.height * self.scale_roi)

    # Get the min/max x and y values from the scaled ROI
    min_x = int(self.roi.x_offset + self.roi.width * (1.0 - self.scale_roi) /
2.0)
    max_x = min_x + scaled_width
```

```
        min_y = int(self.roi.y_offset + self.roi.height * (1.0 - self.scale_roi) /
2.0)
        max_y = min_y + scaled_height

        # Get the average depth value over the ROI
        for x in range(min_x, max_x):
            for y in range(min_y, max_y):
                try:
                    # Get a depth value in millimeters
                    z = self.depth_array[y, x]

                    # Convert to meters
                    z /= 1000.0

                except:
                    # It seems to work best if we convert exceptions to 0
                    z = 0

                # Check for values outside max range
                if z > self.max_z:
                    continue

                # Increment the sum and count
                sum_z = sum_z + z
                n_z += 1

        # Stop the robot's forward/backward motion by default
        linear_x = 0

        # If we have depth values...
        if n_z:
            mean_z = float(sum_z) / n_z

            # Don't let the mean fall below the minimum reliable range
            mean_z = max(self.min_z, mean_z)

            # Check the mean against the minimum range
            if mean_z > self.min_z:
                # Check the max range and goal threshold
                if mean_z < self.max_z and (abs(mean_z - self.goal_z) >
self.z_threshold):
                    speed = (mean_z - self.goal_z) * self.z_scale
                    linear_x = copysign(min(self.max_linear_speed,
max(self.min_linear_speed, abs(speed))), speed)

        if linear_x == 0:
            # Stop the robot smoothly
            self.move_cmd.linear.x *= self.slow_down_factor
        else:
            self.move_cmd.linear.x = linear_x
    finally:
        # Release the lock
        self.lock.release()
```

ROS callbacks operate in their own thread so first we set a lock to protect any variables that might also be updated in the main body of our script. This includes the variable `self.move_cmd` for controlling the robot's motion and the self.target_visible flag that indicates whether or not we have lost the target. If we didn't use a lock, unpredictable results could occur when both the callback and the body of the script try to use these variables at the same time.

First we get the range of `x` and `y` values that covers an area slightly smaller than the ROI. We shrink the ROI by the factor `self.scale_roi` (default: `0.9`) so that we don't pick up distance values from the background near the edges of the tracked object. A more sophisticated approach would be to drop depth values that exceed the average depth by a certain threshold.

Next we loop over the x-y coordinates of the reduced ROI and pull the depth value at each point from the depth array we got from our depth image callback. Note how the order of x and y are reversed when indexing the depth array. We also check for exceptions and set the depth to zero for these points. We could also just drop these points but it turns out that by setting them to zero we actually get better following behavior when the person is close to the robot.

Finally, we compute the average depth and set the linear speed component of the robot to either move toward or away from the target depending on whether we are too far away or too close compared to our goal distance set in the `self.goal_z` parameter. In case we don't have any valid depth values, we stop the robot smoothly by reducing the linear speed by 10%. (Don't be confused by the use of "z" for depth info from the camera and "x" for the linear speed of the robot. This is simply a consequence of the two different coordinate conventions used for cameras and robot motion.)

When we are done with the callback, we release the lock.

In the meantime, our main loop is operating as follows;

```
        while not rospy.is_shutdown():
            # If the target is not visible, stop the robot smoothly
            self.lock.acquire()

            try:
                if not self.target_visible:
                    self.move_cmd.linear.x *= self.slow_down_factor
                    self.move_cmd.angular.z *= self.slow_down_factor
                else:
                    # Reset the flag to False by default
                    self.target_visible = False

            finally:
                self.lock.release()

            # Send the Twist command to the robot
            self.cmd_vel_pub.publish(self.move_cmd)
```

```
            # Sleep for 1/self.rate seconds
            r.sleep()
```

The only difference here compared to the same loop used in the `object_tracker.py` script is that we now acquire a lock at the beginning and release it at the end of each update cycle. This is to protect the variables `self.move_cmd` and `self.target_visible` that can also be modified by our callback `set_cmd_vel()`.

11.3.2 Testing the Object Follower with a Simulated Robot

Since the object follower requires depth information, the script will only work with a depth camera such as the Kinect or Xtion Pro. Therefore, start by attaching your camera to your computer and launching the OpenNI driver:

```
$ roslaunch rbx1_vision openni_node.launch
```

Next, bring up the fake TurtleBot and `RViz`:

```
$ roslaunch rbx1_bringup fake_turtlebot.launch
```

```
$ rosrun rviz rviz -d `rospack find rbx1_nav`/sim.rviz
```

You should see the fake TurtleBot in `RViz`.

Now launch the `face_tracker2` node we developed earlier:

```
$ roslaunch rbx1_vision face_tracker2.launch
```

Position the camera so that it can detect your face. Remember that you can hit the '`c`' key over the video window to clear the tracked points and force the node to re-detect your face.

Now bring up the `object_follower` node:

```
$ roslaunch rbx1_apps object_follower.launch
```

Make sure the `RViz` window is visible so that you can see the fake TurtleBot. If you now move your head in front of the camera, the fake TurtleBot should move to track your motion. The goal distance set in the `object_follower.launch` file is 0.7 meters so if you move your head within that distance of the camera, the robot will move backward. If you move further away than 0.7 meters, the robot will move forward.

You can try the same experiment using the CamShift node to track a colored object. Simply terminate the face tracker launch file and run the camshift launch file instead:

```
$ roslaunch rbx1_vision camshift.launch
```

Select the object you want to track with the mouse and watch the fake TurtleBot in `RViz` as you move the object in front of the camera.

11.3.3 Object Following on a Real Robot

We're now ready to try the object follower on a real robot. Before starting, make sure your robot has lots of room to move about.

If you are using a TurtleBot with its fixed camera location, you might have to get on your hands and knees to do face tracking. Alternatively, you can run the CamShift tracker and hold a colored object in from the robot to test tracking.

First terminate any launch files you might have running from the previous section. Then start up your robot's launch files. For the original TurtleBot we would run:

```
$ roslaunch rbx1_bringup turtlebot_minimal_create.launch
```

and

```
$ roslaunch rbx1_vision openni_node.launch
```

Next launch either the CamShift tracker or face tracker:

```
$ roslaunch rbx1_vision camshift.launch
```

or

```
$ roslaunch rbx1_vision face_tracker2.launch
```

Finally, bring up the object follower node:

```
$ roslaunch rbx1_apps object_follower.launch
```

If you launched the Camshift tracker, move a brightly colored object in front of the camera, select it with the mouse, and adjust the parameters so you get good isolation of the target in the backprojection window. Now move the object left, right or forward and back from the camera and the robot should then move to keep the object centered and at roughly the goal distance set in the launch file.

If you are running the face tracker instead, move in front of your robot's camera so your face is in the field of view, wait for your face to be detected, then move to the left or right, forward or back. The robot should move to track your motion.

Try adjusting the parameters in the `object_tracker.launch` file to get the response sensitivity you desire.

As yet another option, try running the `lk_tracker.launch` node rather than the face tracker or CamShift. This will allow the robot to follow any object you select with the mouse by tracking the keypoints on its surface.

11.4 Person Follower

Our second application is designed to make our robot follow a person as they walk around the room. If you have a TurtleBot, you can use the most excellent turtlebot_follower application by Tony Pratkanis that uses PCL and is written in C++. Our goal will be to write a similar application in Python where we don't have access to a full PCL API.

The ROS `sensor_msgs` package defines a class for the `PointCloud2` message type and a module called `point_cloud2.py` that we can use to access individual depth values. Tony Pratkanis' `turtlebot_follower` program doesn't really know what a person looks like. Instead, it uses the following strategy to detect a "person-like blob" in front of it and then keep that object within a certain distance:

- First, define the minimum and maximum size of the blob in the x, y and z dimensions. This way the robot won't tend to fixate on pieces of furniture or chair legs.
- Next, define how close we want the robot to stay to the blob (person). For a depth camera, the z-coordinate is the relevant dimension here.
- Start the main loop:
    - If the robot is too far or too close to the person, move forward or backward accordingly.
    - If the person is to the left or right of the robot, rotate right or left appropriately.
    - Publish the corresponding movement as a `Twist` message on the `/cmd_vel` topic.

Let's now program a similar ROS application in Python.

11.4.1 Testing the Follower Application in Simulation

The Python script that implements the follower application is called `follower.py` in the `rbx1_apps/nodes` directory. Before looking at the code, you can try it out in the ArbotiX simulator.

First, make sure your camera is plugged in, then run the openni driver:

```
$ roslaunch rbx1_vision openni_node.launch
```

Next, launch the follower application:

```
$ roslaunch rbx1_apps follower.launch
```

Finally, bring up the simulator and `RViz` as we have done before:

```
$ roslaunch rbx1_bringup fake_turtlebot.launch
```

```
$ rosrun rviz rviz -d `rospack find rbx1_nav`/sim.rviz
```

You should see the simulated TurtleBot in `RViz`. If you now move your body toward or away from the camera, the TurtleBot should move forward or backward. If you move to the right or left, the robot should rotate to track you. Since your body is not actually part of the simulation, the robot will continue to move as long as your are not centered or at the goal distance from the camera.

11.4.2 Understanding the Follower Script

Let's now take a look at the follower code.

Link to source: follower.py

```
1  #!/usr/bin/env python
2
3  import rospy
4  from roslib import message
5  from sensor_msgs.msg import PointCloud2
6  from sensor_msgs import point_cloud2
7  from geometry_msgs.msg import Twist
8  from math import copysign
9
10 class Follower():
11     def __init__(self):
12         rospy.init_node("follower")
13
14         # Set the shutdown function (stop the robot)
15         rospy.on_shutdown(self.shutdown)
16
17         # The dimensions (in meters) of the box in which we will search
```

```
18          # for the person (blob). These are given in camera coordinates
19          # where x is left/right,y is up/down and z is depth (forward/backward)
20          self.min_x = rospy.get_param("~min_x", -0.2)
21          self.max_x = rospy.get_param("~max_x", 0.2)
22          self.min_y = rospy.get_param("~min_y", -0.3)
23          self.max_y = rospy.get_param("~max_y", 0.5)
24          self.max_z = rospy.get_param("~max_z", 1.2)
25
26          # The goal distance (in meters) to keep between the robot
            # and the person
27          self.goal_z = rospy.get_param("~goal_z", 0.6)
28
29          # How far away from the goal distance (in meters) before the robot
reacts
30          self.z_threshold = rospy.get_param("~z_threshold", 0.05)
31
32          # How far away from being centered (x displacement) on the person
33          # before the robot reacts
34          self.x_threshold = rospy.get_param("~x_threshold", 0.1)
35
36          # How much do we weight the goal distance (z) when making a movement
37          self.z_scale = rospy.get_param("~z_scale", 1.0)
38
39          # How much do we weight x-displacement of the person when
            # making a movement
40          self.x_scale = rospy.get_param("~x_scale", 2.5)
41
42          # The maximum rotation speed in radians per second
43          self.max_angular_speed = rospy.get_param("~max_angular_speed", 2.0)
44
45          # The minimum rotation speed in radians per second
46          self.min_angular_speed = rospy.get_param("~min_angular_speed", 0.0)
47
48          # The max linear speed in meters per second
49          self.max_linear_speed = rospy.get_param("~max_linear_speed", 0.3)
50
51          # The minimum linear speed in meters per second
52          self.min_linear_speed = rospy.get_param("~min_linear_speed", 0.1)
53
54          # Publisher to control the robot's movement
55          self.cmd_vel_pub = rospy.Publisher('/cmd_vel', Twist)
56
57          rospy.Subscriber('point_cloud', PointCloud2, self.set_cmd_vel)
58
59          # Wait for the point cloud topic to become available
60          rospy.wait_for_message('point_cloud', PointCloud2)
61
62      def set_cmd_vel(self, msg):
63          # Initialize the centroid coordinates and point count
64          x = y = z = n = 0
65
66          # Read in the x, y, z coordinates of all points in the cloud
67          for point in point_cloud2.read_points(msg, skip_nans=True):
68              pt_x = point[0]
69              pt_y = point[1]
70              pt_z = point[2]
71
```

```
72                # Keep only those points within our designated boundaries
                  #  and sum them up
73                if -pt_y > self.min_y and -pt_y < self.max_y and  pt_x <
self.max_x and pt_x > self.min_x and pt_z < self.max_z:
74                    x += pt_x
75                    y += pt_y
76                    z += pt_z
77                    n += 1
78
79            # Stop the robot by default
80            move_cmd = Twist()
81
82            # If we have points, compute the centroid coordinates
83            if n:
84                x /= n
85                y /= n
86                z /= n
87
88                # Check our movement thresholds
89                if (abs(z - self.goal_z) > self.z_threshold) or (abs(x) >
self.x_threshold):
90                    # Compute the linear and angular components of the movement
91                    linear_speed = (z - self.goal_z) * self.z_scale
92                    angular_speed = -x * self.x_scale
93
94                    # Make sure we meet our min/max specifications
95                    linear_speed = copysign(max(self.min_linear_speed,
96                                                min(self.max_linear_speed,
abs(linear_speed))), linear_speed)
97                    angular_speed = copysign(max(self.min_angular_speed,
98                                                 min(self.max_angular_speed,
abs(angular_speed))), angular_speed)
99
100                   move_cmd.linear.x = linear_speed
101           move_cmd.angular.z = angular_speed
102
103       # Publish the movement command
104       self.cmd_vel_pub.publish(move_cmd)
105
106
107   def shutdown(self):
108       rospy.loginfo("Stopping the robot...")
109       self.cmd_vel_pub.publish(Twist())
110       rospy.sleep(1)
111
112   if __name__ == '__main__':
113       try:
114           Follower()
115           rospy.spin()
116       except rospy.ROSInterruptException:
117           rospy.loginfo("Follower node terminated.")
```

The overall strategy behind the script is fairly simple. First, sample all the points in the depth cloud that lie within a search box in front of the robot. From those points, compute the centroid of the region; i.e. the average x, y and z value for all the points. If

there is a person in front of the robot, the z-coordinate of the centroid tells us how far away they are and the x-coordinate reflects whether they are to the right or left. From these two numbers we can compute an appropriate `Twist` message to keep the robot near the person.

Let's now look at the key lines of the script.

```
4  from roslib import message
5  from sensor_msgs import point_cloud2
6  from sensor_msgs.msg import PointCloud2
```

To access the points in the depth cloud, we need the `message` class from `roslib` and the `point_cloud2` library from the ROS `sensor_msgs` package. We also need the `PointCloud2` message type.

The long list of parameters should be fairly self-explanatory from the comments in the code. The heart of the script is the `set_cmd_vel()` callback on the point cloud topic:

```
56          rospy.Subscriber('point_cloud', PointCloud2, self.set_cmd_vel)
```

Note how we use a generic topic name ('`point_cloud`') in the `Subscriber` statement. This allows us to remap the cloud topic in the launch file. Typically we will use the topic `/camera/depth/points` but if we filter the cloud beforehand, we might use a different topic.

```
61      def set_cmd_vel(self, msg):
62          # Initialize the centroid coordinates and point count
63          x = y = z = n = 0
```

Each time we receive a message on the `point_cloud` topic, we begin the `set_cmd_vel` callback by zeroing the centroid coordinates and point count.

```
66          for point in point_cloud2.read_points(msg, skip_nans=True):
67              pt_x = point[0]
68              pt_y = point[1]
69              pt_z = point[2]
```

Here we use the `point_cloud2` library to cycle through all the points in the cloud. The `skip_nans` parameter is handy since a `NaN` (not a number) can occur when the point is inside or outside the camera's depth range.

```
72              if -pt_y > self.min_y and -pt_y < self.max_y and  pt_x <
self.max_x and pt_x > self.min_x and pt_z < self.max_z:
73                  x += pt_x
74                  y += pt_y
75                  z += pt_z
76                  n += 1
```

For each point in the cloud message, we check to see if it falls within the search box. If so, add its x, y and z coordinates to the centroid sums and increment the point count.

```
79          move_cmd = Twist()
```

Initialize the movement command to the empty `Twist` message which will stop the robot by default.

```
82          if n:
83              x /= n
84              y /= n
85              z /= n
```

Assuming we found at least one non-NaN point, we compute the centroid coordinates by dividing by the point count. If there is a person in front of the robot, these coordinates should give us a rough idea of how far away they are and whether they are to the left or right.

```
88              if (abs(z - self.goal_z) > self.z_threshold) or (abs(x) >
self.x_threshold):
89
90                  linear_speed = (z - self.goal_z) * self.z_scale
91                  angular_speed = -x * self.x_scale
```

If the person is closer or further than the goal distance by more than the `z_threshold`, or the left/right displacement exceeds the `x_threshold`, set the robot's linear and angular speed appropriately, weighing each by the `z_scale` and `x_scale` parameters.

```
103     self.cmd_vel_pub.publish(move_cmd)
```

Finally, publish the movement command to move (or stop) the robot.

11.4.3 Running the Follower Application on a TurtleBot

If you have a TurtleBot, you can compare our Python follower node to the C++ version by Tony Pratkanis which is found in the `turtlebot_follower` package. In either case, start with your robot in the middle of a large room as far away from walls, furniture, or other people as possible.

Make sure the TurtleBot is powered on, then launch the startup file:

```
$ roslaunch rbx1_bringup turtlebot_minimal_create.launch
```

Next, bring up the depth camera:

```
$ roslaunch rbx1_vision openni_node.launch
```

Fire up the `follower` node using the `follower.launch` file:

```
$ roslaunch rbx1_apps follower.launch
```

Then move in front of the robot to see if you can get it to follow you. You can adjust the robot's behavior by changing the parameters in the launch file. Note that if you move too close to a wall or another object, the robot will likely lock onto it instead of you and you might have to pick up the robot and rotate it way from the distracting object. Also note that dark clothing, especially black, tends not to reflect the IR pattern used by depth cameras such as the Kinect or Xtion Pro. So if you find the robot is not following you very well, make sure you aren't wearing black pants.

If you find that the robot is a little slow to track your movements, it is likely because of the load we are putting on the CPU by checking *every* point in the point cloud to see if it falls within our tracking ranges. And we are doing this checking in Python which is not particularly suitable for such a task. A faster and more efficient way to run the follower program is to first filter the cloud using a number of PCL nodelets which are written in C++ and run much faster than a bunch of Python if-then statements. This is the approach we take in the next section and it should result in noticeably improved responsiveness during the robot's following behavior.

11.4.4 Running the Follower Node on a Filtered Point Cloud

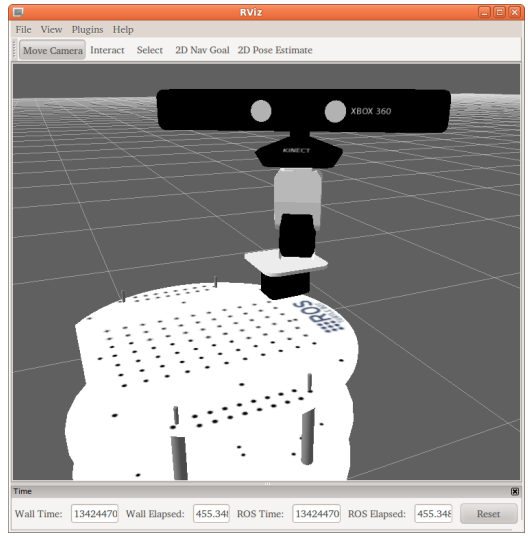
In our current `follower.py` script, we test every point in the cloud to see if it falls within the search box. You might be wondering, why not use the `PassThrough` nodelet we learned about earlier to pre-filter the cloud so this test isn't necessary? Indeed, this is a viable alternative and an implementation can be found in the `follower2.launch` file. This file runs a `VoxelGrid` filter plus a number of `PassThrough` filters to create the search box, then launches the `follower2.py` node. This new script is very similar to the original but now we can skip the test to see if a point falls within the search box.

By this point in the book, you can probably follow both the `follower2.launch` file and the `follower2.py` script on your own so we will leave it to the reader as an exercise. To test the script on a TurtleBot, follow the same instructions as in the previous section but run the `follower2.launch` file instead of `follower.launch`. If you are starting from scratch, run the following three launch files:

```
$ roslaunch rbx1_bringup turtlebot_minimal_create.launch
$ roslaunch rbx1_vision openni_node.launch
$ roslaunch rbx1_apps follower2.launch
```

12. Dynamixel Servos and ROS

It is hard to beat a pan-and-tilt head for adding life-like behavior to your robot. With just two servos, the robot can track your face or other moving objects, or simply look around without having to rotate its entire body. Indeed, you don't even need a robot for this chapter: just a camera mounted on a pair of servos.



A number of third-party ROS packages provide all the tools we need to get started with joint control. At the time of this writing, our only choice of servo is the Robotis Dynamixel, primarily because these servos provides real-time feedback for position, speed and torque which is key to the way that ROS manages robot joints. A pair of the low-end AX-12 servos are generally sufficient to support the weight of a Kinect or Asus Xtion Pro camera. However, the Kinect is heavy enough that the tilt servo should not be held too far away from the vertical position for more than a few minutes at a time to prevent overheating.

In this chapter we will cover the following topics:

- Adding a pan-and-tilt head to a TurtleBot
- Choosing a Dynamixel controller and ROS package
- Understanding the ROS `JointState` message type
- Using basic joint control commands
- Connecting the hardware controller and setting the servo IDs
- Configuring the launch files and parameters
- Testing the servos
- Programming a head tracking node
- Combining head tracking with face tracking

12.1 A TurtleBot with a Pan-and-Tilt Head

For this chapter, we will use a modified TurtleBot with the Kinect placed on top of a pair of pan and tilt Dynamixel servos as shown in the picture above. (An Asus Xtion would work as well.)

If you have a TurtleBot and would like to add a pan-and-tilt head using Dynamixel servos, you can either build your own using Robotis brackets or, if you have more money than time, you might pick up a PhantomX Turret Kit from Trossen Robotics. Note that this kit can be purchased without servos and without the ArbotiX controller. We will be using a USB2Dynamixel controller instead.

To test the URDF model of the modified TurtleBot, make sure you terminate any other launch files that might be running, then run the following commands:

```
$ roslaunch rbx1_description test_turtlebot_with_head.launch
```

and then in another terminal:

```
$ rosrun rviz rviz -d `rospack find rbx1_description`/urdf.rviz
```

If all goes well, you should see the robot in `RViz` as it appears in the picture at the start of the chapter. In addition, a small pop-up window will appear with a number of slider controls that allow you to test the movement of the servos. Try moving the sliders and make sure the servos rotate the head appropriately. (See if you can use the right-hand rule to figure out why a positive tilt rotates the camera downward and a positive pan causes the head to rotate counter-clockwise.)

Type `Ctrl-C` in the `test_turtlebot_with_head.launch` window when you are finished.

12.2 Choosing a Dynamixel Hardware Controller

There are two Dynamixel hardware controllers that work well with ROS: the ArbotiX controller from Vanadium Labs used on the Mini Max and Maxwell robots, and the Robotis USB2Dynamixel controller used with the Crustcrawler Smart Arm. We will use the USB2Dynamixel controller since we don't need the onboard base controller that is also provided by the ArbotiX. You can find existing tutorials for using the ArbotiX controller on the Mini Max Wiki page.

You will also need a way to power the servos. The USB2Dynamixel controller does not itself have a power connection so you will need something like the SMPS2Dynamixel together with a 12V power supply.

12.3 A Note Regarding Dynamixel Hardware

While Dynamixel servos are very sophisticated pieces of hardware, the cable connections between the servos need to be secured with tie wraps or other methods to ensure reliability. Even a slight movement of the connectors in the sockets can cause a loss of signal which in turn can cause the software drivers to crash. In such cases, power cycling the servos and restarting the driver is often necessary. The same precautions should be taken for the power connection between the battery source and the SMPS2Dynamixel device.

12.4 Choosing a ROS Dynamixel Package

The two most actively developed ROS Dynamixel projects are the arbotix stack by Michael Ferguson and the dynamixel_motor stack by Antons Rebguns. Both stacks can be used with a USB2Dynamixel controller while the `arbotix` stack can also be used with the ArbotiX controller. However, we will use the `dynamixel_motor` stack to be consistent with earlier tutorials on the Pi Robot website.

To install the `dynamixel_motor` stack, run the command:

```
$ sudo apt-get install ros-hydro-dynamixel-motor
```

That's all there is to it!

12.5 Understanding the ROS `JointState` Message Type

The ROS `JointState` message type is designed to keep track of all of the joints on a robot. As with all message types, we can display its fields using the `rosmsg` command:

```
$ rosmsg show sensor_msgs/JointState
```

which should produce the following output:

```
Header header
  uint32 seq
  time stamp
  string frame_id
string[] name
float64[] position
float64[] velocity
float64[] effort
```

As you can see, the `JointState` message includes a standard ROS `Header` component made up of a sequence number (`seq`), a timestamp (`stamp`) and a `frame_id`. This is followed by four arrays: a string array called `name` to hold the names of the joints, then

three float arrays to hold the `position`, `velocity` and `effort` (usually torque) for each joint.

We can now see why Dynamixel servos are a good fit with ROS: namely, we can query each servo for its current position, velocity and torque which is exactly the data we need to fill in the `JointState` message. Once we have chosen a hardware controller for the Dynamixel servos, a ROS driver for that controller would be expected to publish the servo states as a `JointState` message on some topic. This topic is typically called `/joint_states` and other nodes can subscribe to the topic to find out the current state of all the joints.

While most joints used on robots are *revolute* like servos, ROS can also deal with linear joints which are referred to as *prismatic*. A prismatic joint can be used to move a robot's torso up and down like on the PR2, or to extend or retract an arm. In either case, the position, velocity and torque of a prismatic joint can be specified in the same way as a servo-type joint and so ROS can easily handle both.

12.6 Controlling Joint Position, Speed and Torque

Setting the position, speed and torque of a servo or linear joint in ROS is done using topics and services as we would expect. The `dynamixel_motor` stack contains the `dynamixel_controllers` package that works in a manner similar to the one used on the Willow Garage PR2: first a controller manager node is launched that connects to the Dynamixel bus (in our case a USB2Dynamixel controller). The controller node then launches a number of individual controllers, one for each servo.

Each controller uses a topic to control the servo's position and a collection of services for setting the servo's speed, torque, and other Dynamixel properties. Fortunately, the authors of the `dynamixel_motor` stack (Antons Rebguns, Cody Jorgensen and Cara Slutter) have nicely documented their package on the ROS Wiki. The part of the documentation we are interested in right now can be found on the `dynamixel_controllers` page. Click on the link and then click on the **electric** button at the top of the page to see the documentation. (For some reason, the full documentation is not showing up on the Groovy and Hydro Wiki pages.) Finally, click on the link called "Common Joint Controller Interface" in the table of contents. Here is a screen shot of that part of the page:

2.4 Common Joint Controller Interface

2.4.1 Subscribed Topics

`<joint_controller_name>/command` (std_msgs/Float64)

Listens for a joint angle (in radians) to be sent to the controller.

`motor_states/<serial_port_name>` (dynamixel_msgs/MotorStateList)

Listens for motor status feedback published by low level driver.

2.4.2 Published Topics

`<joint_controller_name>/state` (dynamixel_msgs/JointState)

Provides current joint status information (current goal, position, velocity, load, etc.)

2.4.3 Services

`<joint_controller_name>/set_speed` (dynamixel_controllers/SetSpeed)

Change the current velocity of the joint (specified in radians per second).

`<joint_controller_name>/torque_enable` (dynamixel_controllers/TorqueEnable)

Turn joint torque on or off.

`<joint_controller_name>/set_compliance_slope` (dynamixel_controllers/SetComplianceSlope)

Change the level of torque near goal position (see Dynamixel documentation for more details).

`<joint_controller_name>/set_compliance_margin` (dynamixel_controllers/SetComplianceMargin)

Change allowable error between goal position and present position (see Dynamixel documentation for more details).

`<joint_controller_name>/set_compliance_punch` (dynamixel_controllers/SetCompliancePunch)

Change minimum amount of torque at goal position (see Dynamixel documentation for more details).

`<joint_controller_name>/set_torque_limit` (dynamixel_controllers/SetTorqueLimit)

Change the maximum amount of torque (see Dynamixel documentation for more details).

The topics and services listed here allow us to control many aspects of the servos. The ones we are interested in for now are setting position, speed and torque so let's look at them next.

12.6.1 Setting Servo Position

To set a servo's position in radians from center, we publish the desired position on the topic called:

`<joint_controller_name>/command`

So if the controller for moving a head pan servo is called `head_pan_joint`, then we would use the following command to set the position of the servo to 1.0 radians clockwise from center:

```
$ rostopic pub -1 /head_pan_joint/command std_msgs/Float64 -- -1.0
```

The pair of hyphens (`--`) before the position value is required when publishing a negative value so that `rostopic` does not think we are supplying an option like "`-r`". We can also use them when publishing positive values but they are not required. To position the servo 1.0 radians *counter*-clockwise from center, we would use the command:

```
$ rostopic pub -1 /head_pan_joint/command std_msgs/Float64 -- 1.0
```

or without the hyphens:

```
$ rostopic pub -1 /head_pan_joint/command std_msgs/Float64 1.0
```

12.6.2 Setting Servo Speed

To set a servo's speed in radians per second, use the `set_speed` service called:

`<joint_controller_name>/set_speed`

So to set the speed of of the head pan servo to 0.5 radians per second, we would use the command:

```
$ rosservice call /head_pan_joint/set_speed 0.5
```

12.6.3 Controlling Servo Torque

The `dynamixel_controllers` package provides two services related to torque: `torque_enable` and `set_torque_limit`. The `torque_enable` service allows us to relax the torque completely or to turn it back on. Starting your robot with the servos in the relaxed state conveniently allows you to position the joints by hand before running any tests.

To relax the head pan servo, we would use the following command:

```
$ rosservice call /head_pan_joint/torque_enable False
```

And to turn it back on:

```
$ rosservice call /head_pan_joint/torque_enable True
```

The `set_torque_limit` service allows you to set how hard you want the servo to work against a load. For example, if your robot has a multi-jointed arm, you might want the torque limits set just high enough to lift the arm's own weight. This way if the arm bumps against a person or object, the servos won't mindless attempt to push through the obstacle. The following command would set the head pan servo torque limit to 0.1:

```
$ rosservice call /head_pan_joint/set_torque_limit 0.1
```

This is a relatively low limit and when we try it out on a real servo later in the chapter, we will find that you can still rotate the servo by hand but now it will rotate back to its starting position when you let go.

12.7 Checking the USB2Dynamixel Connection

To check the connection to the USB2Dynamixel controller, first make sure that the micro switch on the side of the device is moved to the correct setting. For 3-pin AX-12, AX-18 or the new "T" series servos (e.g. MX-28T), you want the TTL setting. For any of the 4-pin or "R" series servos (e.g. MX-28R, RX-28, EX-106+), you need the RS-485 setting.

Next, unplug any other USB devices from your computer if possible, then connect your controller to a USB port. You can also connect it to a USB hub that is connected to your computer. Once connected, a red LED on the controller should illuminate. Then issue the following command to see what USB ports you have connected:

```
$ ls /dev/ttyUSB*
```

Hopefully you will see something like the following output:

```
/dev/ttyUSB0
```

If instead you get back the message:

```
ls: cannot access /dev/ttyUSB*: No such file or directory
```

then your USB2Dynamixel has not been recognized. Try plugging it into a different USB port, use a different cable, or check your USB hub.

If you have no other USB devices attached, your USB2Dynamixel should be on `/dev/ttyUSB0`. If you need to have other USB devices connected at the same time, plug in the USB2Dynamixel first so that it gets assigned the device `/dev/ttyUSB0`. If it has to be on a different device, e.g. `/dev/ttyUSB1`, simply make a note of it for the following configuration sections.

12.8 Setting the Servo Hardware IDs

If you have already set the hardware IDs of your Dynamixel servos, you can skip this section. Otherwise, read on.

All Dynamixels are shipped with an ID of 1 so if you are using more than one servo on the bus, at least one ID has to be changed. In the case of our pan-and-tilt head, we'll assume we want the ID for the pan servo to be 1 and 2 for the tilt servo. You can set them to be whatever you like but be sure to remember your choices for the next section on configuration.

If both servos still have their default value of 1, connect the tilt servo to the bus on its own. In other words, disconnect the pan servo from the bus if is already connected. Then power up the servo. Assuming your USBDynamixel is still plugged into your computer from the previous section, bring up the `arbotix_terminal` application as follows:

```
$ arbotix_terminal /dev/ttyUSB0 1000000
```

Note the device name `/dev/ttyUSB0` on the command line. If your controller is using a different device, such as `/dev/ttyUSB1`, use that instead. The second parameter is the baud rate of the USB2Dynamixel which is always `1000000`.

If all goes well, you should see the following on your screen:

```
ArbotiX Terminal --- Version 0.1
Copyright 2011 Vanadium Labs LLC
>>
```

To list the servos on the bus, run the `ls` command at the `>>` prompt. Hopefully your screen will then look something like this:

```
ArbotiX Terminal --- Version 0.1
Copyright 2011 Vanadium Labs LLC
>>  ls
   1 .... .... .... .... .... .... .... ....
.... .... .... .... .... .... .... .... ....
>>
```

Notice the 1 before all the …. characters. This indicates that a servo with an ID of 1 was found on the bus. If no IDs are shown, run the `ls` command a second time. If still no IDs are displayed, check the connection between your servo and the USB2Dynamixel controller. Also double-check that the servo has power. If all else fails, try replacing the USB cable between your PC and the USB2Dynamixel controller. (This happened to me once.)

To change this servo's ID from 1 to 2, use the `mv` command:

```
>>  mv 1 2
```

then issue the `ls` command again. If all goes well, you should see that the servo now has ID 2:

```
>>  ls
....    2 .... .... .... .... .... .... ....
.... .... .... .... .... .... .... .... ....
```

Next, disconnect the tilt servo and connect the pan servo instead, again ensuring that it has power. Run the `ls` command to find its current ID. If it is already set to 1, you can leave it as is. Otherwise, use the `mv` command to set it to 1.

Finally, connect both servos at the same time and run the `ls` command. The result should be:

```
>>  ls
   1    2 .... .... .... .... .... .... ....
.... .... .... .... .... .... .... .... ....
```

To exit the `arbotix_terminal` program, type `Ctrl-C`.

12.9 Configuring and Launching `dynamixel_controllers`

In this section we will learn how to interpret the servo configuration file and examine the `dynamixel_controllers` launch file.

12.9.1 The `dynamixel_controllers` Configuration File

The servo configuration parameters are store in the file `dynamixel_params.yaml` found in the `rbx1_dynamixels/config` subdirectory and looks like this:

```
joints: ['head_pan_joint', 'head_tilt_joint']

head_pan_joint:
    controller:
        package: dynamixel_controllers
        module: joint_position_controller
        type: JointPositionController
    joint_name: head_pan_joint
    joint_speed: 2.0
    motor:
        id: 1
        init: 512
        min: 0
        max: 1024

head_tilt_joint:
    controller:
        package: dynamixel_controllers
        module: joint_position_controller
```

```
        type: JointPositionController
    joint_name: head_tilt_joint
    joint_speed: 2.0
    motor:
        id: 2
        init: 512
        min: 300
        max: 800
```

First we define a list parameter called `joints` that contains the names of our servos. Next, we have a block for each servo controller beginning with the controller name. In our case, the two controllers are called `head_pan_joint` and `head_tilt_joint`. These are the names that are used in the topics and services we learned about in the previous section.

For each servo controller, we specify the type of controller (`JointPositionController`), as well as its hardware ID, the initial position value and its minimum and maximum position values. If your servo IDs are different than 1 and 2, edit this file accordingly.

The `init/min/max` numbers are given in servo ticks which vary from 0 to 1023 for the AX-12's. In the configuration above, we give the head tilt controller less than full range since it cannot go all the way forward or back without hitting the mounting plate. (We also specify these limits in the robot's URDF file but using radians instead of servo ticks.)

12.9.2 The `dynamixel_controllers` Launch File

The file `dynamixels.launch` in the `rbx1_dynamixels/launch` directory illustrates how to fire up the servo controllers when we have a USB2Dynamixel controller on device `/dev/ttyUSB0` and two Dynamixel servos on the bus with hardware IDs 1 and 2. Let's take a look at it now:

```
<launch>
    <param name="/use_sim_time" value="false" />

    <!-- Load the URDF/Xacro model of our robot -->
       <param name="robot_description" command="$(find xacro)/xacro.py '$(find
rbx1_description)/urdf/turtlebot_with_head.xacro'" />

    <!-- Publish the robot state -->
       <node name="robot_state_publisher" pkg="robot_state_publisher"
type="state_publisher">
       <param name="publish_frequency" value="20.0"/>
    </node>

   <!-- Start the Dynamixel low-level driver manager with parameters -->
   <node name="dynamixel_manager" pkg="dynamixel_controllers"
      type="controller_manager.py" required="true" output="screen">
      <rosparam>
         namespace: turtlebot_dynamixel_manager
         serial_ports:
```

```
            dynamixel_ax12:
               port_name: "/dev/ttyUSB0"
               baud_rate: 1000000
               min_motor_id:  1
               max_motor_id: 2
               update_rate: 20
      </rosparam>
   </node>

    <!-- Load the joint controller configuration from a YAML file -->
    <rosparam file="$(find rbx1_dynamixels)/config/dynamixel_params.yaml"
command="load"/>

   <!-- Start the head pan and tilt controllers -->
   <node name="dynamixel_controller_spawner_ax12" pkg="dynamixel_controllers"
   type="controller_spawner.py"
        args="--manager=turtlebot_dynamixel_manager
              --port=dynamixel_ax12
              --type=simple
        head_pan_joint
        head_tilt_joint"
        output="screen" />

   <!-- Start the Dynamixel Joint States Publisher -->
   <node name="dynamixel_joint_states_publisher" pkg="rbx1_dynamixels"
type="dynamixel_joint_state_publisher.py" output="screen" />

   <!-- Start all Dynamixels in the relaxed state -->
   <node pkg="rbx1_dynamixels" type="relax_all_servos.py"
name="relax_all_servos" />

</launch>
```

If your USB2Dynamixel controller is on a device other than `/dev/ttyUSB0` and/or your servo IDs are other than 1 and 2, edit this file now before going any further.

The launch file first loads the URDF model for the robot and runs a `robot_state_publisher` node to publish the state of the robot to `tf`. We then launch the `controller_manager.py` node, load the `dynamixels_param.yaml` file we looked at earlier, then spawn a pair of joint controllers, one for each servo.

The next node launched is called `dynamixel_joint_state_publisher.py` and is found in the `rbx1_dynamixels/nodes` directory. This is not part of the `dynamixel_controllers` package but is necessary to correct one inconsistency in the way the package publishes joint states. Rather than use the standard ROS `JointState` message type that we introduced earlier, the `dynamixel_controllers` package uses a custom message type to publish the joint states including some additional useful information such as servo temperature. If you look at the code in the `dynamixel_joint_state_publisher.py` node, you will see that it simply republishes the custom joint state information as a standard `JointState` message on the `/joint_states` topic.

Finally, the launch file runs the `relax_all_servos.py` node found in the `rbx1_dynamixels/nodes` directory which relaxes the torque on each servo and sets a reasonable default speed and torque limit. (More on this below.)

12.10 Testing the Servos

To test the pan and tilt servos, first connect your servos and USB2Dynamixel to a power source, then make sure your USB2Dynamixel is connected to a USB port on your computer.

12.10.1 Starting the Controllers

Before running the next command, be sure to terminate the `test_turtlebot_with_head.launch` file we used in the previous section if you haven't already.

Next fire up the `dynamixels.launch` file. This launch file also loads the URDF model for the TurtleBot with a Kinect on pan and tilt servos:

```
$ roslaunch rbx1_dynamixels dynamixels.launch
```

You should see a number of startup messages that look something like this:

```
process[robot_state_publisher-1]: started with pid [11415]
process[dynamixel_manager-2]: started with pid [11416]
process[dynamixel_controller_spawner_ax12-3]: started with pid
[11417]
process[fake_pub-4]: started with pid [11418]
process[dynamixel_joint_states_publisher-5]: started with pid
[11424]
process[relax_all_servos-6]: started with pid [11426]
process[world_base_broadcaster-7]: started with pid [11430]
[INFO] [WallTime: 1340671865.017257] Pinging motor IDs 1 through
2...
[INFO] [WallTime: 1340671865.021896] Found motors with IDs: [1,
2].
[INFO] [WallTime: 1340671865.054116] dynamixel_ax12
controller_spawner: waiting for controller_manager
turtlebot_dynamixel_manager to startup in global namespace...
[INFO] [WallTime: 1340671865.095946] There are 2 AX-12+ servos
connected
[INFO] [WallTime: 1340671865.096249] Dynamixel Manager on port
/dev/ttyUSB0 initialized
[INFO] [WallTime: 1340671865.169167] Starting Dynamixel Joint
State Publisher at 20Hz
```

```
[INFO] [WallTime: 1340671865.363797] dynamixel_ax12
controller_spawner: All services are up, spawning controllers...
[INFO] [WallTime: 1340671865.468773] Controller head_pan_joint
successfully started.
[INFO] [WallTime: 1340671865.530030] Controller head_tilt_joint
successfully started.
[dynamixel_controller_spawner_ax12-3] process has finished
cleanly.
```

NOTE: If one of the startup messages gives an error containing the message:

```
serial.serialutil.SerialException: could not open port
/dev/ttyUSB0:
```

then try the following. First, power-cycle the USB2Dynamixel controller (unplug the power supply and plug it back it again), then unplug the USB cable from your computer and plug it back in again. This will fix this error 9 times out of 10. Also double-check that your USB2Dynamixel really is connected to `/dev/ttyUSB0` and not some other port such as `/dev/ttyUSB1`.

12.10.2 Monitoring the Robot in `RViz`

To view the robot in `RViz` and observe the motion of the servos as we start testing below, bring up `RViz` now as follows:

```
$ rosrun rviz rviz -d `rospack find rbx1_dynamixels`/dynamixels.rviz
```

12.10.3 Listing the Controller Topics and Monitoring Joint States

Once the `dynamixels.launch` file is up and running, bring up another terminal and list the active topics:

```
$ rostopic list
```

Among the topics listed, you should see the following dynamixel-related topics:

```
/diagnostics
/head_pan_joint/command
/head_pan_joint/state
/head_tilt_joint/command
/head_tilt_joint/state
/joint_states
/motor_states/dynamixel_ax12
```

For now, the topics we will look at are:

```
/head_pan_joint/command
/head_tilt_joint/command
/head_pan_joint/state
/head_tilt_joint/state
/joint_states
```

The two `command` topics are for setting the the servo positions as we saw earlier and we will test them below. The `/joint_states` topic (which is actually published by our auxiliary node `dynamixel_joint_state_publisher.py`) holds the current state of the servos as a `JointState` message. Use the `rostopic echo` command to see the messages:

```
$ rostopic echo /joint_states
```

which should yield a stream of messages that look something like the following:

```
header:
  seq: 11323
  stamp:
    secs: 1344138755
    nsecs: 412811040
  frame_id: 'base_link'
name: ['head_pan_joint', 'head_tilt_joint']
position: [-0.5266667371740702, 0.08692557798018634]
velocity: [0.0, 0.0]
effort: [0.0, 0.0]
```

Here we see the header with the sequence number and timestamp, a `frame_id` set to `base_link` and then the four arrays for `name`, `position`, `velocity` and `effort`. Since our servos are not moving, the velocities are 0 and since they are not under load, the effort (torque) is 0.

While the messages are still streaming on the screen, try moving the servos by hand. You should immediately see the position, velocity and effort values change to match your movements. You should also see the movements in `RViz`.

Returning to the full list of dynamixel topics above, the two controller `state` topics are useful for monitoring the servo temperatures and other parameters. Try viewing the messages on the `state` topic for the head pan controller:

```
$ rostopic echo /head_pan_joint/state
```

A typical message looks like this:

```
header:
  seq: 25204
  stamp:
    secs: 1344221332
    nsecs: 748966932
  frame_id: ''
name: head_pan_joint
motor_ids: [1]
motor_temps: [37]
goal_pos: 0.0
current_pos: -0.00511326929295
error: -0.00511326929295
velocity: 0.0
load: 0.0
is_moving: False
```

Perhaps the most important number here is the servo temperature listed in the `motor_temps` field. A value of 37 degrees is fairly typical for a resting AX-12 servo. When the temperature starts getting up around 50 degrees or higher, it's probably a good idea to give it rest. To visually monitor the temperates of both servos, you could keep a running plot using `rqt_plot`:

```
$ rqt_plot /head_pan_joint/state/motor_temps[0], \
  /head_tilt_joint/state/motor_temps[0]
```

(We have to index the `motor_temps` fields since the `dynamixel_controllers` package uses an array variable for this field to accommodate joints that use more than one motor.)

Since you now know the topics that include servo temperatures, you could subscribe to these topics in your scripts and relax the servos or move them to a neutral position if the temperature gets too high. A more sophisticated approach uses ROS diagnostics which is outside the scope of this volume.

12.10.4 Listing the Controller Services

We can also list the services used by the `dynamixel_controllers` package. To list all currently active services, use the `rosservice list` command:

```
$ rosservice list
```

Among the topics listed, you should see the following `dynamixel_controllers` services:

```
/head_pan_joint/set_speed
/head_pan_joint/set_torque_limit
/head_pan_joint/torque_enable
/head_tilt_joint/set_speed
/head_tilt_joint/set_torque_limit
/head_tilt_joint/torque_enable
```

(There are quite a few additional services provided by `dynamixel_controllers` but we will not be using them in this book.)

These are just the services we have already met in an earlier section. Let's now test the position command topic and these services on the live servos.

12.10.5 Setting Servo Position, Speed and Torque

Once the dynamixel controllers are up and running, bring up a new terminal and send a couple of simple pan and tilt commands. The first command should pan the head fairly slowly to the left (counter-clockwise) through 1 radian or about 57 degrees:

```
$ rostopic pub -1 /head_pan_joint/command std_msgs/Float64 -- 1.0
```

Re-center the servo with the command:

```
$ rostopic pub -1 /head_pan_joint/command std_msgs/Float64 -- 0.0
```

Now try tilting the head downward half a radian (about 28 degrees):

```
$ rostopic pub -1 /head_tilt_joint/command std_msgs/Float64 -- 0.5
```

And bring it back up:

```
$ rostopic pub -1 /head_tilt_joint/command std_msgs/Float64 -- 0.0
```

To change the speed of the head pan servo to 1.0 radians per second, use the `set_speed` service:

```
$ rosservice call /head_pan_joint/set_speed 1.0
```

Then try panning the head again at the new speed:

```
$ rostopic pub -1 /head_pan_joint/command std_msgs/Float64 -- -1.0
```

To relax a servo so that you can move it by hand, use the `torque_enable` service:

```
$ rosservice call /head_pan_joint/torque_enable False
```

Now try rotating the head by hand. Note that relaxing a servo does not prevent it from moving when it receives a new position command. For example, re-issue the last panning command:

```
$ rostopic pub -1 /head_pan_joint/command std_msgs/Float64 -- -1.0
```

The position command will automatically re-enable the torque and move the servo. To re-enable the torque manually, run the command:

```
$ rosservice call /head_pan_joint/torque_enable True
```

And try again rotating the servo by hand (but don't force it!)

Finally, set the torque limit to a small value:

```
$ rosservice call /head_pan_joint/set_torque_limit 0.1
```

Now try rotating the head by hand. This time you should feel a little more resistance compared to a completely relaxed servo. Furthermore, when you release your grip, the servo will return to the position it held before you moved it.

Now set the torque limit back to a moderate value:

```
$ rosservice call /head_pan_joint/set_torque_limit 0.5
```

Note: If the torque limit is set very low, you will find that there is also a limit to how fast you can move the servo regardless of how high you set the speed using the `set_speed` service.

12.10.6 Using the `relax_all_servos.py` Script

To relax all the servos at once, use the `relax_all_servos.py` utility script which is included in the `rbx1_dynamixels/nodes` directory:

```
$ rosrun rbx1_dynamixels relax_all_servos.py
```

If you examine the `relax_all_servos.py` script, you will see that relaxing is done by using a `torque_enable(False)` service call to each servo. The `set_speed` service is also used to set the servo speeds to a relatively low value (0.5 rad/s) so that we are not surprised by a fast motion the first time a position command is sent. Finally, the `set_torque_limit` service is called on each servo to set the maximum torque to a moderate value (0.5). You can edit this script to set your own defaults as desired.

After running the script, sending a position command to a servo will automatically re-enable its torque so there is no need to explicitly turn the torque back on when you want to move the servo.

The `relax_all_servos.py` script is run from the `dynamixels.launch` file we used earlier to start up the servo controllers.

12.11 Tracking a Visual Target

In the chapter on Robot Vision, we developed a number of nodes for tracking a visual target including faces, keypoints, and colors. We now have all the ingredients we need to program a head tracking node that will move the robot's pan-and-tilt camera to follow a moving object.

The script that does the job is called `head_tracker.py` and can be found in the `rbx1_dynamixels/nodes` directory. Before looking at the code, you can try it out with our earlier face tracking node.

12.11.1 Tracking a Face

First make sure your camera driver is up and running. For a Kinect or Xtion Pro:

```
$ roslaunch rbx1_vision openni_node.launch
```

or if you are using a webcam:

```
$ roslaunch rbx1_vision uvc_cam.launch device:=/dev/video0
```

(Change the video device if necessary.)

Next, make sure your servos are powered up and the USB2Dynamixel controller is plugged into a USB port. If you have already launched the `dynamixels.launch` file, you can skip this step:

```
$ roslaunch rbx1_dynamixels dynamixels.launch
```

Make sure your servos are connected by testing the pan servo with the commands:

```
$ rostopic pub -1 /head_pan_joint/command std_msgs/Float64 -- 1.0
$ rostopic pub -1 /head_pan_joint/command std_msgs/Float64 -- 0.0
```

With the servos good to go, fire up the face tracking node we developed earlier:

```
$ roslaunch rbx1_vision face_tracker2.launch
```

When the video window appears, move your face in front of the camera and make sure the tracking is working. Remember that you can hit the '`c`' key to clear tracking and have your face re-detected.

We are now ready to launch the head tracking node with the command:

```
$ roslaunch rbx1_dynamixels head_tracker.launch
```

Assuming the face tracking node still has a lock on your face, the pan and tilt servos should now keep the camera centered on your face as you move in front of the camera. If your face is lost for a long enough period of time, the servos will re-center the camera. If you `Ctrl-C` out of the `head_tracker.launch` process, the servos will also re-center before the node shuts down. If the keypoints in the face tracker window start drifting onto other objects, hit the '`c`' key to clear the points and re-detect your face.

The following video demonstrates the behavior using a printout of the Mona Lisa as the target face: http://youtu.be/KHJL09BTnlY

12.11.2 The Head Tracker Script

The `head_tracker.py` script is rather long but straightforward. The overall process is as follows:

- Initialize the servos.
- Subscribe to the `/roi` topic.
- If the `/roi` moves away from the center of view, command the servos to move the camera in a direction to re-center it.
- If the `/roi` is lost for a given amount of time, re-center the servos to protect them from overheating.

To track the target, the script uses a kind of "speed tracking". If you move the camera to the position where the tracked object is *now*, it may have moved by the time the camera gets there. You might think you could just update the camera's target position at a high rate and therefore keep up with the object. However, this kind of position tracking will result in a jerky staccato-like motion of the camera. A better strategy is to always aim the camera *ahead* of the target, but adjust the servo speeds to be proportional to the displacement of the target from the center of view. This results in much smoother camera motion and will ensure that it moves quickly if the target is far off-center and more slowly if the displacement is small. When the target is centered, the servo speeds will be zero and so the camera will not move.

The `head_tracker.py` script is a little long to display in its entirety, so let's take a look at just the key sections of the code. You can see the entire source file at the following link:

Link to source: `head_tracker.py`

Here now are the key lines.

```
rate = rospy.get_param("~rate", 10)
r = rospy.Rate(rate)
tick = 1.0 / rate

# Keep the speed updates below about 5 Hz; otherwise the servos
# can behave erratically.
speed_update_rate = rospy.get_param("~speed_update_rate", 10)
speed_update_interval = 1.0 / speed_update_rate

# How big a change do we need in speed before we push an update
# to the servos?
self.speed_update_threshold = rospy.get_param("~speed_update_threshold", 0.01)
```

We define two rate parameters near the top of the script. The overall `rate` parameter controls how fast we update the tracking loop which involves changing both the speed and joint angle of the servos depending on the location of the target. The `speed_update_parameter` is typically set lower and defines how often we update the servos speeds. The only reason for doing this is that it turns out that Dynamixel servos can act a little erratically if we try to adjust their speed too often. An update rate less than 10 Hz or so seems to result in better behavior. We also set a `speed_update_threshold` so that we only update the servo speeds if the newly calculated speed differs by this much from the previous speed.

```
self.head_pan_joint = rospy.get_param('~head_pan_joint', 'head_pan_joint')
self.head_tilt_joint = rospy.get_param('~head_tilt_joint', 'head_tilt_joint')

self.joints = [self.head_pan_joint, self.head_tilt_joint]
```

We need to know the name of the pan and tilt joints in the URDF model of the robot. If your joint names differ from the defaults, use these two parameters in the `head_tracker.launch` file to set them accordingly.

```
# Joint speeds are given in radians per second
self.default_joint_speed = rospy.get_param('~default_joint_speed', 0.3)
self.max_joint_speed = rospy.get_param('~max_joint_speed', 0.5)

# How far ahead or behind the target (in radians) should we aim for?
self.lead_target_angle = rospy.get_param('~lead_target_angle', 1.0)

# The pan/tilt thresholds indicate what percentage of the image window
# the ROI needs to be off-center before we make a movement
self.pan_threshold = rospy.get_param('~pan_threshold', 0.025)
self.tilt_threshold = rospy.get_param('~tilt_threshold', 0.025)

# The gain_pan and gain_tilt parameter determine how responsive the
# servo movements are. If these are set too high, oscillation can
# result.
self.gain_pan = rospy.get_param('~gain_pan', 1.0)
self.gain_tilt = rospy.get_param('~gain_tilt', 1.0)

# Set limits on the pan and tilt angles
self.max_pan = rospy.get_param('~max_pan', radians(145))
self.min_pan = rospy.get_param('~min_pan', radians(-145))
```

```
self.max_tilt = rospy.get_param('~max_tilt', radians(90))
self.min_tilt = rospy.get_param('~min_tilt', radians(-90))
```

Next comes a list of parameters for controlling the tracking behavior. Most the parameters are easily understood from the embedded comments. The `gain_pan` and `gain_tilt` parameters control how quickly the servos will respond to a displacement of the target from the camera's field of view. If these are set too high, oscillation will result. If they are set too low, the camera's motion will lag behind a moving target.

```
self.recenter_timeout = rospy.get_param('~recenter_timeout', 5)
```

The `recenter_timeout` parameter determines how long (in seconds) a target can be lost before we recenter the servos. When a target goes out of sight, the `head_tracker.py` script stops the servos so that they are holding the camera in the last position they had before the target was lost. However, this can cause the servos to overheat if the camera is held this way for too long. Re-centering the servos allows them to return to a neutral position and cool down.

```
# Get a lock for updating the self.move_cmd values
self.lock = thread.allocate_lock()
```

Here we allocate a thread lock object and assign it to the variable `self.lock`. We will need this lock to make our overall program thread safe since we will be updating the joint positions and speeds in two places: the main body of the script and the callback function (defined below) assigned to the `/roi` topic. Since ROS spins a separate thread for each subscriber callback, we need to protect our joint updates with a lock as we will show further on.

```
self.init_servos()
```

The initialization of the servos is tucked away in the `init_servos()` function which looks like this.

```
def init_servos(self):
    # Create dictionaries to hold the speed, position and torque controllers
    self.servo_speed = dict()
    self.servo_position = dict()
    self.torque_enable = dict()
```

First we define three Python dictionaries to store the servo controllers for speed, position and torque.

```
    for joint in sorted(self.joints):
```

We then loop through all the joints listed in the `self.joints` parameter. In our case, there are only two servos named `head_pan_joint` and `head_tilt_joint`.

```
            set_speed_service = '/' + joint + '/set_speed'
            rospy.wait_for_service(set_speed_service)
            self.servo_speed[joint] = rospy.ServiceProxy(set_speed_service,
SetSpeed, persistent=True)
```

Recall that the `dynamixel_controller` package uses a `set_speed` service for each servo to set the servo's speed. We therefore connect to the `set_speed` service for each servo controller. Using the `persistent=True` argument in the `ServiceProxy` statement is important. Otherwise `rospy` has to reconnect to the `set_speed` service every time we want to adjust the speed of a servo. Since we will be updating servo speeds continuously during tracking, we want to avoid this connection delay.

```
            self.servo_speed[name](self.default_joint_speed)
```

Once we are connected to the `set_speed` services, we can initialize each servo speed to the default speed.

```
            torque_service = '/' + joint + '/torque_enable'
            rospy.wait_for_service(torque_service)
            self.torque_enable[name] = rospy.ServiceProxy(torque_service,
TorqueEnable)
            # Start each servo in the disabled state so we can move them by hand
            self.torque_enable[name](False)
```

In a similar manner, we connect to the `torque_enable` service for each servo and initialize them to the relaxed state so we can move them by hand if necessary.

```
            self.servo_position[name] = rospy.Publisher('/' + joint + '/command',
Float64)
```

The position controller uses a ROS publisher rather a than service so we define one for each servo. This completes servo initialization.

```
 rospy.Subscriber('roi', RegionOfInterest, self.set_joint_cmd)
```

We assume that the position of the target is published on the `/roi` topic as it will be if we are using any of our earlier vision nodes such as the face tracker or camshift nodes. The callback function `set_joint_cmd()` will set the servo speeds and target positions to track the target.

```
 self.joint_state = JointState()
 rospy.Subscriber('joint_states', JointState, self.update_joint_state)
```

We also keep tabs on the current servo positions by subscribing to the `/joint_states`.

```
 while not rospy.is_shutdown():
```

```
        # Acquire the lock
        self.lock.acquire()

        try:
            # If we have lost the target, stop the servos
            if not self.target_visible:
                self.pan_speed = 0.0
                self.tilt_speed = 0.0

                # Keep track of how long the target is lost
                target_lost_timer += tick
            else:
                self.target_visible = False
                target_lost_timer = 0.0
```

This is the start of the main tracking loop. First we acquire a lock at the beginning of each update cycle. This is to protect the variables `self.pan_speed`, `self.tilt_speed` and `self.target_visible` which are also modified by our callback `set_joint_cmd()`. We use the variable `self.target_visible` to indicate if we have lost the ROI. This variable is set to `True` in the `set_joint_cmd` callback as we shall see below. Otherwise, it defaults to `False`. If the target is lost, we stop the servos by setting their speed to 0 and increment a timer to keep track of how long the target remains lost. Otherwise, we use the pan and tilt speeds set in the `set_joint_cmd` callback and reset the timer to 0. We also set the `self.target_visible` flag back to `False` so that we have to explicitly set it to `True` when the next ROI message is received.

```
            if target_lost_timer > self.recenter_timeout:
                rospy.loginfo("Cannot find target.")
                self.center_head_servos()
                target_lost_timer = 0.0
```

If the target is lost long enough, recenter the servos by calling the `center_head_servos()` function defined later in the script. This not only prevents the servos from overheating but also places the camera in a more central position to reacquire the target.

```
            else:
                # Update the servo speeds at the appropriate interval
                if speed_update_timer > speed_update_interval:
                    if abs(self.last_pan_speed - self.pan_speed) >
self.speed_update_threshold:
                        self.set_servo_speed(self.head_pan_joint, self.pan_speed)
                        self.last_pan_speed = self.pan_speed

                    if abs(self.last_tilt_speed - self.tilt_speed) >
self.speed_update_threshold:
                        self.set_servo_speed(self.head_tilt_joint, self.tilt_speed)
                        self.last_tilt_speed = self.tilt_speed

                        speed_update_timer = 0.0
```

```
                # Update the pan position
                if self.last_pan_position != self.pan_position:
                    self.set_servo_position(self.head_pan_joint,
self.pan_position)
                    self.last_pan_position = self.pan_position

                # Update the tilt position
                if self.last_tilt_position != self.tilt_position:
                    self.set_servo_position(self.head_tilt_joint,
self.tilt_position)
                    self.last_tilt_position = self.tilt_position

            speed_update_timer += tick

        finally:
            # Release the lock
            self.lock.release()

        r.sleep()
```

Here we finally update the servo speeds and positions. First we check if we have reached the `speed_update_interval`. Otherwise, we leave the speeds alone. Recall that we do this because the Dynamixels can behave erratically if we attempt to update their speeds too frequently. We also check to see if the new speeds differ significantly from the previous speeds; otherwise we skip the speed update.

The variables `self.pan_speed` and `self.pan_position` are set in the `set_joint_cmd()` callback which we will look at below. The callback also sets the pan and tilt angles, `self.pan_position` and `self.tilt_position` based on the location of the target relative to the camera's center of view.

At the end of the update cycle, we release the lock and sleep for 1/self.rate seconds.

Finally, let's look at the `set_joint_cmd()` callback function which fires whenever we receive a message on the `/roi` topic.

```
def set_joint_cmd(self, msg):
    # Acquire the lock
    self.lock.acquire()

    try:
        # Target is lost if it has 0 height or width
        if msg.width == 0 or msg.height == 0:
            self.target_visible = False
            return

        # If the ROI stops updating this next statement will not happen
        self.target_visible = True
```

First we acquire a lock to protect the joint variables and the `target_visible` flag. Then we check for zero width or height for the incoming ROI message as this would indicate a region with zero area. In such cases, we set the target visibility flat to `False` and return immediately.

If we make it past the first check, we set the `target_visible` flag to `True`. As we saw earlier in the main program loop, the flag is reset to `False` on each cycle.

```
    # Compute the displacement of the ROI from the center of the image
    target_offset_x = msg.x_offset + msg.width / 2 - self.image_width / 2
    target_offset_y = msg.y_offset + msg.height / 2 - self.image_height / 2
```

Next, we compute how far off-center the middle of ROI lies. Recall that the `x_offset` and `y_offset` fields in an ROI message specify coordinates of the upper left corner of the ROI. (The width and height of the image are determined by the `get_camera_info()` callback which in turn is assigned to the subscriber to the `camera_info` topic.)

```
    try:
        percent_offset_x = float(target_offset_x) / (float(self.image_width) /
2.0)
        percent_offset_y = float(target_offset_y) / (float(self.image_height) /
2.0)
    except:
        percent_offset_x = 0
        percent_offset_y = 0
```

To accommodate different image resolutions, it is better to work in relative displacements. The `try-except` block is used since the `camera_info` topic can sometimes hiccup and send us a value of 0 for the image width and height.

```
    # Get the current position of the pan servo
    current_pan =
self.joint_state.position[self.joint_state.name.index(self.head_pan_joint)]
```

We will need the current pan position of the servo which we can get from the array `self.joint_state.position`. Recall that `self.joint_state` is set in the callback assigned to the subscriber to the `joint_state` topic.

```
    # Pan the camera only if the x target offset exceeds the threshold
    if abs(percent_offset_x) > self.pan_threshold:
        # Set the pan speed proportional to the target offset
        self.pan_speed = min(self.max_joint_speed, max(0, self.gain_pan *
abs(percent_offset_x)))

        if target_offset_x > 0:
            self.pan_position = max(self.min_pan, current_pan -
self.lead_target_angle)
        else:
```

```
            self.pan_position = min(self.max_pan, current_pan +
self.lead_target_angle)

    else:
        self.pan_speed = 0
        self.pan_position = current_pan
```

If the horizontal displacement of the target exceeds our threshold, we set the pan servo speed proportional to the displacement. We then set the pan position to the current position plus or minus the lead angle depending on the direction of the displacement. If the target displacement falls below the threshold, then we set the servo speed to 0 and the goal position to the current position.

The process is then repeated for the tilt servo, setting its speed and target position depending on the vertical displacement of the target.

```
  finally:
      # Release the lock
      self.lock.release()
```

Finally, we release the lock. And that completes the most important parts of the script. The rest should be fairly self-explanatory from the comments in the code.

12.11.3 Tracking Colored Objects

We can use the same `head_tracker.py` node to track colored objects or arbitrary objects selected using the mouse. The head tracker script simply follows the coordinates published on the `/roi` topic, so any node that publishes `RegionOfInterest` messages on that topic can control the movement of the camera.

To track a colored object, simply launch the CamShift node we developed earlier instead of the face tracker node used in the first head tracking example.

The complete sequence of steps would be as follows. Skip any launch files that you already have running. But `Ctrl-C` out of the face tracker launch if it is still running.

First make sure your camera driver is up and running. For a Kinect or Xtion Pro:

```
$ roslaunch rbx1_vision openni_node.launch
```

or if you are using a webcam:

```
$ roslaunch rbx1_vision uvc_cam.launch device:=/dev/video0
```

(Change the video device if necessary.) Now fire up the servo launch file if is not already running:

```
$ roslaunch rbx1_dynamixels dynamixels.launch
```

Launch the CamShift node:

```
$ roslaunch rbx1_vision camshift.launch
```

When the CamShift video window appears, select the target region with your mouse and adjust the hue/value slider controls as necessary to isolate the target from the background. When you are finished, bring up the head tracker node:

```
$ roslaunch rbx1_dynamixels head_tracker.launch
```

The servos should now move the camera to track the colored target.

12.11.4 Tracking Manually Selected Targets

Recall that the `lk_tracker.py` node allows us to select an object with the mouse and the script will then use keypoints and optical flow to track the object. Since the coordinates of the tracked object are published on the `/roi` topic, we can use the head tracker node just as we did with faces and colors.

This demo works best if the target object has a highly textured surface with robust keypoints such as the cover of a book or other graphic. It also helps if the background is fairly homogeneous such as a wall.

The complete sequence of steps would be as follows. Skip any launch files that you already have running. But `Ctrl-C` out of either the CamShift or face tracker if it is still running.

First make sure your camera driver is up and running. For a Kinect or Xtion Pro:

```
$ roslaunch rbx1_vision openni_node.launch
```

or if you are using a webcam:

```
$ roslaunch rbx1_vision uvc_cam.launch device:=/dev/video0
```

(Change the video device if necessary.)

Fire up the servos:

```
$ roslaunch rbx1_dynamixels dynamixels.launch
```

Launch the `lk_tracker.py` node:

```
$ roslaunch rbx1_vision lk_tracker.launch
```

When the video window appears, select the target object with your mouse then bring up the head tracker node:

```
$ roslaunch rbx1_dynamixels head_tracker.launch
```

The servos should now move the camera to track the selected target. There will be a limit to how fast you can move the target otherwise the optical flow tracker will not keep up. Also, you can re-select the target region at any time with your mouse.

12.12 A Complete Head Tracking ROS Application

The last three examples all have 3 out of 4 launch files in common. We can create a single launch file that includes the three common files and then selects the particular vision launch file we want based on a user provided argument.

Take a look at the `head_tracker_app.launch` file in the `rbx1_apps/launch` directory and reproduced below:

```
<launch>
  <!-- For a  Kinect or Xtion, set the depth_camera arg = True.
       For a webcam, set the value = False -->
  <arg name="depth_camera" default="True" />

  <!-- These arguments determine which vision node we run -->
  <arg name="face" default="False" />
  <arg name="color" default="False" />
  <arg name="keypoints" default="False" />

  <!-- Launch the appropriate camera drivers based on the depth_camera arg -->
  <include if="$(arg depth_camera)" file="$(find
rbx1_vision)/launch/openni_node.launch" />
  <include unless="$(arg depth_camera)" file="$(find
rbx1_vision)/launch/uvc_cam.launch" />

  <include if="$(arg face)" file="$(find
rbx1_vision)/launch/face_tracker2.launch" />
  <include if="$(arg color)" file="$(find
rbx1_vision)/launch/camshift.launch" />
  <include if="$(arg keypoints)" file="$(find
rbx1_vision)/launch/lk_tracker.launch" />

  <include file="$(find rbx1_dynamixels)/launch/dynamixels.launch" />

  <include file="$(find rbx1_dynamixels)/launch/head_tracker.launch" />
</launch>
```

The launch file uses a number of arguments to control which other launch files are included in any particular run. The `depth_camera` argument determines whether we load the OpenNI drivers for a Kinect or Xtion or the UVC drivers for a webcam. The default value of `True` will load the OpenNI drivers as we will see later in the launch file.

The next three arguments, `face`, `color`, and `keypoints` determine which vision node we will run. All three default to `False` so we must set one of them to `True` on the command line. Following the argument definitions, we run either the `openni_node.launch` file or the `uvc_camera.launch` file depending on the value of `depth_camera`.

One of the next three lines is run depending on which vision argument we set to `True` on the command line. For example, if use set the `color` argument to `True`, then the `camshift.launch` file will be run.

Next we run the `dynamixels.launch` file to fire up the servos. And finally, we run the `head_tracker_app.launch` file to start the head tracking.

Using our new launch file, we can now start the entire head tracking application to track either faces, colors or keypoints with a single command. For example, to track a face using either a Kinect or Xtion camera (the default), we would run:

```
$ roslaunch rbx1_apps head_tracker_app.launch face:=True
```

To track colors instead, we would run:

```
$ roslaunch rbx1_apps head_tracker_app.launch color:=True
```

Or to use a webcam and keypoint tracking:

```
$ roslaunch rbx1_apps head_tracker_app.launch \
  depth_camera:=False keypoints:=True
```

One final note: if you forget to select one of the vision modes on the command line, you can always run its launch file afterward. In other words, the command:

```
$ roslaunch rbx1_apps head_tracker_app.launch face:=True
```

is equivalent to running:

```
$ roslaunch rbx1_apps head_tracker_app.launch
```

followed by:

```
$ roslaunch rbx1_vision face_tracker2.launch
```

13. Where to Go Next?

Hopefully this book has provided enough examples to help you get a good start on programming your own ROS applications. You should now have the tools you need to write applications that combine computer vision with speech recognition, text-to-speech, robot navigation and servo control. For example, how about writing an application that periodically scans the room for a person's face and when someone is detected, speaks a greeting of some sort, moves toward the person, then asks if they need any help? Or how about placing an object on your robot and using a voice command to take it to a person in another room?

As we mentioned in the introduction, there are still many areas of ROS to explore including:

- Controlling a multi-jointed arm like the TurtleBot arm.
- 3-D image processing using PCL (Point Cloud Library).
- Identifying and grasping objects on a table top. (Or how about playing chess?)
- Identifying your friends and family using face_recognition.
- Creating a URDF robot model for your own robot.
- Adding sensors such as a gyro, sonar, IR, or laser scanner to your robot.
- Running realistic simulations with Gazebo.
- Programming state machines using SMACH.
- Building knowledge bases with knowrob.
- Recognizing arbitrary 3-D objects using roboearth.
- Learning from experience using reinforcement_learning.
- Many additional options to explore using OpenCV including template matching, motion analysis, and machine learning.

There are now over 2000 packages and libraries available for ROS. Click on the Browse Software link at the top of the ROS Wiki for a list of all ROS packages and stacks that have been submitted for indexing. When you are ready, you can contribute your own package(s) back to the ROS community. Welcome to the future of robotics. Have fun and good luck!

Notes