

ROS

By Example

Volume 2

Packages and Programs for Advanced
Robot Behaviors



R. Patrick Goebel

ROS By Example

VOLUME 2

PACKAGES AND PROGRAMS FOR
ADVANCED ROBOT BEHAVIORS

A PI ROBOT PRODUCTION

R. PATRICK GOEBEL

Version 1.1.0

For ROS Indigo

ROS BY EXAMPLE. Copyright © 2014 by R. Patrick Goebel

All Rights Reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN: 978-1-312-39266-3

Version 1.1.0 for ROS Indigo: May 2015

Products and company names mentioned herein may be the trademarks of their respective owners. Rather than use a trademark symbol with every occurrence of a trademark name, we are using the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Information contained in this work (Paperback or eBook) has been obtained from sources believed to be reliable. However, the author does not guarantee the accuracy or completeness of any information contained in it, and the author shall not be responsible for any errors, omissions, losses, or damages caused or alleged to be caused directly or indirectly by the information published herein. This work is published with the understanding that the author is supplying information but is not attempting to render professional services. This product almost certainly contains errors. It is your responsibility to examine, verify, test, and determine the appropriateness of use, or request the assistance of an appropriate professional to do so.

PREFACE

In February 2013, ROS stewardship was transferred from Willow Garage to the Open Source Robotics Foundation ([OSRF](#)). The stated mission of OSRF is "to support the development, distribution, and adoption of open source software for use in robotics research, education, and product development." The two major projects overseen by OSRF are [ROS](#) and [Gazebo](#), the advanced 3D robot simulator.

New ROS packages continue to be released on a regular basis by individuals and groups working in both university labs and the robotics industry. Interest in learning ROS continues to grow, and the popularity of *ROS By Example Volume 1* has exceeded my expectations. Yet going beyond the basics still requires a fairly steep learning curve and realistic examples dealing with more advanced topics are not always easy to find.

The overall goal of *Volume 2* is to introduce a collection of ROS packages and tools that are essential for programming truly autonomous robots. This includes the use of executive task managers like SMACH and Behavior Trees, creating custom URDF models for different robots, monitoring the health of the robot using ROS diagnostics, multiplexing control inputs and assigning priorities, controlling a multi-jointed arm for reaching and grasping, monitoring and controlling the robot through a web browser, and performing realistic simulations using Gazebo. When we are finished, you will have the tools necessary to program an "always on" robot that can monitor its own subsystems and perform a series of tasks without human intervention.

Please note that the code in this book is written for ROS Indigo. While some of it might work on earlier versions of ROS, it is highly recommended that the reader use ROS Indigo with this book.

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 ask questions about the book or the sample code, or receive notifications regarding updates and bug fixes, please join the [ros-by-example Google Group](#).

Main Chapter Headings

Preface.....	vii
Printed vs PDF Versions of the Book.....	ix
Changes Since ROS Hydro.....	xix
1. Scope of this Volume.....	1
2. Installing the ros-by-example Code.....	3
3. Task Execution using ROS.....	7
4. Creating a URDF Model for your Robot.....	89
5. Controlling Dynamixel Servos: Take 2.....	147
6. Robot Diagnostics.....	167
7. Dynamic Reconfigure.....	189
8. Multiplexing Topics with mux & yocs.....	201
9. Head Tracking in 3D.....	213
10. Detecting and Tracking AR Tags.....	231
11. Arm Navigation using MoveIt!.....	251
12. Gazebo: Simulating Worlds and Robots.....	369
13. Rosbridge: Building a Web GUI for your Robot.....	403
Appendix: Plug and Play USB Devices for ROS: Creating udev Rules.....	437

Contents

Preface.....	vii
Printed vs PDF Versions of the Book.....	ix
Changes Since ROS Hydro.....	xix
1. Scope of this Volume.....	1
2. Installing the ros-by-example Code.....	3
3. Task Execution using ROS.....	7
3.1 A Fake Battery Simulator.....	8
3.2 A Common Setup for Running the Examples.....	10
3.3 A Brief Review of ROS Actions.....	11
3.4 A Patrol Bot Example.....	12
3.5 The Patrol Bot using a Standard Script.....	13
3.6 Problems with the Script Approach.....	16
3.7 SMACH or Behavior Trees?.....	16
3.8 SMACH: Tasks as State Machines.....	17
3.8.1 SMACH review.....	18
3.8.2 Patrolling a square using SMACH.....	19
3.8.3 Testing SMACH navigation in the ArbotiX simulator.....	23
3.8.4 Accessing results from a SimpleActionState.....	26
3.8.5 SMACH Iterators.....	27
3.8.6 Executing commands on each transition.....	29
3.8.7 Interacting with ROS topics and services.....	30
3.8.8 Callbacks and Introspection.....	35
3.8.9 Concurrent tasks: Adding the battery check to the patrol routine.....	36
3.8.10 Comments on the battery checking Patrol Bot.....	43
3.8.11 Passing user data between states and state machines.....	43
3.8.12 Subtasks and hierarchical state machines.....	47
3.8.13 Adding the battery check to the house cleaning robot.....	53
3.8.14 Drawbacks of state machines.....	53
3.9 Behavior Trees.....	54
3.9.1 Behavior Trees versus Hierarchical State Machines.....	55
3.9.2 Key properties of behavior trees.....	56
3.9.3 Building a behavior tree.....	57
3.9.4 Selectors and sequences.....	59
3.9.5 Customizing behaviors using decorators (meta-behaviors).....	60
3.10 Programming with Behavior Trees and ROS.....	61
3.10.1 Installing the pi_trees library.....	61
3.10.2 Basic components of the pi_trees library.....	62
3.10.3 ROS-specific behavior tree classes.....	67
3.10.4 A Patrol Bot example using behavior trees.....	72
3.10.5 A housing cleaning robot using behavior trees.....	78
3.10.6 Parallel tasks.....	84
3.10.7 Adding and removing tasks.....	86

4. Creating a URDF Model for your Robot.....	89
4.1 Start with the Base and Wheels.....	90
4.1.1 The robot_state_publisher and joint_state_publisher nodes.....	91
4.1.2 The base URDF/Xacro file.....	92
4.1.3 Alternatives to using the /base_footprint frame.....	97
4.1.4 Adding the base to the robot model.....	97
4.1.5 Viewing the robot's transform tree.....	98
4.1.6 Using a mesh for the base.....	99
4.2 Simplifying Your Meshes.....	104
4.3 Adding a Torso.....	104
4.3.1 Modeling the torso.....	105
4.3.2 Attaching the torso to the base.....	106
4.3.3 Using a mesh for the torso.....	107
4.3.4 Adding the mesh torso to the mesh base.....	108
4.4 Measure, Calculate and Tweak.....	110
4.5 Adding a Camera.....	110
4.5.1 Placement of the camera.....	111
4.5.2 Modeling the camera.....	112
4.5.3 Adding the camera to the torso and base.....	114
4.5.4 Viewing the transform tree with torso and camera.....	115
4.5.5 Using a mesh for the camera.....	116
4.5.6 Using an Asus Xtion Pro instead of a Kinect.....	118
4.6 Adding a Laser Scanner (or other Sensors).....	118
4.6.1 Modeling the laser scanner.....	119
4.6.2 Attaching a laser scanner (or other sensor) to a mesh base.....	120
4.6.3 Configuring the laser node launch file.....	121
4.7 Adding a Pan and Tilt Head.....	122
4.7.1 Using an Asus Xtion Pro instead of a Kinect.....	124
4.7.2 Modeling the pan-and-tilt head.....	124
4.7.3 Figuring out rotation axes.....	127
4.7.4 A pan and tilt head using meshes on Pi Robot.....	128
4.7.5 Using an Asus Xtion Pro mesh instead of a Kinect on Pi Robot.....	129
4.8 Adding One or Two Arms.....	129
4.8.1 Placement of the arm(s).....	129
4.8.2 Modeling the arm.....	129
4.8.3 Adding a gripper frame for planning.....	132
4.8.4 Adding a second arm.....	133
4.8.5 Using meshes for the arm servos and brackets.....	135
4.9 Adding a Telescoping Torso to the Box Robot.....	137
4.10 Adding a Telescoping Torso to Pi Robot.....	138
4.11 A Tabletop One-Arm Pi Robot.....	138
4.12 Testing your Model with the ArbotiX Simulator.....	140
4.12.1 A fake Box Robot.....	141
4.12.2 A fake Pi Robot.....	143
4.13 Creating your own Robot Description Package.....	144
4.13.1 Copying files from the rbx2_description package.....	145
4.13.2 Creating a test launch file.....	145
5. Controlling Dynamixel Servos: Take 2.....	147

5.1	Installing the ArbotiX Packages.....	147
5.2	Launching the ArbotiX Nodes.....	148
5.3	The ArbotiX Configuration File.....	152
5.4	Testing the ArbotiX Joint Controllers in Fake Mode.....	158
5.5	Testing the Arbotix Joint Controllers with Real Servos.....	160
5.6	Relaxing All Servos.....	163
5.7	Enabling or Disabling All Servos.....	166
6.	Robot Diagnostics.....	167
6.1	The DiagnosticStatus Message.....	168
6.2	The Analyzer Configuration File.....	169
6.3	Monitoring Dynamixel Servo Temperatures.....	170
6.3.1	Monitoring the servos for a pan-and-tilt head.....	170
6.3.2	Viewing messages on the /diagnostics topic.....	173
6.3.3	Protecting servos by monitoring the /diagnostics topic.....	175
6.4	Monitoring a Laptop Battery.....	179
6.5	Creating your Own Diagnostics Messages.....	180
6.6	Monitoring Other Hardware States.....	186
7.	Dynamic Reconfigure.....	189
7.1	Adding Dynamic Parameters to your own Nodes.....	190
7.1.1	Creating the .cfg file.....	190
7.1.2	Making the .cfg file executable.....	191
7.1.3	Configuring the CMakeLists.txt file.....	192
7.1.4	Building the package.....	192
7.2	Adding Dynamic Reconfigure Capability to the Battery Simulator Node.....	192
7.3	Adding Dynamic Reconfigure Client Support to a ROS Node.....	196
7.4	Dynamic Reconfigure from the Command Line.....	199
8.	Multiplexing Topics with mux & yocs.....	201
8.1	Configuring Launch Files to use mux Topics.....	202
8.2	Testing mux with the Fake TurtleBot.....	203
8.3	Switching Inputs using mux Services.....	204
8.4	A ROS Node to Prioritize mux Inputs.....	205
8.5	The YOCS Controller from Yujin Robot.....	208
8.5.1	Adding input sources.....	211
9.	Head Tracking in 3D.....	213
9.1	Tracking a Fictional 3D Target.....	214
9.2	Tracking a Point on the Robot.....	215
9.3	The 3D Head Tracking Node.....	218
9.3.1	Real or fake head tracking.....	218
9.3.2	Projecting the target onto the camera plane.....	219
9.4	Head Tracking with Real Servos.....	222
9.4.1	Real servos and fake target.....	223
9.4.2	Real servos, real target.....	224
9.4.3	The nearest_cloud.py node and launch file.....	226

10. Detecting and Tracking AR Tags.....	231
10.1 Installing and Testing the ar_track_alvar Package.....	232
10.1.1 Creating your own AR Tags.....	232
10.1.2 Generating and printing the AR tags.....	234
10.1.3 Launching the camera driver and ar_track_alvar node.....	234
10.1.4 Testing marker detection.....	236
10.1.5 Understanding the /ar_pose_marker topic.....	236
10.1.6 Viewing the markers in RViz.....	238
10.2 Accessing AR Tag Poses in your Programs.....	239
10.2.1 The ar_tags_cog.py script.....	239
10.2.2 Tracking the tags with a pan-and-tilt head.....	243
10.3 Tracking Multiple Tags using Marker Bundles.....	244
10.4 Following an AR Tag with a Mobile Robot.....	244
10.4.1 Running the AR follower script on a TurtleBot	247
10.5 Exercise: Localization using AR Tags.....	248
11. Arm Navigation using MoveIt!.....	251
11.1 Do I Need a Real Robot with a Real Arm?.....	252
11.2 Degrees of Freedom.....	252
11.3 Joint Types.....	253
11.4 Joint Trajectories and the Joint Trajectory Action Controller.....	254
11.5 Forward and Inverse Arm Kinematics.....	257
11.6 Numerical versus Analytic Inverse Kinematics.....	258
11.7 The MoveIt! Architecture.....	258
11.8 Installing MoveIt!.....	260
11.9 Creating a Static URDF Model for your Robot	260
11.10 Running the MoveIt! Setup Assistant.....	261
11.10.1 Load the robot's URDF model.....	262
11.10.2 Generate the collision matrix.....	263
11.10.3 Add the base_odom virtual joint.....	264
11.10.4 Adding the right arm planning group.....	265
11.10.5 Adding the right gripper planning group.....	268
11.10.6 Defining robot poses.....	270
11.10.7 Defining end effectors.....	272
11.10.8 Defining passive joints.....	272
11.10.9 Generating the configuration files.....	272
11.11 Configuration Files Created by the MoveIt! Setup Assistant.....	274
11.11.1 The SRDF file (robot_name.srdf).....	274
11.11.2 The fake_controllers.yaml file.....	275
11.11.3 The joint_limits.yaml file.....	276
11.11.4 The kinematics.yaml file.....	277
11.12 The move_group Node and Launch File.....	279
11.13 Testing MoveIt! in Demo Mode.....	279
11.13.1 Exploring additional features of the Motion Planning plugin.....	283
11.13.2 Re-running the Setup Assistant at a later time.....	284
11.14 Testing MoveIt! from the Command Line.....	285
11.15 Determining Joint Configurations and End Effector Poses.....	288
11.16 Using the ArbotiX Joint Trajectory Action Controllers.....	291

11.16.1 Testing the ArbotiX joint trajectory action controllers in simulation.....	291
11.16.2 Testing the ArbotiX joint trajectory controllers with real servos.....	299
11.17 Configuring MoveIt! Joint Controllers.....	300
11.17.1 Creating the controllers.yaml file.....	301
11.17.2 Creating the controller manager launch file.....	303
11.18 The MoveIt! API.....	304
11.19 Forward Kinematics: Planning in Joint Space.....	304
11.20 Inverse Kinematics: Planning in Cartesian Space.....	313
11.21 Pointing at or Reaching for a Visual Target.....	321
11.22 Setting Constraints on Planned Trajectories.....	323
11.22.1 Executing Cartesian Paths.....	323
11.22.2 Setting other path constraints.....	329
11.23 Adjusting Trajectory Speed.....	333
11.24 Adding Obstacles to the Planning Scene.....	336
11.25 Attaching Objects and Tools to the Robot.....	345
11.26 Pick and Place.....	347
11.27 Adding a Sensor Controller.....	359
11.28 Running MoveIt! on a Real Arm.....	363
11.28.1 Creating your own launch files and scripts.....	364
11.28.2 Running the robot's launch files.....	364
11.28.3 Forward kinematics on a real arm.....	365
11.28.4 Inverse kinematics on a real arm.....	365
11.28.5 Cartesian paths on a real arm.....	366
11.28.6 Pick-and-place on a real arm.....	366
11.28.7 Pointing at or reaching for a visual target.....	367
12. Gazebo: Simulating Worlds and Robots.....	369
12.1 Installing Gazebo.....	370
12.2 Hardware Graphics Acceleration.....	371
12.3 Installing the ROS Gazebo Packages.....	372
12.4 Installing the Kobuki ROS Packages.....	373
12.5 Installing the UBR-1 Files.....	373
12.6 Using the Gazebo GUI.....	373
12.7 Testing the Kobuki Robot in Gazebo.....	375
12.7.1 Accessing simulated sensor data.....	378
12.7.2 Adding safety control to the Kobuki.....	382
12.7.3 Running the nav_square.py script from Volume 1.....	384
12.8 Loading Other Worlds and Objects.....	385
12.9 Testing the UBR-1 Robot in Gazebo.....	386
12.9.1 UBR-1 joint trajectories.....	387
12.9.2 The UBR-1 and MoveIt!.....	388
12.10 Real Pick-and-Place using the UBR-1 Perception Pipeline.....	390
12.10.1 Limitations of depth cameras.....	391
12.10.2 Running the demo.....	391
12.10.3 Understanding the real_pick_and_place.py script.....	397
12.11 Running Gazebo Headless + RViz.....	400
13. Rosbridge: Building a Web GUI for your Robot.....	403

13.1 Installing the rosbridge Packages.....	403
13.2 Installing the web_video_server Package.....	404
13.3 Installing a Simple Web Server (mini-htpd).....	406
13.4 Starting mini-htpd, rosbridge and web_video_server.....	407
13.5 A Simple rosbridge HTML/Javascript GUI.....	409
13.6 Testing the GUI with a Fake TurtleBot.....	411
13.7 Testing the GUI with a Real Robot.....	412
13.8 Viewing the Web GUI on another Device on your Network.....	412
13.9 Using the Browser Debug Console.....	413
13.10 Understanding the Simple GUI.....	414
13.10.1 The HTML layout: simple_gui.html.....	415
13.10.2 The JavaScript code: simple_gui.js.....	419
13.11 A More Advanced GUI using jQuery, jqWidgets and KineticJS.....	430
13.12 Rosbridge Summary.....	435

Appendix: Plug and Play USB Devices for ROS: Creating udev Rules.....437

13.13 Adding yourself to the dialout Group.....	437
13.14 Determining the Serial Number of a Device.....	438
13.15 UDEV Rules.....	439
13.16 Testing a UDEV Rule.....	440
13.17 Using a UDEV Device Name in a ROS Configuration File.....	440

CHANGES SINCE ROS HYDRO

If Indigo is your first experience with ROS and this book, then you can safely skip this chapter. However, if you have already been using the previous version of this book with ROS Hydro, there are a few changes to be noted. You can read the official list of differences between ROS Hydro and Indigo on the [Hydro → Indigo migration](#) page.

Here are some of the items that affect the code used with this book.

- The `ar_track_alvar` package now stores its message types in a separate package, `ar_track_alvar_msgs`. The import statement in the sample nodes `ar_tag_cog.py` and `ar_follower.py` has therefore been updated accordingly.
- The `laptop_battery.py` script has been moved out of the `turtlebot` meta package and now lives in a package of its own, `ros-indigo-laptop-battery-monitor`. Similarly, the `LaptopChargeStatus` message type that used to be found in the `linux_hardware` package has been replaced with the `SmartBatteryStatus` message found in the `smart_battery_msgs` package.
- The `fake_localization` node now requires remapping from `base_ground_truth` topic to `odom` or `odom_combined`. This update has been applied to the `rbx2` sample code.
- The ArbotiX gripper controller node launched in the `pi_robot_with_gripper.launch` file now uses minimum and maximum opening parameters to check on valid position commands.
- The `mjpeg_server` image streaming package has been deprecated and replaced by the new [`web_video_server`](#) package. The chapter on Rosbridge has been updated accordingly.
- The previous `openni` meta-package and has been superseded by two new packages: [`openni2_camera`](#) for use with the Asus Xtion, Xtion Pro and Primesense 1.08/1.09 cameras and [`freenect_camera`](#) for the Microsoft Kinect. This has some potentially important consequences depending on which camera you use:
 - The earlier `openni` drivers for the Kinect supported resolutions down to 160x120 pixels. Unfortunately, the **lowest resolution** supported by the `freenect` driver for the Kinect is 640x480 pixels (VGA) while the lowest resolution supported on the Xtion Pro using the `openni2` driver is 320x240 (QVGA). While these resolutions will generally work OK when using a fast laptop or PC, be aware that lower-power CPUs or smaller single-board computers may struggle to process video streams at resolutions above

320x240. For example, it can be difficult to get smooth object tracking at 640x480 pixels on a Core Duo processor without graphics acceleration since tracking tends to lag behind the video processing at this resolution.

Fortunately, we can often use the camera driver's *data skipping* function to get around this as we will see in Chapter 8.

- When using the earlier `openni` (version 1) driver with previous revisions of this book, the default color image topic was `/camera/rgb/image_color` for both the Kinect and Xtion Pro cameras. However, when using the `openni2` driver with an Asus Xtion Pro, the default color image topic is `/camera/rgb/image_raw` while the topic `/camera/rgb/image_color` is not used at all. On the other hand, when using the `freenect` driver with a Microsoft Kinect, the color image data is published on *both* `/camera/rgb/image_raw` and `/camera/rgb/image_color`. Since `/camera/rgb/image_raw` is used by both cameras, we will switch to this topic instead of `/camera/rgb/image_color` for this revision of the book. The sample code launch files have therefore been updated to use this topic name.
- The `openni2` and `freenect` drivers use **different units for depth image values**. The `freenect` driver uses **millimeters** when publishing depth images while the `openni2` driver uses **meters**. To convert the `freenect` values to meters we divide by 1000. This has been done in the sample code. Note that this only applies to depth *images*. When it comes to point clouds, both drivers publish depth values in meters.
- The `openni2` driver does not appear to generate **disparity data** for the Asus Xtion cameras. Consequently, the `disparity_view` node that we used in previous book revisions can no longer be used to view colored depth images from these cameras. However, we can still view a grayscale version of the depth image. For the Kinect, the `freenect` driver **does** publish disparity data on the topic `/camera/depth/disparity`.
- The original **openni.org** website was shut down on April 23, 2014. However, OpenNI 2 binaries are being preserved on the [Structure website](#). The `freenect` drivers are being developed through the [OpenKinect](#) project.
- The **default camera resolution** in all sample code has been changed from 320x240 to 640x480 to match the default resolution used by the camera launch files distributed with ROS and the minimum resolution supported by the `freenect` driver for the Kinect. If you are running vision processing on a low-powered CPU, and you are using an Asus Xtion camera, you will get significantly better frame rates and tracking performance if you reduce the resolution to 320x240 using `rqt_reconfigure` or adjusting the settings in the camera's launch file. As noted above, the lowest resolution available on the

Kinect when using the `freenect` driver is 640x480 so using a smaller resolution is not possible for this camera.

- The OpenCV `cv_to_imgmsg` function in the `cv_bridge` package has been replaced with the OpenCV2 version `cv2_to_imgmsg` that uses image arrays instead of the older OpenCV image matrix format. This update has been done for the `rbx1` code. At the same time, the older `cv.fromarray()` function is no longer necessary to convert image arrays to the older OpenCV matrix format.
- Likewise, the OpenCV `imgmsg_to_cv` function in the `cv_bridge` package has been replaced with the OpenCV2 version `imgmsg_to_cv2` that uses image arrays instead of the older OpenCV image matrix format. At the same time, the older `cv.toarray()` function is no longer necessary to convert image arrays to the older OpenCV matrix format.
- Creating a publisher without an explicit `queue_size` parameter will now generate a warning and force the publisher to behave synchronously. It is preferable to specify an explicit `queue_size` parameter which will also place the publisher in asynchronous mode. For more information please see the [Publisher and Subscriber Overview](#) on the ROS Wiki. The Indigo `ros-by-example` code has been updated to include a `queue_size` parameter for all publishers. **NOTE:** Not all third-party ROS Indigo packages have been updated to include a `queue_size` parameter when defining a publisher. You will therefore occasionally see a warning about a missing `queue_size` parameter when running various packages. These warnings should be harmless.
- A new `CallbackTask` has been added to the `pi_trees` library that turns any Python function into a task.
- At the time of this writing, there are bugs in both OpenRAVE and the `moveit_ikfast` package that prevent building an IKFast plugin for MoveIt under ROS Indigo. Consequently, this interim revision does not include section 11.29, *Creating a Custom IK Fast Plugin*. The section will reappear once the problems with OpenRAVE and `moveit_ikfast` have been resolved.

1. SCOPE OF THIS VOLUME

In *Volume 1* we learned how to program a robot using the fundamental components of ROS including the control of a mobile base, SLAM, robot vision (OpenCV, OpenNI and a little bit of PCL), speech recognition, and joint control using Dynamixel servos. In this volume we will tackle some of the more advanced ROS concepts and packages that are essential for programming truly autonomous robot behaviors including:

- executive task managers such as [smach](#) and [behavior trees](#)
- creating a model for your own robot using [URDF/Xacro](#) descriptions, including a pan-and-tilt head, multi-jointed arm(s) and gripper(s), a telescoping torso, and the placement of sensors such as a laser scanner
- configuring the [arbotix](#) packages for controlling Dynamixel servos
- using the ROS [diagnostics](#) package to enable your robot to monitor its own systems such as battery levels and servo temperatures
- altering your robot's parameters on the fly using [dynamic_reconfigure](#)
- multiplexing ROS topics using [mux](#) and [yocs](#) so that control inputs can be prioritized and not work against each other
- using [AR tags](#) for object detection and tracking
- tracking objects in 3D and perceiving the spatial relationship between an object and the robot
- controlling a multi-jointed arm and gripper using the new [MoveIt!](#) framework including the execution of forward and inverse kinematics, collision avoidance, grasping, and pick-and-place tasks
- working with simulated robots and environments using the sophisticated [Gazebo](#) simulator
- creating a web-based GUI for your robot using [rosbridge](#), HTML and Javascript

NOTE: The chapter on Gazebo assumes we already have a robot model like the [Kobuki](#) or [UBR-1](#) whose developers have provided the necessary Gazebo properties and plugins to handle the simulated physics of the robot. Creating either worlds or Gazebo plugins

for a new robot from scratch is outside the scope of this volume but is covered in the online [Gazebo Tutorials](#).

Several of the topics we will cover could consume an entire book on their own so we will focus on the key concepts while providing additional references for the reader to explore further details if desired. As in the first volume, all the code samples are written in Python which tends to be a little more accessible than C++ for an introductory text.

Once you have mastered the concepts presented in this volume, you should be able to create a URDF model for your robot, including an arm and pan-and-tilt head, then program it to perform a series of autonomous tasks and behaviors, monitor its own subsystems, prioritize its control inputs to match the current situation, track objects in 3-dimensional space, understand how to program a multi-jointed arm and gripper using collision-aware inverse kinematics, and write your own HTML/Javascript interface to monitor and control your robot using a web browser.

2. INSTALLING THE ROS-BY-EXAMPLE CODE

Please note that the code in this book is written for **ROS Indigo**. While some of the code might work on earlier versions of ROS, it is highly recommended that the reader use ROS Indigo with this book. In particular, the prerequisite packages listed below and the MoveIt! sample programs for arm navigation are specific to Indigo.

Before installing the *Volume 2* 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. (If you are reading the printed version of the book, see below for alternate instructions.) 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-indigo-arbotix ros-indigo-openni-camera \
ros-indigo-dynamixel-motor ros-indigo-rosbridge-suite \
ros-indigo-mjpeg-server ros-indigo-rgbd-launch \
ros-indigo-moveit-full ros-indigo-moveit-ikfast \
ros-indigo-turtlebot-* ros-indigo-kobuki-* ros-indigo-moveit-python \
python-pygraph python-pygraphviz python-easygui \
mini-httpd ros-indigo-laser-pipeline ros-indigo-ar-track-alvar \
ros-indigo-laser-filters ros-indigo-hokuyo-node \
ros-indigo-depthimage-to-laserscan ros-indigo-moveit-ikfast \
ros-indigo-gazebo-ros ros-indigo-gazebo-ros-pkgs \
ros-indigo-gazebo-msgs ros-indigo-gazebo-plugins \
ros-indigo-gazebo-ros-control ros-indigo-cmake-modules \
ros-indigo-kobuki-gazebo-plugins ros-indigo-kobuki-gazebo \
ros-indigo-smach ros-indigo-smach-ros ros-indigo-grasping-msgs \
ros-indigo-executive-smach ros-indigo-smach-viewer \
ros-indigo-robot-pose-publisher ros-indigo-tf2-web-republisher \
ros-indigo-move-base-msgs ros-indigo-fake-localization \
graphviz-dev libgraphviz-dev gv python-scipy liburdfdom-tools \
ros-indigo-laptop-battery-monitor ros-indigo-ar-track-alvar*
```

If you are reading the printed version of the book, then copy and paste is probably not an option. Instead, you can use the following commands to download a small shell script called `rbx2-prereq.sh` that will run the `apt-get` command above:

```
$ cd ~
$ wget https://raw.githubusercontent.com/pirobot/rbx2/indigo-devel/\
rbx2-prereq.sh
$ sh rbx2-prereq.sh
```

We will also need the code from the *Volume 1* repository (`rbx1`) even if you do not have the book. To install the `rbx1` code for ROS Indigo (in case you don't already have it), run the following commands:

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

To clone and build the *Volume 2* repository (`rbx2`) for ROS Indigo, follow these steps:

```
$ cd ~/catkin_ws/src  
$ git clone -b indigo-devel https://github.com/pirobot/rbx2.git  
$ cd ~/catkin_ws  
$ catkin_make  
$ source ~/catkin_ws/devel/setup.bash
```

NOTE : 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 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/rbx2  
$ git pull  
$ cd ~/catkin_ws  
$ catkin_make  
$ source devel/setup.bash
```

Staying Up-To-Date: If you'd like to ask questions about the code, report bugs, or receive notifications of updates, please join the [ros-by-example Google Group](#).

All of the *ROS By Example Volume 2* packages begin with the letters `rbx2`. To list the packages, move into the parent of the `rbx2` meta-package and use the Linux `ls` command:

```
$ roscd rbox2  
$ cd ..  
$ ls -F
```

which should result in the following listing:

```
pedestal_pi_no_gripper_moveit_config/    rbt2_bringup/      rbt2_msgs/
pedestal_pi_with_gripper_moveit_config/   rbt2_description/  rbt2_nav/
pi_robot_moveit_config/                 rbt2_diagnostics/ rbt2_tasks/
rbt2/                                    rbt2_dynamixels/ rbt2_utils/
rbt2_arm_nav/                          rbt2_gazebo/       rbt2_vision/
rbt2_ar_tags/                         rbt2_gui/          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 `rbt2_dynamixels` package, you would use the command:

```
$ roscl rbt2_dynamixels
```

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 Indigo branch of the `rbt2` and `rbt1` repositories on both machines.

3. TASK EXECUTION USING ROS

As we discovered in *Volume 1*, it is relatively straightforward to program a robot to execute a particular behavior such as face tracking, navigating between locations, or following a person. But a fully autonomous robot will be expected to select its own actions from a larger repertoire of behaviors depending upon the task at hand and the current conditions.

In this chapter, we will learn how to use two different ROS-enabled task execution frameworks: [SMACH](#) (using state machines and pronounced "smash") and [pi_trees](#) (behavior trees). Each approach has its strengths and weaknesses and one might seem easier or harder depending on your programming background. But both methods provide a more structured approach to task management than simply writing a large list of if-then statements.

The overall task controller is often called the *task executive* and most such executives are expected to include at least the following key features:

- **task priorities:** A lower priority task should be preempted if a higher priority task requires the same resources (e.g. drive motors).
- **pause and resume:** It is often desirable to pause a currently running task (or tasks) when a higher-priority task is given control and then to resume the preempted task(s) when the executive yields back control. For example, the Neato vacuum cleaner is able to return to the place it left off after recharging.
- **task hierarchy:** Tasks can often be broken down into subtasks to handle the details. For instance, a high level task called *recharge* might consist of three subtasks: *navigate to the docking station*, *dock the robot*, and *charge the battery*. Similarly, the docking task could be broken down into: *align with beacon*; *drive forward*; *stop when docked*.
- **conditions:** Both sensor data and internal programming variables can place constraints on when and how a task will be executed. In ROS, these variables often take the form of messages being published by various nodes. For example, the battery monitoring task will subscribe to a diagnostics topic that includes the current battery level. When a low battery level is detected, a *check battery* condition should fire and the recharging task will begin execution while other tasks are paused or aborted.

- **concurrency**: Multiple tasks can run in parallel. For example, both the navigation task (getting to the next waypoint) and the battery monitoring task must run at the same time.

To keep things concrete, we will use two example scenarios throughout the chapter: a "Patrol Bot" that must visit a series of locations in sequence while keeping its battery from running down, and a "house cleaning robot" that must visit a number of rooms and perform various cleaning tasks relevant to each room.

3.1 A Fake Battery Simulator

An always-on robot will need to monitor its battery levels and recharge when necessary. To make our examples more realistic, we will therefore use a node that simulates a battery by publishing a decreasing value on a battery level ROS topic. Other nodes can then subscribe to this topic and respond accordingly when the battery level falls too low.

The battery simulator node [`battery_simulator.py`](#) can be found in the directory `rbx2_utils/nodes`. The script is fairly straightforward except perhaps for the parts relating to dynamic reconfigure which we will cover in detail in Chapter 7. For now, we only need to note the following.

The node takes three parameters:

- `rate`: (default 1 Hz) – how often to publish the battery level
- `battery_runtime`: (default 60 seconds) – how long in seconds it takes the battery to run down to 0
- `initial_battery_level`: (default 100) – the initial charge level of the battery when we first start the node

The node publishes a floating point value on the `/battery_level` topic starting with the `initial_battery_level` and counting down to 0 over a time period specified by the `battery_runtime` parameter. Both parameters can be specified in a launch file as we have done in the [`battery_simulator.launch`](#) file found in the `rbx2_utils/launch` directory. You can also specify the battery runtime (in seconds) on the command line as an argument. Let's test the simulator by running the launch file now:

```
$ roslaunch rbx2_utils battery_simulator.launch
```

You can verify that the simulator is working by opening another terminal and running the command:

```
$ rostopic echo /battery_level
```

which should yield an output similar to:

```
data: 91.0
---
data: 90.6666641235
---
data: 90.3333358765
---
etc
```

The node also defines a ROS service called `set_battery_level` that takes a floating point value as an argument and sets the battery charge to that level. We will use this service to simulate a recharge of the battery by setting the level to 100 or to simulate a sudden depletion of the battery by setting a low value.

The `set_battery_level` service can be used as follows:

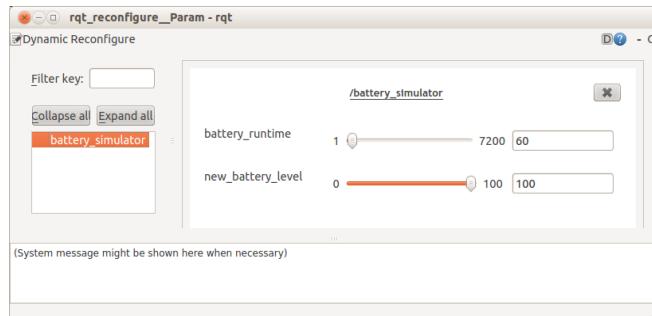
```
$ rosservice call /battery_simulator/set_battery_level 100
```

where the argument is a number between 0 and 100. To simulate a recharge, use the `set_battery_level` service to set the level to a high number such as 100. To simulate a sudden drop in the battery level, set the value to a lower level such as 30. This will allow us to test how various nodes respond to a low battery condition.

The battery simulator can also be controlled using `rqt_reconfigure`. To change the battery runtime or manually set the battery level, bring up `rqt_reconfigure`:

```
$ rosrun rqt_reconfigure rqt_reconfigure
```

then click on the `battery_simulator` node to see the following options:

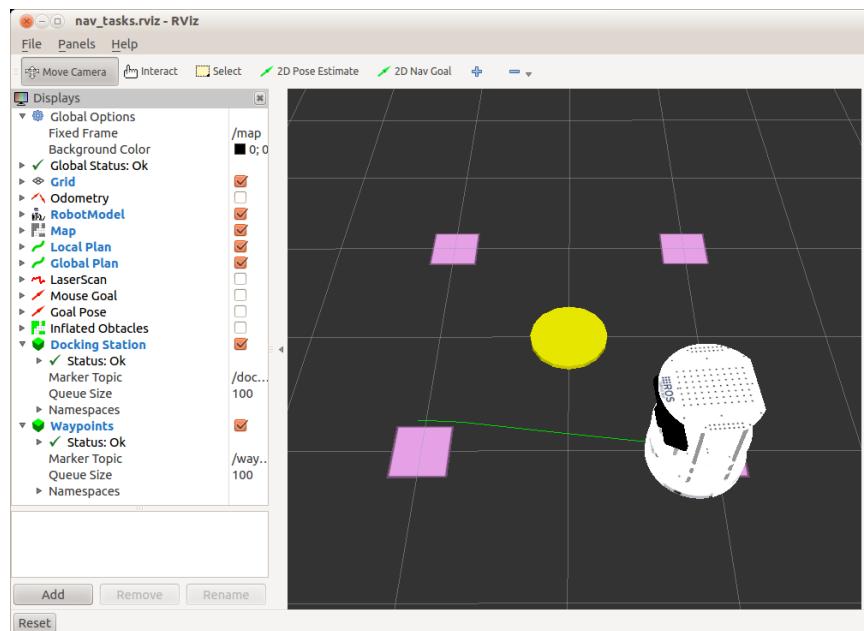


Use the sliders or text boxes to change the battery runtime or battery level.

We will use the battery simulator throughout this chapter to compare how different task frameworks enable us to handle a low battery condition while simultaneously performing other tasks.

3.2 A Common Setup for Running the Examples

All of our examples will share a common setup. There will be four waypoints (target locations) located at the corners of a square measuring 1 meter on a side. The docking station will be positioned in the middle of the square. Both the waypoints and the docking station will appear in `RViz` as visualization markers. The waypoints will appear as colored squares and the docking station as a yellow disc. These markers are not meant to have any depth so the robot can freely pass over them. The basic setup is shown below along with a fake TurtleBot.



The setup of these variables is taken care of by the file `task_setup.py` located in the directory `rbx2_tasks/src/rbx2_tasks`. In each of our examples, we will import this file to establish the basic environment. Some of the more important variables that are set in `task_setup.py` are as follows:

- `square_size` (default: 1.0 meter)

- `low_battery_threshold` (default: 50)
- `n_patrols` (default: 2)
- `move_base_timeout` (default: 10 seconds)

We also define the waypoint locations as well as the location of the docking station. Any of these can be changed to suit your needs.

Finally, we define a `move_base` client and a `cmd_vel` publisher for controlling the movement of the robot.

3.3 A Brief Review of ROS Actions

Since we will be using the `move_base` action quite a bit in this chapter, it's a good idea to review the general concept of a ROS action. Be sure to start with the [actionlib overview](#) on the ROS Wiki, then work through the online [tutorials](#) where examples are provided in both C++ and Python.

Recall that a ROS action expects a *goal* to be submitted by an action client. The action server will then typically provide *feedback* as progress is made toward the goal and a *result* when the goal is either succeeded, aborted, or preempted.

Perhaps the most familiar example of a ROS action is the `MoveBaseAction` used with the ROS navigation stack. The `move_base` package implements an action server that accepts a goal pose for the robot (position and orientation) and attempts to reach that goal by publishing `Twist` messages while monitoring odometry and laser scan data to avoid obstacles. Along the way, feedback is provided in the form of a time-stamped pose representing the state of the robot, as well as the goal status (e.g. ACTIVE, SUCCEEDED, ABORTED, etc). The result of the action is simply a time-stamped status message indicating that the goal succeeded or was aborted, preempted etc.

You can view the full definition of the `MoveBaseAction` using the command:

```
$ rosmsg show MoveBaseAction
```

To view just the feedback message syntax, use the command:

```
$ rosmsg show MoveBaseActionFeedback
```

And to see the list of possible statuses returned by the result, run the command:

```
$ rosmsg show MoveBaseActionResult
```

Recall from *Volume 1* that we programmed our robot to navigate a square by using a series of `move_base` actions. For each corner of the square, we sent the corresponding pose to the `move_base` action server, then waited for a result before submitting the next goal pose. However, suppose that as the robot moves between waypoints, we also want the robot to execute a number of subtasks at each location. For example, one task might be to look for a particular object and record its location or pick it up if the robot has an arm and gripper. At the same time, we want the robot to monitor its battery levels and navigate to the docking station, and so on.

All of this can be done using ROS actions but one would have to create an action server for each task, then coordinate the tasks with a number of `if-then` conditions or interacting callbacks among the actions. While certainly possible, the resulting code would be rather tedious. Fortunately, both SMACH and behavior trees help to make these more complicated situations easier to program.

3.4 A Patrol Bot Example

Suppose our robot's task is to patrol the perimeter of a square by navigating from corner to corner in sequence. If the battery level falls below a certain threshold, the robot should stop its patrol and navigate to the docking station. After recharging, the robot should continue the patrol where it left off.

The general components of the patrol task look something like this:

- **Initialization:**
 - set waypoint coordinates
 - set docking station coordinates
 - set number of patrols to perform
- **Tasks (ordered by priority):**
 - CHECK_BATTERY
 - RECHARGE
 - PATROL
- **Sensors and actuators:** ros-indigo-fake-localization
 - battery sensor; laser scanner, RGB-D camera, etc.
 - drive motors

The `CHECK_BATTERY` task simply sets a flag when the battery level falls below a set threshold.

The `RECHARGE` task can be broken down into the following subtasks:

`RECHARGE: NAV.Dock → CHARGE`

where `NAV.Dock` means navigate to the docking station. The `PATROL` task can also be broken down into a sequence of navigation subtasks:

`PATROL: NAV.0 → NAV.1 → NAV.2 → NAV.3`

where we have indexed each navigation task by the waypoint number (one for each corner of the square). The navigation tasks can then be implemented using standard ROS `MoveBaseAction` goals and the navigation stack as we did in *Volume 1*.

Before learning how to implement the Patrol Bot using `SMACH` or behavior trees, let's review how it could be done using a standard script.

3.5 The Patrol Bot using a Standard Script

Our script will subscribe to the battery level topic with a callback function that sets a `low_battery` flag to `True` if the level falls below a given threshold as follows:

```
def battery_cb(self, msg):
    if msg.data < self.low_battery_threshold:
        self.low_battery = True
    else:
        self.low_battery = False
```

This check is done at the same frequency as the rate at which messages are received on the battery level topic.

In the meantime, the our main control loop might start off looking something like this:

```
while n_patrols < max_patrols:
    if low_battery:
        recharge()
    else:
        patrol()
```

At the start of each patrol, we check the battery level and recharge if necessary. Otherwise, we start the patrol. Of course, this simple strategy won't work in practice since the battery is likely to run down in between battery checks when the robot is part way around the course. Let's see how we might correct this problem.

The `patrol()` routine moves the robot through the sequence of waypoints something like this:

```
def patrol():
    for location in waypoints:
        nav_to_waypoint(location)
```

When we write it out this way, we see that we should move the battery check inside the `nav_to_waypoint()` function:

```
def nav_to_waypoint(location):
    if low_battery:
        recharge()
    else:
        move_to(location)
```

At least now we are checking the battery level before we move on to each waypoint. However, the `move_to(location)` function could take some time to complete depending on how far away the next waypoint is located. So we really need to move the battery check even deeper and place it inside the `move_to()` routine.

In ROS, the `move_to()` function will likely implemented as call to the `MoveBaseAction` server so the battery check would be done inside the feedback callback for the `move_base` client. The result would look something like this:

```
move_base.send_goal(goal, feedback_cb=self.nav_feedback_cb)

def nav_feedback_cb(self, msg):
    if self.low_battery:
        self.recharge()
```

Now we are checking the battery status every time we receive a feedback message from the `MoveBaseAction` server which should be frequent enough to avoid a dead battery. The `recharge()` function will cancel the current `move_base` goal before sending a new goal to the `MoveBaseAction` server to navigate the robot to the docking station for charging.

The entire script outlined here can be found in the file `patrol_script.py` located in the `rbx2_tasks/nodes` directory.

Link to source: [patrol_script.py](#)

The script is fairly straightforward and will not be described in detail. However, you can test it out as follows.

First launch the fake TurtleBot in the ArbotiX simulator using the `fake_turtlebot.launch` file in the `rbx2_tasks/launch` subdirectory. This file will bring up the fake TurtleBot, a `move_base` action server with a blank map, and the battery simulator node with a default runtime of 60 seconds:

```
$ rosrun rbx2_tasks fake_turtlebot.launch
```

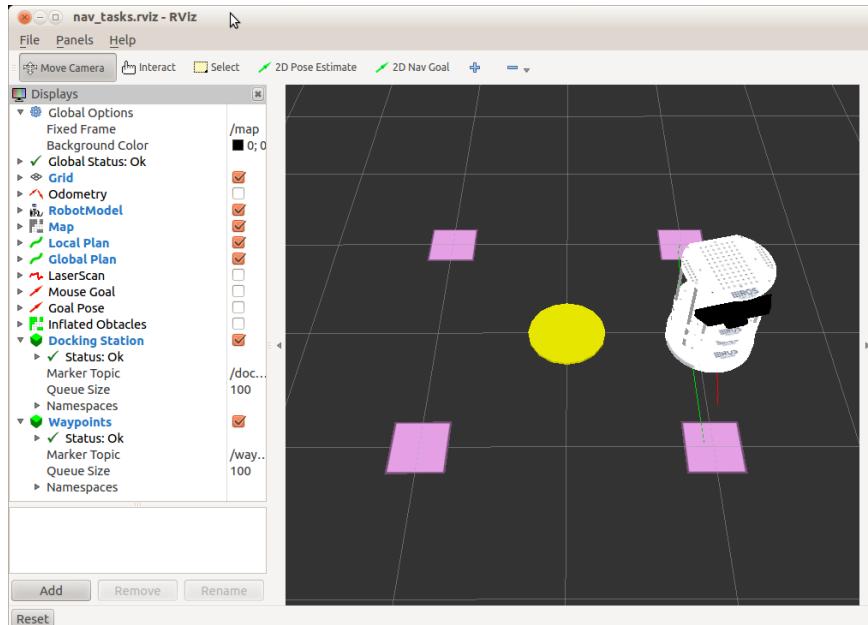
Next, bring up RViz with the `nav_tasks.rviz` config file:

```
$ rosrun rviz rviz -d `rospack find rbx2_tasks`/nav_tasks.rviz
```

Finally, run the `patrol_script.py` script:

```
$ rosrun rbx2_tasks patrol_script.py
```

The view in RViz should look something like this:



The robot should execute two loops around the square while monitoring its battery level. Whenever the battery falls below the threshold defined in the script (50), the robot will move to the circular docking station in the middle of the square. Once recharged (battery level is set back to 100), the robot should continue its patrol where it left off.

For example, if it did not quite make it to the second waypoint when it was forced to recharge, it should head back to that waypoint first before continuing the loop.

3.6 Problems with the Script Approach

The main issue with the standard script approach described above is that we had to bury the battery check deep within the navigation routine. While this works fine for this particular example, it becomes less efficient as we add more tasks to the robot's behavior. For example, suppose we want the robot to scan for the presence of a person at each waypoint by panning the camera left and right before moving on to the next location. Our patrol routine might then look like this:

```
def patrol():
    for location in waypoints:
        nav_to_waypoint(location)
        scan_for_humans()
```

While the camera is being panned back and forth, we still need to keep tabs on the battery level but now we are no longer running a `move_base` goal. This means we need to add a second battery check, this time to the head panning routine. If the battery level falls below threshold at this point, we no longer need to cancel a navigation goal; instead, we need to recenter the camera before moving to the docking station.

For each additional task, we need to add another battery check which results in redundancy in the code and makes our overall task executive less modular. Conceptually, the battery check should appear prominently near the top of our task hierarchy and we should really only have to check the battery level once on each pass through the task list.

Another shortcoming of the script approach is that we are making direct calls to ROS actions, topics and services rather than through reusable wrappers that hide the common details. For example, in addition to subscribing to the battery level topic, it is likely that we will also subscribe to other topics generated by additional sensors such as a depth camera, bump sensors, or a laser scanner. All these subscribers share a similar setup pattern in terms of a topic name, topic type and callback function. Task frameworks like SMACH define wrappers for ROS topics, services and actions that take care of the setup details thereby allowing these objects to be added to or deleted from a given task executive without having to repeat numerous lines of code.

3.7 SMACH or Behavior Trees?

[SMACH](#) is a Python library for building complex robot behaviors using hierarchical state machines. The [smach_ros](#) package provides tight integration with ROS topics, services and actions and there are a number of [SMACH tutorials](#) covering virtually all

of its features. For this reason, SMACH is usually a good place to start for most beginning ROS users. However, if you are not already familiar with [finite state machines](#), this approach might seem a little confusing at first. If you get stuck, there are other options available to you.

A relatively new approach called *behavior trees* first gained popularity among programmers of computer games. More recently, behavior trees have been used in robotics as well. We will use a package called [pi_trees](#) that was written by the author specifically for this book but can be used any way you see fit. Behavior trees organize robot tasks into a tree structure that make them relatively easy to conceptualize and program. Furthermore, many of the properties we require from a task executive such as prioritization (subsumption), pause and resume (preemption), hierarchical organization (subtasks), and condition checking are natural properties of behavior trees that we simply get "for free" once we set up the infrastructure.

3.8 SMACH: Tasks as State Machines

Before we can get started with SMACH, make sure you have the necessary ROS packages installed:

```
$ sudo apt-get install ros-indigo-smach ros-indigo-smach-ros \
  ros-indigo-executive-smach ros-indigo-smach-viewer
```

We will also make use of a simple GUI package called `python-easygui` in some of our examples so let's install that now as well:

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

SMACH enables us to program a sequence of actions using [finite state machines](#). As the name suggests, a state machine can be in one of a number of *states* at any given time. The machine also accepts *inputs*, and depending on the value of the inputs and the current state, the machine can make a *transition* to another state. As a result of making the transition, the state machine may also produce an *outcome*. Finite state machines are also called *automata* or *reactive systems* since we only need to know the machine's current state and its input(s) to predict its next action.

One of the simplest real-world examples of a state machine is a lock and key. The lock can be in one of two states, `LOCKED` or `UNLOCKED` while turning the key clockwise or counterclockwise is the input. (By convention, state names are written in all upper case.) The possible state transitions are from `LOCKED` to `UNLOCKED` and from `UNLOCKED` to `LOCKED` depending on the direction the key is turned. (For completeness, we could also specify the transitions `LOCKED` to `LOCKED` and `UNLOCKED` to `UNLOCKED`.) The outcomes are that the door can be opened or that it cannot.

The [SMACH](#) package provides a standalone Python library for creating state machines and a ROS wrapper for integrating the library with ROS topics, services and actions. The SMACH Wiki pages includes a number of [excellent tutorials](#) and the reader is encouraged to work through as many of them as possible. At the very least, it is essential to understand the [Getting Started](#) page. We will assume some familiarity with these tutorials as we walk through our examples.

SMACH has a lot of features and may appear a little overwhelming at first. So we will take each component in turn using examples that provide some visual feedback. But first, a short review of what you learned in the online tutorials.

3.8.1 SMACH review

We assume that you have worked through at least some of the [tutorials](#) on the SMACH Wiki so we will provide only a review of the essential concepts here. If you need more details on a given class or function, you can look it up in the [SMACH API](#). You can also go directly to the [source on GitHub](#).

SMACH *states* are Python classes that extend the `smach.State` class by overriding the `execute()` method to return one or more possible `outcomes`. The `execute` method can also take an optional argument defining a collection of `userdata` that can be used to pass information between states. The actual computations performed by the state can be essentially anything you want, but there are a number of predefined state types that can save a lot of unnecessary code. In particular, the [SimpleActionState](#) class turns a regular ROS `action` into a SMACH state. Similarly, the [MonitorState](#) wraps a ROS topic and the [ServiceState](#) handles ROS services. The [CBState](#) uses the `@smach.cb_interface` decorator to turn nearly any function you like into a SMACH state.

A SMACH *state machine* is another Python class (`smach.StateMachine`) that can contain a number of states. A state is added to a state machine by defining a set of transitions from the state's `outcomes` to other states in the machine. When a state machine is run, these transitions determine the flow of execution from state to state:

```
input → STATE_1 → {outcome, transition} → STATE_2  
input → STATE_2 → {outcome, transition} → STATE_3  
etc.
```

A state machine itself must have an outcome and can therefore be a state in another state machine. In this way, state machines can be nested to form a hierarchy. For example, a state machine called "clean house" could contain the nested state machines "vacuum living room", "mop kitchen", "wash tub" and so on. The higher level state machine

"clean house" will determine how the outcomes of the nested state machines map into the overall outcome (e.g. "all tasks complete" or "not all tasks complete").

There are a number of predefined SMACH containers that can also save you a lot of programming. The [Concurrence](#) container returns an outcome that depends on more than one state and allows one state to preempt another. The [Sequence](#) container automatically generates sequential transitions among the states that are added to it. And the [Iterator](#) container allows you to loop through one or more states until some condition is met. We will learn more about these containers as we need them.

3.8.2 Patrolling a square using SMACH

In *Volume 1*, we programmed our robot to navigate around a square using a variety of methods including `move_base` actions. And earlier in this chapter we used a Python script to do the same thing while also monitoring a simulated battery level. Let us now see how we can use SMACH to accomplish the same goal.

In this section, we will leave out the battery check and simply move the robot around the square. In the next section we will add in the battery check and enable the robot to recharge when necessary.

One way to conceptualize the patrol problem is that we want the robot to be in one of four states—namely, the poses that define the four corners of the square. Furthermore, we want the robot to transition through these states in a particular order. Equivalently, we could say that we want the robot to execute four tasks; namely, to move to each of the corner locations in sequence.

Let us name the four states `NAV_STATE_0` through `NAV_STATE_3`. Our state machine will then be defined by the following states and transitions:

```
NAV_STATE_0 → NAV_STATE_1  
NAV_STATE_1 → NAV_STATE_2  
NAV_STATE_2 → NAV_STATE_3  
NAV_STATE_3 → NAV_STATE_0
```

Here we have defined the last transition to take us back to the starting state which therefore causes the whole state machine to repeat the cycle. If instead we want the robot to stop after navigating the square just once, we can define another state `NAV_STATE_4` to have the same goal pose as `NAV_STATE_0` and then change the last transition above to:

```
NAV_STATE_3 → NAV_STATE_4
```

We could then terminate the machine (and therefore stop the robot) by adding a final transition to the empty state:

```
NAV_STATE_4 → ''
```

In SMACH, the transitions depend on the outcome of the previous state. In the case of our robot moving between locations, the outcome can be "succeeded", "aborted" or "preempted".

These ideas are implemented in the script [patrol_smach.py](#) found in the `rbx2_tasks/nodes` directory. To save space, we won't display the entire listing but will focus on the key sections instead. (You can click on the script name above to go to the online listing or bring up the script in your own editor.)

In the import block near the top of the script, we need to bring in the SMACH objects we want to use:

```
from smach import StateMachine
from smach_ros import SimpleActionState, IntrospectionServer
```

Will we need the `StateMachine` object to build the overall state machine, the `SimpleActionState` to wrap our calls to `move_base` and the `IntrospectionServer` so we can use the `smach_viewer`.

As you know from the [online tutorial](#), the `SimpleActionState` type allows us to wrap a regular ROS action into a SMACH state. Assuming we have already assigned the corner poses to a Python list called `waypoints`, the block of code that turns these poses into simple action states looks like this:

```
nav_states = list()

for waypoint in waypoints:
    nav_goal = MoveBaseGoal()
    nav_goal.target_pose.header.frame_id = 'map'
    nav_goal.target_pose.pose = waypoint

    move_base_state = smach_ros.SimpleActionState('move_base', MoveBaseAction,
goal=nav_goal, exec_timeout=rospy.Duration(10.0))

    nav_states.append(move_base_state)
```

First we create an empty list called `nav_states` to hold our navigation states, one for each corner of the square. Next, we loop through each of the waypoints, creating a standard `MoveBaseGoal` using that waypoint as the desired pose. We then turn this goal into a SMACH state using the statement:

```

move_base_state = smach_ros.SimpleActionState('move_base', MoveBaseAction,
goal=nav_goal, exec_timeout=rospy.Duration(10.0))

```

where we have used the `SimpleActionState` state type to wrap the `MoveBaseAction` action into a state. The `SimpleActionState` class constructor takes the action topic name as the first argument and the action type as the second argument. It also supports the keyword arguments `goal` and `exec_timeout` for specifying the goal of the action and the time we are willing to wait for it to be achieved (10 seconds in the case above.) Finally, we append the state to the `nav_states` list.

The predefined outcomes for a `SimpleActionState` are `succeeded`, `aborted` or `preempted`. The next step is to construct the overall state machine using these outcomes and states:

```

# Initialize the state machine
self.sm_patrol = StateMachine(outcomes=['succeeded', 'aborted', 'preempted'])

# Add the states to the state machine with the appropriate transitions
with self.sm_patrol:
    StateMachine.add('NAV_STATE_0', nav_states[0],
transitions={'succeeded': 'NAV_STATE_1', 'aborted': 'NAV_STATE_1'})
    StateMachine.add('NAV_STATE_1', nav_states[1],
transitions={'succeeded': 'NAV_STATE_2', 'aborted': 'NAV_STATE_2'})
    StateMachine.add('NAV_STATE_2', nav_states[2],
transitions={'succeeded': 'NAV_STATE_3', 'aborted': 'NAV_STATE_3'})
    StateMachine.add('NAV_STATE_3', nav_states[3],
transitions={'succeeded': 'NAV_STATE_0', 'aborted': 'NAV_STATE_0'})

```

First we initialize our patrol state machine with possible outcomes '`succeeded`', '`aborted`' and '`preempted`'. The actual outcome will be determined by the states we add to the state machine and what outcomes they produce when executed.

Next, we add each navigation state to the state machine together with a dictionary of transitions from outcomes to the next state. The first argument in each line is an arbitrary name we assign to the state so that the transitions have something to refer to. By convention, these state names are written in upper case. For example, the first line above adds the state `nav_states[0]` to the state machine and gives it the name `NAV_STATE_0`. The transitions for this state tells us that if the state succeeds (the robot makes it to the goal location), we want the next state to be `NAV_STATE_1` which is defined on the second line and represents the second goal pose stored in the state `nav_states[1]`.

Note how we also map the outcome '`aborted`' to the next state. While this is optional and not always desired, it works nicely with `MoveBase` goals since the base planner might not always succeed in getting the robot to the current goal due to obstacles or time

constraints. In such cases, the outcome would be aborted and instead of simply stopping the robot, we continue on to the next goal.

Note also how the final state transition returns to the first state, NAV_STATE_0. In this case, the state machine and the robot will continue to loop around the square indefinitely. If we want the robot to stop after the first loop, we can create the following state machine instead:

```
with self.sm_patrol:
    StateMachine.add('NAV_STATE_0', nav_states[0],
transitions={'succeeded': 'NAV_STATE_1', 'aborted': 'NAV_STATE_1'})
    StateMachine.add('NAV_STATE_1', nav_states[1],
transitions={'succeeded': 'NAV_STATE_2', 'aborted': 'NAV_STATE_2'})
    StateMachine.add('NAV_STATE_2', nav_states[2],
transitions={'succeeded': 'NAV_STATE_3', 'aborted': 'NAV_STATE_3'})
    StateMachine.add('NAV_STATE_3', nav_states[3],
transitions={'succeeded': 'NAV_STATE_4', 'aborted': 'NAV_STATE_4'})
    StateMachine.add('NAV_STATE_4', nav_states[0],
transitions={'succeeded': '', 'aborted': ''})
```

The final state NAV_STATE_4 is assigned the same pose state as the starting point and we map both outcomes into the empty state thus terminating the state machine and stopping the robot. The script `patrol_smach.py` implements this version of the state machine but places the execution in a loop that enables us to control the number of times the robot completes its patrol as we show next.

To execute the state machine, we use the loop:

```
while self.n_patrols == -1 or self.patrol_count < self.n_patrols:
    sm_outcome = self.sm_patrol.execute()
    self.patrol_count += 1
    rospy.loginfo("FINISHED PATROL NUMBER: " + str(self.patrol_count))
```

The parameter `self.n_patrols` is defined in our `task_setup.py` file which in turn reads it from the ROS parameter server with a default value of 3. (We use the special value of -1 if we want the robot to loop forever.) The counter `self.patrol_count` is also defined in `task_setup.py` and is initialized to 0.

On each pass through the loop, we run `self.sm_patrol.execute()`. The `execute()` method sets the state machine in motion. If the machine terminates, as it does after the final state transition above, then the overall outcome for the state machine will be available in the variable `sm_outcome`.

Note that, unlike a regular script, we cannot simply place a `while` loop around the actual states of our state machine since that would result in looping through the *construction* of the states before they are even run.

We will also tend to add the following pair of lines to most of our scripts:

```
intro_server = IntrospectionServer('nav_square', self.sm_patrol, '/SM_ROOT')
intro_server.start()
```

The SMACH Introspection server enables us to view the running state machine in the graphical `smach_viewer` utility as we will see below.

3.8.3 Testing SMACH navigation in the ArbotiX simulator

Let's try it out the `patrol_smach.py` script using the ArbotiX simulator.

First, run the `fake_turtlebot.launch` file in the `rbx2_tasks` package. This file will bring up a fake TurtleBot, a `move_base` action server with a blank map and the fake battery simulator with a default runtime of 60 seconds, although we won't be using the battery for this example:

```
$ rosrun rbtasks fake_turtlebot.launch
```

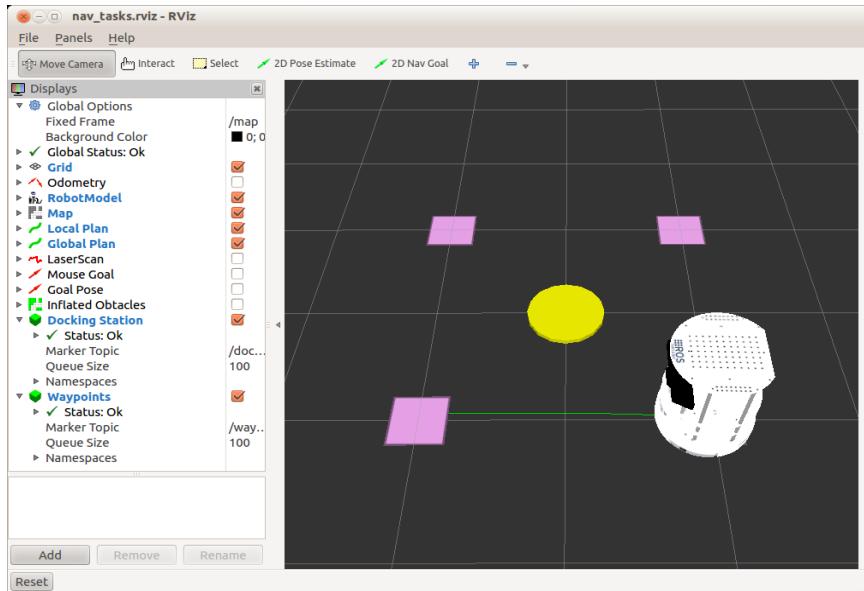
Next, terminate any running instances of `RViz`, then bring it up with the `nav_tasks` config file:

```
$ rosrun rviz rviz -d `rospack find rbtasks`/nav_tasks.rviz
```

Make sure you can see the `RViz` window in the foreground, then run the `patrol_smach.py` script:

```
$ rosrun rbtasks patrol_smach.py
```

You should see the robot move around the square three times and then stop. The view in `RViz` should look something like this:



At the same time, you should see the following messages in the terminal you used to launch the `patrol_smach.py` script:

```
Starting Tasks
[INFO] [WallTime: 1378991934.456861] State machine starting in initial state 'NAV_STATE_0'
with userdata:
[]
[WARN] [WallTime: 1378991934.457513] Still waiting for action server 'move_base' to start...
is it running?
[INFO] [WallTime: 1378991934.680443] Connected to action server 'move_base'.
[INFO] [WallTime: 1378991934.882364] Success rate: 100.0
[INFO] [WallTime: 1378991934.882795] State machine transitioning 'NAV_STATE_0':'succeeded'-->'NAV_STATE_1'
[INFO] [WallTime: 1378991940.684410] Success rate: 100.0
[INFO] [WallTime: 1378991940.685345] State machine transitioning 'NAV_STATE_1':'succeeded'-->'NAV_STATE_2'
[INFO] [WallTime: 1378991946.487312] Success rate: 100.0
[INFO] [WallTime: 1378991946.487737] State machine transitioning 'NAV_STATE_2':'succeeded'-->'NAV_STATE_3'
[INFO] [WallTime: 1378991952.102620] Success rate: 100.0
[INFO] [WallTime: 1378991952.103259] State machine transitioning 'NAV_STATE_3':'succeeded'-->'NAV_STATE_4'
[INFO] [WallTime: 1378991957.705305] Success rate: 100.0
[INFO] [WallTime: 1378991957.705821] State machine terminating
'NAV_STATE_4':'succeeded':'succeeded'
[INFO] [WallTime: 1378991957.706164] State Machine Outcome: succeeded
[INFO] [WallTime: 1378991958.514761] Stopping the robot...
```

Here we see that SMACH reports when each state transition occurs and also the final outcome of the state machine as a whole. The lines that report the "Success rate" is something extra created by our `patrol_smach.py` script that we will examine in more detail in the next section.

We can also see a graph of the running state machine by using the `smach_viewer.py` utility. To fire up the viewer, open another terminal and run:

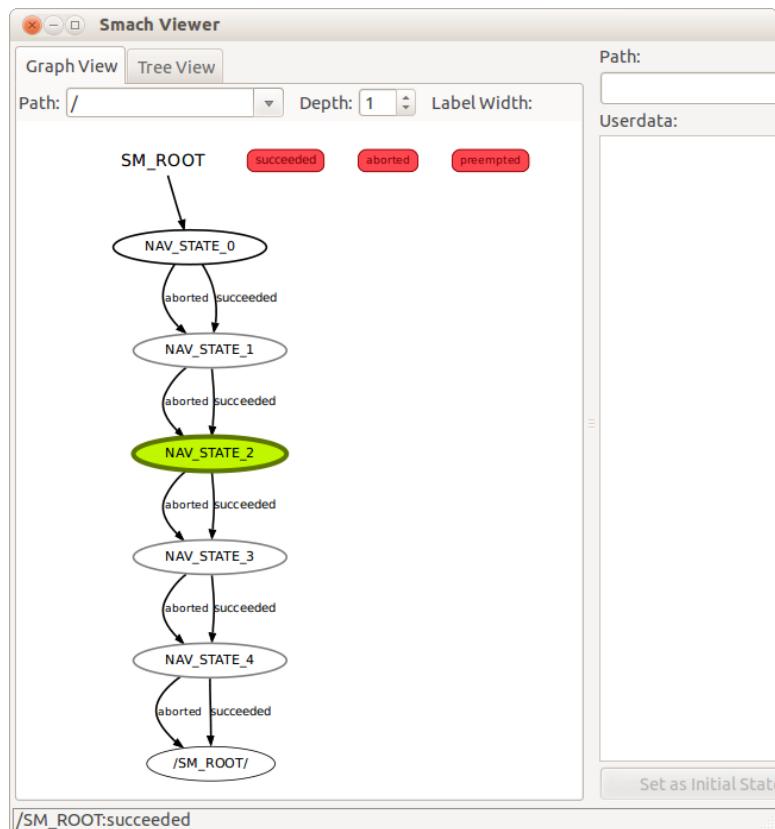
```
$ rosrun smach_viewer smach_viewer.py
```

Now run the `patrol_smach.py` script again:

```
$ rosrun rbx2_tasks patrol_smach.py
```

NOTE: As of this writing, the SMACH viewer described below is [not currently working under ROS Indigo](#). You can subscribe to the issue on Github to receive a notification whenever a comment has been posted.

The display in the SMACH viewer should look something like this:



As the robot moves from state to state (i.e. moves from location to location around the square), you should see the appropriate state highlighted in green in the viewer. For a complete description of the SMACH viewer GUI, see the [smach_viewer](#) Wiki page.

3.8.4 Accessing results from a SimpleActionState

In our current state machine we decided to move on to the next state even if the current transition had to be aborted. But it might be useful to keep a count of how often a goal was successful. For example, if this were a patrol robot and the success rate fell below some threshold, then we might suspect that something was wrong with the robot or that something was getting in its way.

The SMACH SimpleActionState constructor allows us to assign a callback function to obtain the result of the action. The syntax looks like this:

```
move_base_state = SimpleActionState('move_base', MoveBaseAction,
goal=nav_goal, result_cb=self.move_base_result_cb)
```

Note how we use the keyword `result_cb` to assign a function to handle the result. Our `move_base_result_cb` function then looks like this:

```
def move_base_result_cb(self, userdata, status, result):
    if status == actionlib.GoalStatus.SUCCEEDED:
        self.n_succeeded += 1
    elif status == actionlib.GoalStatus.ABORTED:
        self.n_aborted += 1
    elif status == actionlib.GoalStatus.PREEMPTED:
        self.n_preempted += 1

    try:
        rospy.loginfo("Success rate: " + str(100.0 * self.n_succeeded /
(self.n_succeeded + self.n_aborted + self.n_preempted)))
    except:
        pass
```

The callback function takes three arguments: the current `userdata` (which we will explore more later on), as well as the `status` and `result` returned by the underlying ROS action (in this case, `move_base`). As it turns out, the `move_base` action does not use the `result` field. Instead, it places the results in the `status` field which is why our test condition above checks the value of the `status` field.

As you can see from the above code, our callback simply increments the counters for the number of `move_base` attempts that are successful, aborted or preempted. We also print out the percent success so far.

3.8.5 SMACH Iterators

In the `patrol_smach.py` script, we repeated the patrol by placing the machine execution inside a while loop. We will now show how we can accomplish the same result using the SMACH [Iterator](#) container. The new script is called `patrol_smach_iterator.py` and can be found in the `rbx2_tasks/nodes` directory. Since most of the script is the same as `patrol_smach.py`, we will only highlight the differences.

The key lines of code in the program are as follows:

```
# Initialize the top level state machine
self.sm = StateMachine(outcomes=['succeeded', 'aborted', 'preempted'])

with self.sm:
    # Initialize the iterator
    self.sm_patrol_iterator = Iterator(outcomes =
        ['succeeded', 'preempted', 'aborted'],
        input_keys = [],
        it = lambda: range(0, self.n_patrols),
        output_keys = [],
        it_label = 'index',
        exhausted_outcome = 'succeeded')

    with self.sm_patrol_iterator:
        # Initialize the patrol state machine
        self.sm_patrol = StateMachine(outcomes =
            ['succeeded', 'aborted', 'preempted', 'continue'])

        # Add the states to the state machine with the appropriate transitions
        with self.sm_patrol:
            StateMachine.add('NAV_STATE_0', nav_states[0],
transitions={'succeeded': 'NAV_STATE_1', 'aborted': 'NAV_STATE_1', 'preempted': 'NAV_STATE_1'})
            StateMachine.add('NAV_STATE_1', nav_states[1],
transitions={'succeeded': 'NAV_STATE_2', 'aborted': 'NAV_STATE_2', 'preempted': 'NAV_STATE_2'})
            StateMachine.add('NAV_STATE_2', nav_states[2],
transitions={'succeeded': 'NAV_STATE_3', 'aborted': 'NAV_STATE_3', 'preempted': 'NAV_STATE_3'})
            StateMachine.add('NAV_STATE_3', nav_states[3],
transitions={'succeeded': 'NAV_STATE_4', 'aborted': 'NAV_STATE_4', 'preempted': 'NAV_STATE_4'})
            StateMachine.add('NAV_STATE_4', nav_states[0],
transitions={'succeeded': 'continue', 'aborted': 'continue', 'preempted': 'continue'})

        # Close the sm_patrol machine and add it to the iterator
        Iterator.set_contained_state('PATROL_STATE', self.sm_patrol,
loop_outcomes=['continue'])

    # Close the top level state machine
    StateMachine.add('PATROL_ITERATOR', self.sm_patrol_iterator,
{'succeeded': 'succeeded', 'aborted': 'aborted'})
```

Let's now break this down.

```
# Initialize the top level state machine
self.sm = StateMachine(outcomes=['succeeded', 'aborted', 'preempted'])

with self.sm:
    # Initialize the iterator
    self.sm_patrol_iterator = Iterator(outcomes =
        ['succeeded', 'preempted', 'aborted'],
        input_keys = [],
        it = lambda: range(0, self.n_patrols),
        output_keys = [],
        it_label = 'index',
        exhausted_outcome = 'succeeded')
```

After initializing the top level state machine, we construct an `Iterator` that will loop `self.n_patrols` times. The core of the `Iterator` is the `it` argument that is set to a list of objects to be iterated over. In our case, we define the list using the Python `lambda` function to create a list of integers over the `range(0, self.n_patrols)`. The `it_label` argument (set to 'index' in our case) holds the current value of the key as it iterates over the list. The `exhausted_outcome` argument sets the outcome to emit when the `Iterator` has reached the end of its list.

```
with self.sm_patrol_iterator:
    # Initialize the patrol state machine
    self.sm_patrol =
StateMachine(outcomes=['succeeded', 'aborted', 'preempted', 'continue'])

    # Add the states to the state machine with the appropriate transitions
    with self.sm_patrol:
        StateMachine.add('NAV_STATE_0', nav_states[0],
transitions={'succeeded': 'NAV_STATE_1', 'aborted': 'NAV_STATE_1', 'preempted': 'NAV_STATE_1'})
        StateMachine.add('NAV_STATE_1', nav_states[1],
transitions={'succeeded': 'NAV_STATE_2', 'aborted': 'NAV_STATE_2', 'preempted': 'NAV_STATE_2'})
        StateMachine.add('NAV_STATE_2', nav_states[2],
transitions={'succeeded': 'NAV_STATE_3', 'aborted': 'NAV_STATE_3', 'preempted': 'NAV_STATE_3'})
        StateMachine.add('NAV_STATE_3', nav_states[3],
transitions={'succeeded': 'NAV_STATE_4', 'aborted': 'NAV_STATE_4', 'preempted': 'NAV_STATE_4'})
        StateMachine.add('NAV_STATE_4', nav_states[0],
transitions={'succeeded': 'continue', 'aborted': 'continue', 'preempted': 'continue'})
```

Next we create the patrol state machine as we have done before with two differences. First, the state machine is tucked inside the "with `self.sm_patrol_iterator`" statement. Second, we have added a new outcome labeled 'continue' to both the overall patrol machine and the final state, `NAV_STATE_4`. Why we do this will become clear in the final lines below.

```

        Iterator.set_contained_state('PATROL_STATE', self.sm_patrol,
loop_outcomes=['continue'])

# Close the top level state machine
StateMachine.add('PATROL_ITERATOR', self.sm_patrol_iterator,
{'succeeded': 'succeeded', 'aborted': 'aborted'})

```

The first line above adds the patrol state machine to the `Iterator` as the contained state and sets the `loop_outcomes` parameter to 'continue'. This means that when the contained state machine emits an outcome of 'continue', the `Iterator` will move to the next value in its list. As you can see from our patrol state machine, `NAV_STATE_4` maps all outcomes to 'continue', so the `Iterator` will start the next cycle once we have completed `NAV_STATE_4`. If the `Iterator` reaches the end of its list, it will terminate with an outcome set by the `exhausted_outcomes` parameter which we set to 'succeeded' when constructing the `Iterator`.

The final line adds the `Iterator` as a state in the overall state machine.

To test the script, make sure you have the fake TurtleBot up and running as in the previous sections as well as `RViz` with the `nav_tasks.rviz` config file, then run the iterator script:

```
$ rosrun rbx2_tasks patrol_smach_iterator.py
```

The result should be the same as before: the robot should make two complete patrols of the square.

3.8.6 Executing commands on each transition

Suppose that we want the robot to execute one or more functions each time it transitions from one state to the next. For example, perhaps we want the patrol robot to scan for people in each room and if it sees someone, says hello and then continues.

This type of execution can be accomplished using a transition callback function. Our demo script is called `patrol_smach_callback.py` located in the directory `rbx2_tasks/nodes`. The majority of the script is the same as the `patrol_smach.py` script described earlier so we will only highlight the differences.

First we set the transition callback on the state machine as follows:

```
self.sm_patrol.register_transition_cb(self.transition_cb, cb_args=[])
```

The `register_transition_cb()` function takes two arguments: the callback function that we want to execute, and a list of callback arguments, which can just be an empty list as we have used here. Our callback function, `self.transition_cb()` then looks like this:

```
def transition_cb(self, userdata, active_states, *cb_args):
    if self.rand.randint(0, 3) == 0:
        rospy.loginfo("Greetings human!")http://www.projectorcentral.com/BenQ-
MX522-projection-calculator-pro.htm
        rospy.sleep(1)
        rospy.loginfo("Is everything OK?")
        rospy.sleep(1)
    else:
        rospy.loginfo("Nobody here.")
```

Here we have simply pretended to check to the presence of a person by randomly selecting a number between 0 and 3. If it comes up 0, we emit a greeting, otherwise, we report that no one is present. Of course, in a real application, you might include code that scans the room by panning the robot's camera and uses text-to-speech to speak to talk to someone if they are detected.

To test the script, make sure you have the fake TurtleBot up and running as in the previous sections as well as `RViz` with the `nav_tasks.rviz` config file, then run the iterator script:

```
$ rosrun rbx2_tasks patrol_smach_callback.py
```

Unlike the previous scripts, this one will run indefinitely until you abort it with `Ctrl-C`.

3.8.7 Interacting with ROS topics and services

Suppose we want to monitor the robot's battery level and if it falls below a certain threshold, the robot should pause or abort what it is doing, navigate to the docking station and recharge, then continue the previous task where it left off. To begin, we need to know how to use SMACH to monitor the battery level. Let's use the fake battery simulator introduced earlier to illustrate the process.

SMACH provides the pre-defined states `MonitorState` and `ServiceState` to interact with ROS topics and services from within a state machine. We'll use a `MonitorState` to track the simulated battery level and a `ServiceState` to simulate a recharge. Before integrating the battery check into our Patrol Bot state machine, let's look at a simpler state machine that simply monitors the battery level and then issues a recharge service call when the level falls below threshold.

The demo script is called `monitor_fake_battery.py` script in the `rbx2_tasks/nodes` directory and looks like the following.

Link to source: [monitor_fake_battery.py](#)

```
1  #!/usr/bin/env python
2
3  import rospy
4  from smach import State, StateMachine
5  from smach_ros import MonitorState, ServiceState, IntrospectionServer
6  from rbx2_msgs.srv import *
7  from std_msgs.msg import Float32
8
9  class main():
10     def __init__(self):
11         rospy.init_node('monitor_fake_battery', anonymous=False)
12
13         rospy.on_shutdown(self.shutdown)
14
15         # Set the low battery threshold (between 0 and 100)
16         self.low_battery_threshold = rospy.get_param('~low_battery_threshold',
17
17
18         # Initialize the state machine
19         sm_battery_monitor = StateMachine(outcomes[])
20
21         with sm_battery_monitor:
22             # Add a MonitorState to subscribe to the battery level topic
23             StateMachine.add('MONITOR_BATTERY',
24                             MonitorState('battery_level', Float32, self.battery_cb),
25                             transitions={'invalid': 'RECHARGE_BATTERY',
26                                           'valid': 'MONITOR_BATTERY',
27                                           'preempted': 'MONITOR_BATTERY'},)
28
29             # Add a ServiceState to simulate a recharge using the
30             # set_battery_level service
31             StateMachine.add('RECHARGE_BATTERY',
32                             ServiceState('battery_simulator/set_battery_level',
33                             SetBatteryLevel, request=100),
34                             transitions={'succeeded': 'MONITOR_BATTERY',
35                                           'aborted': 'MONITOR_BATTERY',
36                                           'preempted': 'MONITOR_BATTERY'})
```

```

46         rospy.loginfo("Battery Level: " + str(msg))
47         if msg.data < self.low_battery_threshold:
48             return False
49         else:
50             return True
51
52     def shutdown(self):
53         rospy.loginfo("Stopping the battery monitor...")
54         rospy.sleep(1)
55
56 if __name__ == '__main__':
57     try:
58         main()
59     except rospy.ROSInterruptException:
60         rospy.loginfo("Battery monitor finished.")

```

Let's break down the key lines of the script.

```

4  from smach import State, StateMachine
5  from smach_ros import MonitorState, ServiceState, IntrospectionServer
6  from rbx2_msgs.srv import *
7  from std_msgs.msg import Float32

```

In addition to the usual `State` and `StateMachine` objects, we also import `MonitorState` and `ServiceState` from `smach_ros`. Since the service we will be connecting to (`set_battery_level`) lives in the `rbx2_msgs` package, we will import all service definitions from there. Finally, since the battery level is published using the `Float32` message type, we also import this type from the ROS `std_msgs` package.

```
16  self.low_battery_threshold = rospy.get_param('~low_battery_threshold', 50)
```

The `low_battery_threshold` is read in as a ROS parameter with a default of 50 under the assumption that 100 is fully charged.

```
19  sm_battery_monitor = StateMachine(outcomes=[])
```

We create a top-level state machine called `sm_battery_monitor` and assign it an empty outcome since it won't actually yield a result on its own.

```

21  with sm_battery_monitor:
22      # Add a MonitorState to subscribe to the battery level topic
23      StateMachine.add('MONITOR_BATTERY',
24          MonitorState('battery_level', Float32, self.battery_cb),
25          transitions={'invalid': 'RECHARGE_BATTERY',
26                      'valid': 'MONITOR_BATTERY',
27                      'preempted': 'MONITOR_BATTERY'},)

```

The first state we add to the state machine is called MONITOR_BATTERY and it uses the SMACH MonitorState to keep tabs on the battery level. The arguments to the MonitorState constructor are the topic we want to monitor, the message type for that topic and a callback function (here called `self.battery_cb`) that is described below. The key names in the transitions dictionary come from the pre-defined outcomes for the MonitorState type which are valid, invalid and preempted although the valid and invalid outcomes are actually represented by the values True and False, respectively. In our case, our callback function will use the invalid outcome to mean that the battery level has fallen below threshold so we map this key to a transition to the RECHARGE_BATTERY state described next.

```
30 StateMachine.add('RECHARGE_BATTERY',
31     ServiceState('battery_simulator/set_battery_level',
32     SetBatteryLevel, request=100),
33     transitions={'succeeded': 'MONITOR_BATTERY',
34                 'aborted': 'MONITOR_BATTERY',
35                 'preempted': 'MONITOR_BATTERY'})
```

The second state we add to the state machine is the RECHARGE_BATTERY state that uses the SMACH ServiceState. The arguments to the ServiceState constructor are the service name, the service type, and the request value to send to the service. The SetBatteryLevel service type is set in the `rbx2_msgs` package which is why we imported `rbx2_msgs.srv` at the top of our script. Setting the `request` value to 100 essentially performs a simulated recharge of the battery to full strength. The ServiceState returns the traditional outcomes of succeeded, aborted and preempted. We map all outcomes back to the MONITOR_BATTERY state.

The final part of the script that requires explaining is the callback function for our MonitorState:

```
35 def battery_cb(self, userdata, msg):
36     rospy.loginfo("Battery Level: " + str(msg))
37     if msg.data < self.low_battery_threshold:
38         return False
39     else:
40         return True
```

Any callback function assigned to a MonitorState state automatically receives the messages being published to the topic being subscribed to and the state's `userdata`. In this callback, we will only make use of the topic messages (passed in as the `msg` argument) and not the `userdata` argument.

Recall that the messages we are monitoring are simple `Float32` numbers representing the charge level of the simulated battery. The first line of the `battery_cb` function

above simply displays the level to the screen. We then test the level against the `low_battery_threshold` set earlier in the script. If the current level is below the `low_battery_threshold`, we return `False` which is the equivalent of `invalid` when it comes to a `MonitorState`. Otherwise, we return `True` which is the same as an outcome of `valid`. As we saw earlier, when the `MONITOR_BATTERY` state generates an outcome of `invalid` via its callback function, it transitions to the `RECHARGE_BATTERY` state.

Now that we understand the function of the script, let's try it out. First make sure the fake battery is running. If you still have the `fake_turtlebot.launch` file running from previous sections, that will work. Otherwise, you can launch the fake battery on its own:

```
$ roslaunch rbx2_utils battery_simulator.launch
```

Then fire up the `monitor_fake_battery.py` script:

```
$ rosrun rbx2_tasks monitor_fake_battery.py
```

You should then see a series of message similar to the following:

```
[INFO] [WallTime: 1379002809.494314] State machine starting in initial state  
'MONITOR_BATTERY' with userdata:  
[]  
[INFO] [WallTime: 1379002812.213581] Battery Level: data: 70.0  
[INFO] [WallTime: 1379002813.213005] Battery Level: data: 66.6666641235  
[INFO] [WallTime: 1379002814.213802] Battery Level: data: 63.3333320618  
[INFO] [WallTime: 1379002815.213758] Battery Level: data: 60.0  
[INFO] [WallTime: 1379002816.213793] Battery Level: data: 56.6666679382  
[INFO] [WallTime: 1379002817.213799] Battery Level: data: 53.3333320618  
[INFO] [WallTime: 1379002818.213819] Battery Level: data: 50.0  
[INFO] [WallTime: 1379002819.213816] Battery Level: data: 46.6666679382  
[INFO] [WallTime: 1379002819.219875] State machine transitioning  
'MONITOR_BATTERY':'invalid'-->'RECHARGE_BATTERY'  
[INFO] [WallTime: 1379002819.229251] State machine transitioning  
'RECHARGE_BATTERY':'succeeded'-->'MONITOR_BATTERY'  
[INFO] [WallTime: 1379002820.213889] Battery Level: data: 100.0  
[INFO] [WallTime: 1379002821.213805] Battery Level: data: 96.6666641235  
[INFO] [WallTime: 1379002822.213807] Battery Level: data: 93.3333358765  
etc
```

The first INFO message above indicates that the state machine is initialized in the MONITOR_STATE state. The next series of lines shows a count down of the battery level which is the result of our `rospy.loginfo()` statements in the `battery_cb` function described earlier. When the level falls below threshold, we see the MONITOR_STATE returns an outcome of `invalid` which triggers a state transition to the RECHARGE_BATTERY state. Recall that the RECHARGE_STATE calls the `set_battery_level` service and sets the battery level back to 100. It then returns an outcome of `succeeded` which triggers a state transition back to MONITOR_STATE. The process then continues indefinitely.

3.8.8 Callbacks and Introspection

One of the strengths of SMACH is the ability to determine the state of the machine at any given time and to set callbacks on state transitions or termination. For example, if a given state machine is preempted, we might want to know the last state it was in before it was interrupted. We will use this feature in the next section to determine where to continue the patrol after a recharge. We can also use introspection to improve on the way we track the success rate: instead of simply keeping a count of the number of waypoints reached, we can actually record the waypoint ID as well.

You can find details on all possible callbacks from the online [SMACH API](#). The callback we will use most often is fired on each state transition within a given state machine. The callback function itself is set using the `register_transition_cb()` method on the state machine object. For example, to register a transition callback on our patrol state machine in the `patrol_smach.py` script, we would use the syntax:

```
self.sm_patrol.register_transition_cb(self.patrol_transition_cb, cb_args=[])
```

We would then define the function `self.patrol_transition_cb` to execute whatever code we want on each state transition:

```
def patrol_transition_cb(self, userdata, active_states, *cb_args):
    Do something awesome with userdata, active_states or cb_args
```

Here we see that a transition callback always has access to `userdata`, `active_states` and any callback arguments that were also passed. In particular, the variable `active_states` holds the current state of the state machine and we will use this in the next section to determine where the robot was when it was interrupted to recharge its battery.

3.8.9 Concurrent tasks: Adding the battery check to the patrol routine

Now that we know how to patrol the square and check the battery, it is time to put the two together. We want a low battery signal to take top priority so that the robot stops its patrol and navigates to the docking station. Once recharged, we want the robot to continue its patrol beginning with the last waypoint it was heading for before being interrupted.

SMACH provides the [Concurrence](#) container for running tasks in parallel and enabling one task to preempt the other when a condition is met. So we will set up our new state machine with a Concurrence container which will hold the navigation machine and the battery check machine. We will then set up the container so that the battery check can preempt navigation when the battery level falls below threshold.

Before looking at the code, let's try it out. Terminate the `monitor_battery.py` node if it is still running from an earlier session. (It may take a while to respond to Ctrl-C before the node eventually dies.)

If the `fake_turtlebot.launch` file is not already running, bring it up now. Recall that this file also launches the fake battery simulator with a 60 second battery runtime:

```
$ rosrun smach_viewer smach_viewer.py
```

If you don't already have it running, bring up the SMACH viewer with the command:

```
$ rosrun smach_viewer smach_viewer.py
```

Bring up RViz with the `nav_tasks` config file if it is not already running:

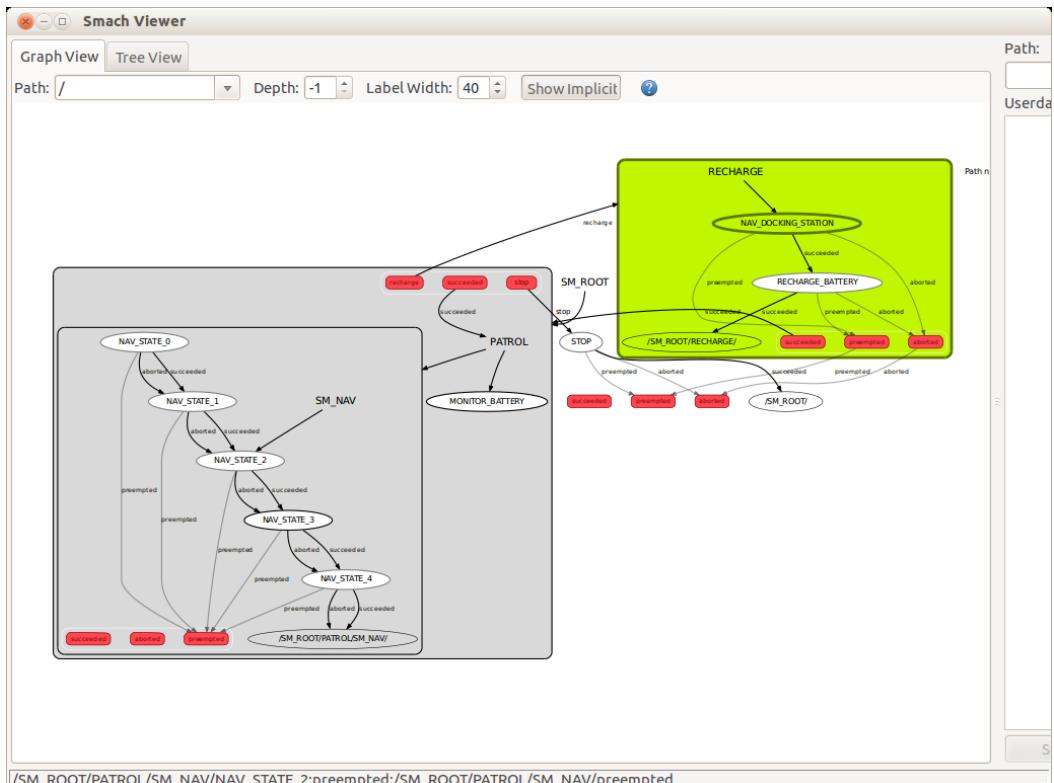
```
$ rosrun rviz rviz -d `rospack find rbx2_tasks`/nav_tasks.rviz
```

Finally, make sure you can see the `RViz` window in the foreground, then run the `patrol_smach_concurrence.py` script:

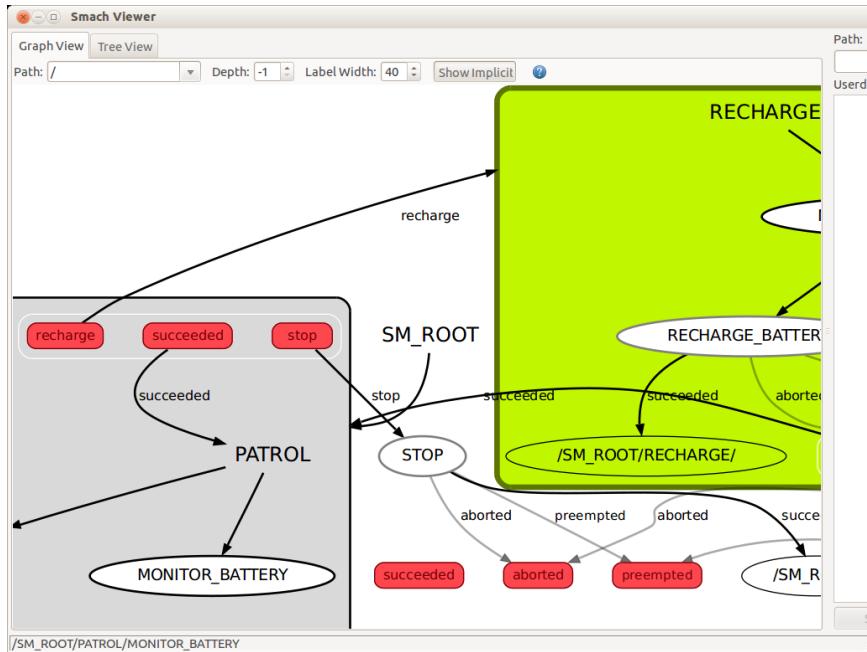
```
$ rosrun rbx2_tasks patrol_smach_concurrence.py
```

You should see the robot move around the square two times and then stop. Whenever the battery falls below threshold (set by default to 50 in the `task_setup.py` file), the robot will interrupt its patrol and head to the docking station for a recharge. Once recharged, it will continue its patrol where it left off.

While the script is running, you can also view the state machine in the SMACH viewer. The image should look something like the following:



(If you don't see this image in the SMACH viewer, shut it down then bring it back up again.) The image on your screen should be much clearer than we can reproduce here. The larger gray box on the left represents the concurrence container that contains the navigation state machine `SM_NAV` inside the inner box and the monitor state `MONITOR_BATTERY` that subscribes to the battery topic. The smaller green box on the right represents the `RECHARGE` state machine and contains the `NAV_DOCKING_STATION` and `RECHARGE_BATTERY` states. The transitions between the two state machines can be seen more clearly in the magnified image below:



The arcing arrow from the red `recharge` outcome on the left to the green `RECHARGE` box on the right indicates the transition from the concurrence container to the `RECHARGE` state machine when the concurrence produces the '`recharge`' outcome.

Let us now examine the code that makes this work. The following is a summary of the overall steps:

- create a state machine called `sm_nav` that takes care of moving the robot from one waypoint to the next
- create a second state machine called `sm_recharge` that moves the robot to the docking station and performs a recharge
- create a third state machine called `sm_patrol` defined as a Concurrence container that pairs the `sm_nav` state machine with a `MonitorState` state that subscribes to the battery topic and can preempt the `sm_nav` machine while firing up the `sm_recharge` state machine
- finally, create a fourth state machine called `sm_top` that includes the `sm_patrol` and `sm_recharge` machines as well as a `STOP` state that allows us to terminate the entire patrol once we have completed the specified number of loops.

The tree diagram of our state machine looks like this:

- sm_top
 - sm_patrol
 - monitor_battery
 - sm_nav
 - sm_recharge

The full script can be found in the file `patrol_smach_concurrence.py` in the directory `rbx2_tasks/nodes`.

Link to source: [patrol_smach_concurrence.py](#)

The top half of the script is the same as the `patrol_smach.py` script without the battery check. So let's look now at the lines that bring in the docking station and the battery check.

```
# Create a MoveBaseAction state for the docking station
nav_goal = MoveBaseGoal()
nav_goal.target_pose.header.frame_id = 'map'
nav_goal.target_pose.pose = self.docking_station_pose
nav_docking_station = SimpleActionState('move_base', MoveBaseAction,
    goal=nav_goal, result_cb=self.move_base_result_cb,
    exec_timeout = rospy.Duration(20.0),
    server_wait_timeout=rospy.Duration(10.0))
```

As with the four navigation waypoints, we create a `SimpleActionState` wrapping a `move_base` action that gets the robot to the docking station. The location of the docking station on the map (`self.docking_station_pose`) is set in our setup file `task_setup.py` included at the top of program.

```
self.sm_nav.register_transition_cb(self.nav_transition_cb, cb_args=[])
```

Next we register a callback function on the `sm_nav` state machine that fires on a state transition. The callback function looks like this:

```
def nav_transition_cb(self, userdata, active_states, *cb_args):
    self.last_nav_state = active_states
```

As you can see, the function simply stores the last active state in the variable `self.last_nav_state`. This will allow us to continue the patrol where we left off after the robot completes a recharge.

Next, let's look at the various state machine definitions in the reverse order that they appear in the script. This is because SMACH defines hierarchical state machines in terms of component state machines that are constructed beforehand. However, in terms of understanding the script, we tend to think from top to bottom in the hierarchy.

The top level state machine `sm_top` looks like this:

```
# Create the top level state machine
self.sm_top = StateMachine(outcomes=['succeeded', 'aborted', 'preempted'])

# Add nav_patrol, sm_recharge and Stop() machines to sm_top
with self.sm_top:
    StateMachine.add('PATROL', self.nav_patrol,
                     transitions={'succeeded': 'PATROL', 'recharge': 'RECHARGE',
                                   'stop': 'STOP'})
    StateMachine.add('RECHARGE', self.sm_recharge,
                     transitions={'succeeded': 'PATROL'})
    StateMachine.add('STOP', Stop(), transitions={'succeeded': ''})
```

Here we see that the overall state machine is composed of three component state machines: the `nav_patrol` machine labeled 'PATROL', the `sm_recharge` machine called 'RECHARGE' and a functionally defined state machine labeled 'STOP'.

Furthermore, the transition table for the `nav_patrol` machine indicates that an outcome of 'succeeded' should lead back to the PATROL state while an outcome of 'recharge' should transition to the RECHARGE state and an outcome of 'stop' should fire up the STOP state. The RECHARGE state has only one transition defined which is to return to the PATROL state if the recharge outcome is 'succeeded'. Finally, the STOP state transitions to an empty state thereby terminating the entire state machine.

Let's now break down each of these component states and state machines. The last state labeled STOP is defined earlier in the script and looks like this:

```
class Stop(State):
    def __init__(self):
        State.__init__(self, outcomes=['succeeded', 'aborted', 'preempted'])
        pass

    def execute(self, userdata):
        rospy.loginfo("Shutting down the state machine")
        return 'succeeded'
```

Here we create a SMACH state that simply prints out a message and returns 'succeeded'. Its only function is to give us a state we can transition to when we are finished all patrols.

The RECHARGE state is another state machine called `sm_recharge` that is defined as follows:

```
with self.sm_recharge:
    StateMachine.add('NAV_DOCKING_STATION', nav_docking_station,
                      transitions={'succeeded':'RECHARGE_BATTERY'})
    StateMachine.add('RECHARGE_BATTERY',
                     ServiceState('battery_simulator/set_battery_level', SetBatteryLevel,
                     100, response_cb=self.recharge_cb), transitions={'succeeded':''})
```

The first state gets the robot to the docking station using the `nav_docking_station` SimpleActionState and the second state simulates a recharge by using a SMACH ServiceState to call the `set_battery_level` service with a value of 100 (full charge).

Finally, the PATROL state is defined by the `nav_patrol` state machine that looks like this:

```
self.nav_patrol = Concurrence(outcomes=['succeeded', 'recharge', 'stop'],
                               default_outcome='succeeded',
                               child_termination_cb=self.concurrence_child_termination_cb,
                               outcome_cb=self.concurrence_outcome_cb)

with self.nav_patrol:
    Concurrence.add('SM_NAV', self.sm_nav)
    Concurrence.add('MONITOR_BATTERY', MonitorState("battery_level", Float32,
                                                    self.battery_cb))
```

Here we have defined the `nav_patrol` state machine as a SMACH Concurrence container. In this case, the Concurrence includes the navigation state machine and a SMACH MonitorState that subscribes to the `battery_level` topic. This sets up the `nav_patrol` machine to move the robot around the square while monitoring the battery level.

The MonitorState state takes three arguments: the topic we want to subscribe to, the message type for that topic, and a callback function that gets called whenever a message is received on the subscribed topic. In other words, it wraps a standard ROS subscribe operation. In our case, the callback function, `self.battery_cb` looks like this:

```
def battery_cb(self, userdata, msg):
    if msg.data < self.low_battery_threshold:
        self.recharging = True
        return False
    else:
        self.recharging = False
        return True
```

Here we see that the callback function returns `False` if the battery level falls below threshold or `True` otherwise. We also set a flag to indicate whether or not we are recharging so we won't count recharging against the overall navigation success rate.

The final question is: how does a low battery level condition preempt the current navigation task, start the recharge task, then resume the previous navigation task? You'll notice that the `Concurrence` container defines two callback functions: the `child_termination_cb` that which fires when *either* child state terminates; and the `outcome_cb` callback that fires when *all* child states have terminated. Let's look at these in turn.

```
def concurrence_child_termination_cb(self, outcome_map):
    # If the current navigation task has succeeded, return True
    if outcome_map['SM_NAV'] == 'succeeded':
        return True
    # If the MonitorState state returns False (invalid), store the current nav
    goal and recharge
    if outcome_map['MONITOR_BATTERY'] == 'invalid':
        rospy.loginfo("LOW BATTERY! NEED TO RECHARGE...")
        if self.last_nav_state is not None:
            self.sm_nav.set_initial_state(self.last_nav_state, UserData())
        return True
    else:
        return False
```

The child termination callback fires when either the `sm_nav` machine or the battery `MonitorState` returns so we have to check for either condition. The outcome map is indexed by the name of the state machines that make up the `Concurrence` container which in our case are `SM_NAV` and `MONITOR_BATTERY`. So first we check the status of the `SM_NAV` machine and if it has succeeded (i.e. the robot just made it to a waypoint) we return `True`. Otherwise, we check the status of the battery monitor state. Recall that a `MonitorState` maps `True` into `valid` and `False` into `invalid`. If we see an outcome of `invalid` (the battery has fallen below threshold), we reset the initial state of the `sm_nav` machine to be the last successful navigation goal and return `True`.

The outcome callback is executed when both the `sm_nav` machine and the `MonitorState` have terminated. In our case, the callback function looks like this:

```
def concurrence_outcome_cb(self, outcome_map):
    # If the battery is below threshold, return the 'recharge' outcome
    if outcome_map['MONITOR_BATTERY'] == 'invalid':
        return 'recharge'
    # Otherwise, if the last nav goal succeeded, return 'succeeded' or 'stop'
    elif outcome_map['SM_NAV'] == 'succeeded':
        self.patrol_count += 1
        rospy.loginfo("FINISHED PATROL LOOP: " + str(self.patrol_count))
    # If we have not completed all patrols, start again at the beginning
```

```

if self.n_patrols == -1 or self.patrol_count < self.n_patrols:
    self.sm_nav.set_initial_state(['NAV_STATE_0'], UserData())
    return 'succeeded'
# Otherwise, we are finished patrolling so return 'stop'
else:
    self.sm_nav.set_initial_state(['NAV_STATE_4'], UserData())
    return 'stop'
# Recharge if all else fails
else:
    return 'recharge'

```

As with the child termination callback, we test the outcome map for each of the Concurrence states. In this case we test the MONITOR_BATTERY outcome first and if it is invalid (the battery is low), we return an outcome of 'recharge'. Recall that in our top-level state machine, this outcome will cause a transition to the RECHARGE state which in turn is defined by the sm_recharge state machine that will move the robot to the docking station. Otherwise, we test the outcome of the SM_NAV state and, if it returns 'succeeded', meaning we have arrived at a waypoint, we then test to see if we have finished the designated number of patrols and can stop or keep patrolling.

3.8.10 Comments on the battery checking Patrol Bot

If you are not used to programming state machines, the previous section might have seemed a little cumbersome just to get the robot to navigate a square while checking its battery level. And indeed, for such a simple case, writing a standard declarative script as we did earlier would probably be easier. State machines tend to show their strength when the problem becomes more complex. Nonetheless, state machines are not for everyone and we will look at another approach later on.

3.8.11 Passing user data between states and state machines

The SMACH Wiki includes [a tutorial](#) describing how to pass user data from one state to another or between the state machine as a whole and any given state. For example, suppose we would like the Patrol Bot to pick the next waypoint randomly rather than always moving in the same sequence. This would make the robot less predictable to a potential intruder. One way to do this is to create a state that selects the next waypoint at random and then passes the result to a navigation state that does the actual moving of the robot to the selected location. Before we write a script to do exactly this, let's review the essential concepts from the one tutorial linked to above.

The key to SMACH data passing is the `userdata` object that is essentially a Python dictionary that maps `input_keys` and `output_keys` to states (and state machines) and each other. Any state (or machine) that needs to output the value of a variable must list that variable among its `output_keys`. Likewise, a state that requires a variable as input must include that variable in its `input_keys`. When constructing the state

machine, a remapping argument provides a dictionary mapping input and output keys to and from intermediary keys that allow the data to be passed between states. The tutorial linked to above provides a few detailed examples. We will now show how these concepts can be used to produce a random Patrol Bot.

Our new script is called [random_patrol_smach.py](#) and is located in the `rbx2_tasks/nodes` directory. The core of the state machine is embodied in the following lines:

```
# Initialize the patrol state machine
self.sm_patrol = StateMachine(outcomes=['succeeded', 'aborted', 'preempted'])

# Set the userdata.waypoints variable to the pre-defined waypoints
self.sm_patrol.userdata.waypoints = self.waypoints

# Add the states to the state machine with the appropriate transitions
with self.sm_patrol:
    StateMachine.add('PICK WAYPOINT', PickWaypoint(),
                     transitions={'succeeded': 'NAV WAYPOINT'},
                     remapping={'waypoint_out': 'patrol_waypoint'})

    StateMachine.add('NAV WAYPOINT', Nav2Waypoint(),
                     transitions={'succeeded': 'PICK WAYPOINT',
                                  'aborted': 'PICK WAYPOINT',
                                  'preempted': 'PICK WAYPOINT'},
                     remapping={'waypoint_in': 'patrol_waypoint'})
```

Let's take each line in turn:

```
self.sm_patrol = StateMachine(outcomes=['succeeded', 'aborted', 'preempted'])
```

Here we create the overall patrol state machine as usual with the standard set of outcomes.

```
self.sm_patrol.userdata.waypoints = self.waypoints
```

To make the pre-defined set of waypoints available to the state machine and any states we subsequently add to it, we assign the array to a `userdata` variable. This variable can have any name you like but it makes sense to simply call it `waypoints` in this case.

```
with self.sm_patrol:
    StateMachine.add('PICK WAYPOINT', PickWaypoint(),
                     transitions={'succeeded': 'NAV WAYPOINT'},
                     remapping={'waypoint_out': 'patrol_waypoint'})
```

The first state we add is called `PICK WAYPOINT` and consists of the custom state `PickWaypoint()` that we will describe later. This state will select a waypoint at random and return it as the output variable called `waypoint_out`. To make this variable available to other states, we use a `remapping` dictionary to map it into an intermediary variable called `patrol_waypoint`. This variable can be any name as long as the same name is used with other states that need to use the variable as we shall see next.

```
StateMachine.add('NAV WAYPOINT', Nav2Waypoint(),
    transitions={'succeeded': 'PICK WAYPOINT',
                 'aborted': 'PICK WAYPOINT',
                 'preempted': 'PICK WAYPOINT'},
    remapping={'waypoint_in': 'patrol_waypoint'})
```

Next we add the second state which we call `NAV WAYPOINT` and represents the custom state `Nav2Waypoint()` which will be defined below. As we will see, this state looks for a data variable named `waypoint_in` that tells the state where to navigate the robot. Note that we don't pass this variable to `Nav2Waypoint()` as an argument; instead, we use a `remapping` dictionary for the state that maps the intermediary `patrol_waypoint` variable to the `waypoint_in`.

The last two pieces to examine are the custom states `PickWaypoint()` and `Nav2Waypoint()`. Here is the `PickWaypoint()` code:

```
class PickWaypoint(State):
    def __init__(self):
        State.__init__(self, outcomes=['succeeded'], input_keys=['waypoints'],
                      output_keys=['waypoint_out'])

    def execute(self, userdata):
        waypoint_out = randrange(len(userdata.waypoints))

        userdata.waypoint_out = waypoint_out

        rospy.loginfo("Going to waypoint " + str(waypoint_out))

        return 'succeeded'
```

As you can see, this state has two arguments defining a set of `input_keys` and `output_keys`. For this state, there is only one `input_key` called '`waypoints`' and one `output_key` called '`waypoint_out`'. The `execute()` function automatically gets passed the `userdata` object as an argument. So first we pick a random number from 0 to the length of the `userdata.waypoints` array, then we assign it to the `userdata.waypoint_out` variable.

Finally, let's look at the `Nav2Waypoint()` state:

```

class Nav2Waypoint(State):
    def __init__(self):
        State.__init__(self, outcomes=['succeeded', 'aborted', 'preempted'],
                      input_keys=['waypoints', 'waypoint_in'])

        # Subscribe to the move_base action server
        self.move_base = actionlib.SimpleActionClient("move_base",
                                                      MoveBaseAction)

        # Wait up to 60 seconds for the action server to become available
        self.move_base.wait_for_server(rospy.Duration(60))

        rospy.loginfo("Connected to move_base action server")

        self.goal = MoveBaseGoal()
        self.goal.target_pose.header.frame_id = 'map'

    def execute(self, userdata):
        self.goal.target_pose.pose = userdata.waypoints[userdata.waypoint_in]

        # Send the goal pose to the MoveBaseAction server
        self.move_base.send_goal(self.goal)

        if self.preempt_requested():
            self.service_preempt()
            return 'preempted'

        # Allow 1 minute to get there
        finished_within_time =
        self.move_base.wait_for_result(rospy.Duration(60))

        # If we don't get there in time, abort the goal
        if not finished_within_time:
            self.move_base.cancel_goal()
            rospy.loginfo("Timed out achieving goal")
            return 'aborted'
        else:
            # We made it!
            state = self.move_base.get_state()
            if state == GoalStatus.SUCCEEDED:
                rospy.loginfo("Goal succeeded!")
            return 'succeeded'

```

In this case, we have two `input_keys`, 'waypoints' and 'waypoint_in' and no `output_keys`. In the lines before the `execute()` function, we simply connect to the `move_base` action server as we have done in the past. The `execute()` function sets the goal pose from the `userdata.waypoints` array using the value of `userdata.waypoint_in` as the index into the array. The rest of the function simply sends that goal to the `move_base` action server, waits for the result and returns 'succeeded', 'preempted' or 'aborted' as appropriate.

To try out the script, run the following commands. If you don't already have the fake TurtleBot running, bring it up now:

```
$ rosrun rbx2_tasks fake_turtlebot.launch
```

Similarly, if the SMACH viewer is not running, fire it up with:

```
$ rosrun smach_viewer smach_viewer.py
```

If you don't already have RViz running with the nav_tasks config file, bring it up with:

```
$ rosrun rviz rviz -d `rospack find rbx2_tasks`/nav_tasks.rviz
```

Finally, make sure you can see the RViz window in the foreground, then run the random_patrol_smach.py script:

```
$ rosrun rbx2_tasks random_patrol_smach.py
```

You should see the robot move randomly from waypoint to waypoint indefinitely.

For bonus credit, the reader is encouraged to write a new script that combines the random patrol with the battery check using a SMACH Concurrence container.

3.8.12 Subtasks and hierarchical state machines

As we have seen, SMACH enables us to nest state machines inside each other. This allows us to break down a complex set of goals into primary tasks and subtasks. For example, a house cleaning robot might be programmed to navigate from one room to the next while carrying out a number of subtasks specific to each room. The procedure might go something like this:

- START_LOCATION → LIVING_ROOM
 - VACCUM_FLOOR → DUST_FURNITURE → CLEAN_WINDOWS
- LIVING_ROOM → KITCHEN
 - MOP_FLOOR → DO_DISHES
- KITCHEN → BATHROOM
 - WASH_TUB → MOP_FLOOR
- BATHROOM → HALLWAY

Each subtask could itself have subtasks such as:

- MOP_FLOOR
 - RETRIEVE_MOP → WASH_FLOOR → RINSE_MOP → STORE_MOP

SMACH makes it relatively easy to build and visualize these more complex state machines. The trick is to create a separate state machine for each subtask which then becomes a state in the parent state machine.

Let's program an imaginary house cleaning robot as an example. Suppose that each corner of our navigation square represents a room and we want the robot to perform one or more cleaning tasks in each room before moving on to the next room.

We will let the corners of the square represent the living room, kitchen, bathroom and hallway. In the living room, the robot will vacuum the carpet; in the kitchen, it will mop the floor; and in the bathroom it will both scrub the tub and mop the floor. For illustrative purposes, we will also use the `easygui` utility to create some pop-up messages along the way and allow the user to control when the robot moves to the next room.

Our script can be found in the file `clean_house_smach.py` in the `rbx2_tasks/nodes` subdirectory. Before looking at the code, let's try it out in the simulator.

If the `fake_turtlebot.launch` file is not already running, bring it up now:

```
$ roslaunch rbx2_tasks fake_turtlebot.launch
```

Next, bring up the SMACH viewer with the command:

```
$ rosrun smach_viewer smach_viewer.py
```

If you don't already have RViz running with the `nav_tasks` config file, fire it up now:

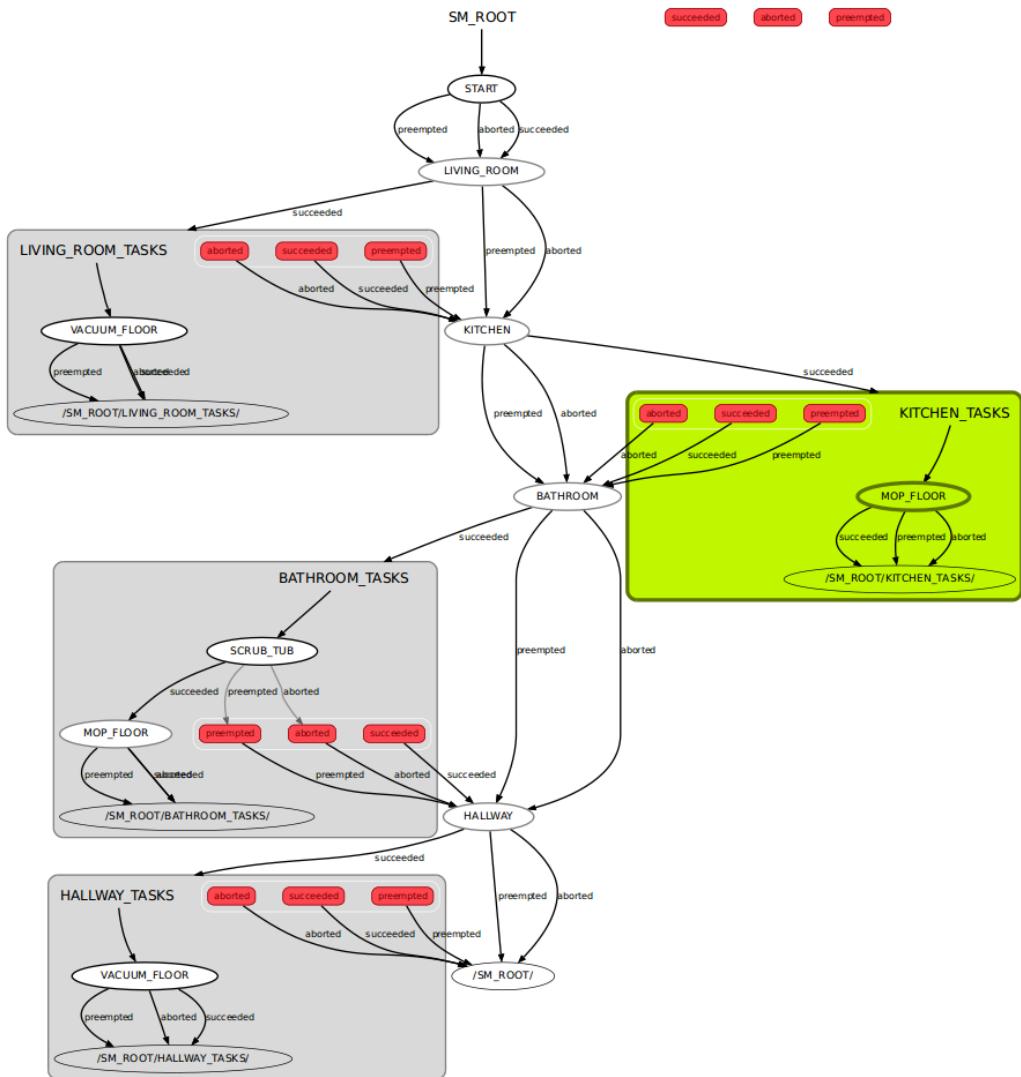
```
$ rosrun rviz rviz -d `rospack find rbx2_tasks`/nav_tasks.rviz
```

Finally, make sure you can see the RViz window in the foreground, then run the `clean_house_smach.py` script:

```
$ rosrun rbx2_tasks clean_house_smach.py
```

You should see the robot move from room to room and perform various cleaning tasks. You can also watch the progress in graphical form in the `smach_viewer` window. As each cleaning task is completed, a pop-up window will appear. Click OK to allow the robot to continue. (The pop-up windows are created by the `easygui` Python module and are used just to illustrate how we can call other programs from within a state machine.)

As the robot moves through the state machine, the image in the `smach_viewer.py` should look something like the following:



The `clean_house_smach.py` script is fairly straightforward so we'll highlight just the key concepts. The most important concept in the script is the construction of a hierarchical state machine. First, we create a state machine for each room like this:

```

# Create a state machine for the kitchen task(s)
sm_living_room = StateMachine(outcomes=['succeeded', 'aborted', 'preempted'])

# Then add the subtask(s)
with sm_living_room:
    StateMachine.add('VACUUM_FLOOR', VacuumFloor('living_room', 5),
transitions={'succeeded': '', 'aborted': '', 'preempted': ''})

```

Here we create a state machine for the living room and then add a state called VACUUM_FLOOR. This state is defined in terms of a custom class VacuumFloor() which we will describe in detail below.

One we have a state machine for each room containing the various cleaning tasks, we put them together into the overall state machine as follows:

```

# Initialize the overall state machine
sm_clean_house =
StateMachine(outcomes=['succeeded', 'aborted', 'preempted'])

# Build the clean house state machine
with sm_clean_house:
    StateMachine.add('START', nav_states['hallway'],
transitions={'succeeded': 'LIVING_ROOM', 'aborted': 'LIVING_ROOM', 'preempted': 'LIVING_ROOM'})

        ''' Add the living room subtask(s) '''
        StateMachine.add('LIVING_ROOM', nav_states['living_room'],
transitions={'succeeded': 'LIVING_ROOM_TASKS', 'aborted': 'KITCHEN', 'preempted': 'KITCHEN'})

        # When the tasks are done, continue on to the kitchen
        StateMachine.add('LIVING_ROOM_TASKS', sm_living_room,
transitions={'succeeded': 'KITCHEN', 'aborted': 'KITCHEN', 'preempted': 'KITCHEN'})

        ''' Add the kitchen subtask(s) '''
        StateMachine.add('KITCHEN', nav_states['kitchen'],
transitions={'succeeded': 'KITCHEN_TASKS', 'aborted': 'BATHROOM', 'preempted': 'BATHROOM'})

        # When the tasks are done, continue on to the bathroom
        StateMachine.add('KITCHEN_TASKS', sm_kitchen,
transitions={'succeeded': 'BATHROOM', 'aborted': 'BATHROOM', 'preempted': 'BATHROOM'})

        ''' Add the bathroom subtask(s) '''
        StateMachine.add('BATHROOM', nav_states['bathroom'],
transitions={'succeeded': 'BATHROOM_TASKS', 'aborted': 'HALLWAY', 'preempted': 'HALLWAY'})

        # When the tasks are done, return to the hallway
        StateMachine.add('BATHROOM_TASKS', sm_bathroom,
transitions={'succeeded': 'HALLWAY', 'aborted': 'HALLWAY', 'preempted': 'HALLWAY'})

    ''' Add the hallway subtask(s) '''

```

```

StateMachine.add('HALLWAY', nav_states['hallway'],
transitions={'succeeded': 'HALLWAY_TASKS', 'aborted': '', 'preempted': ''})

# When the tasks are done, stop
StateMachine.add('HALLWAY_TASKS', sm_hallway,
transitions={'succeeded': '', 'aborted': '', 'preempted': ''})

```

The construction of the `clean_house` state machine reads somewhat like a list of chores:

- Add a state called `START` that navigates state the robot to the hallway, then transition to a state called `LIVING_ROOM`.
- Add a state called `LIVING_ROOM` that navigates the robot to the living room and once there, transitions to a state called `LIVING_ROOM_TASKS`.
- Add a state called `LIVING_ROOM_TASKS` that points to the `sm_living_room` state machine. Recall that the `sm_living_room` state machine contains the `VACUUM_FLOOR` state and when this subtask succeeds, so does the `sm_living_room` state so add a transition to a state called `KITCHEN`.
- Add a state called `KITCHEN` that navigates the robot to the kitchen and once there, transition to a state called `KITCHEN_TASKS`
- etc.

With the overall state machine constructed, we can later add tasks to any of the individual room state machines without having to modify other parts of the code. For example, the bathroom state machine actually has two subtasks:

```

with sm_bathroom:
    StateMachine.add('SCRUB_TUB', ScrubTub('bathroom', 7),
transitions={'succeeded': 'MOP_FLOOR'})
    StateMachine.add('MOP_FLOOR', MopFloor('bathroom', 5),
transitions={'succeeded': '', 'aborted': '', 'preempted': ''})

```

Here the `SCRUB_TUB` state transitions to the `MOP_FLOOR` state but both states are contained within the `sm_bathroom` state machine. So we only need to add the `sm_bathroom` state machine once to the overall `sm_clean_house` machine. In this way, hierarchical state machines allow us to break down large tasks into logical sub-units that can then be nested together.

Let's now turn to the simulated cleaning tasks themselves. Recall that we added the VACUUM_FLOOR state to the sm_living_room state machine using the following line of code:

```
StateMachine.add('VACUUM_FLOOR', VacuumFloor('living_room', 5),
transitions={'succeeded': '', 'aborted': '', 'preempted': ''})
```

Here we have used SMACH's ability to define a state in terms of another class defined elsewhere in the script. As you will recall from running the simulation in the ArbotiX simulator, each "cleaning task" is represented by some scripted motions of the robot that are meant to look like that action. So we need a state that can essentially run arbitrary code to produce this kind of behavior.

SMACH enables us to extend the generic [State class](#) and override the default `execute()` function with whatever code we like. The code below defines the custom state class used in the `clean_house_smach.py` script to represent the `VaccumFloor` state:

```
class VacuumFloor(State):
    def __init__(self, room, timer):
        State.__init__(self, outcomes=['succeeded', 'aborted', 'preempted'])

        self.task = 'vacuum_floor'
        self.room = room
        self.timer = timer
        self.cmd_vel_pub = rospy.Publisher('cmd_vel', Twist)

    def execute(self, userdata):
        rospy.loginfo('Vacuuming the floor in the ' + str(self.room))
        cmd_vel_msg = Twist()
        cmd_vel_msg.linear.x = 0.05
        counter = self.timer
        while counter > 0:
            if self.preempt_requested():
                self.service_preempt()
                return 'preempted'
            self.cmd_vel_pub.publish(cmd_vel_msg)
            cmd_vel_msg.linear.x *= -1
            rospy.loginfo(counter)
            counter -= 1
            rospy.sleep(1)

        self.cmd_vel_pub.publish(Twist())
        message = "Finished vacuuming the " + str(self.room) + "!"
        rospy.loginfo(message)
        easygui.msgbox(message, title="Succeeded")

        update_task_list(self.room, self.task)

        return 'succeeded'
```

The VacuumState state takes a room name and timer as arguments. We then call the `__init__()` function on the generic State object and define two possible outcomes: succeeded and preempted. The generic State object assumes the callback function is called `execute()` which in turns receives the standard argument called `userdata` that we will explore in the next section. Otherwise, we are free to run nearly any code we want here as long as we return one of the outcomes listed earlier.

In the example above, we use a `cmd_vel` publisher to move the robot back and forth while counting down the timer. Note how we also check for a preempt request using the `preempt_requested()` function which is inherited from the `State` object. If a preempt request is received, the state stops what it is doing and returns the outcome preempted.

If the `execute()` function is allowed to proceed all the way to the end, the global function `update_task_list()` is used to check this task off the list of chores and an outcome of succeeded is returned. We also use the `easygui` module to display a pop-up message that has to be clicked before the robot will continue. Simply comment out this line if you want the robot to perform the task without interruption.

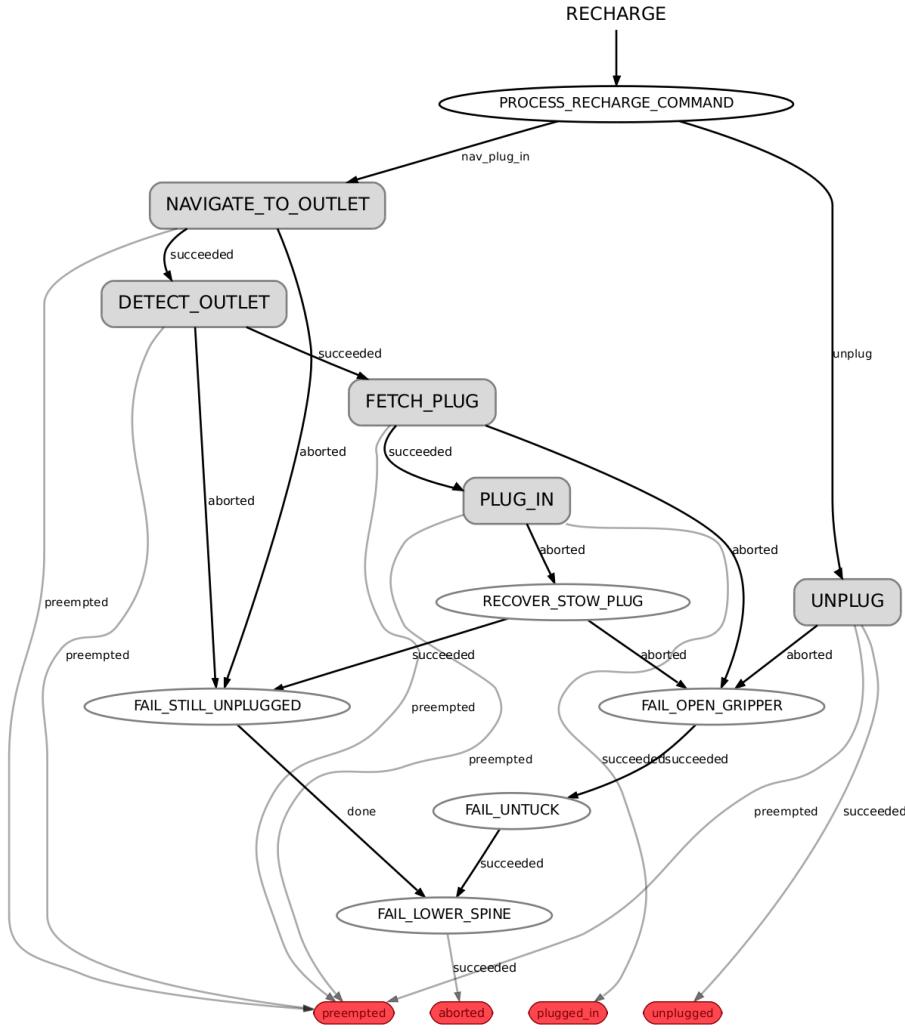
The other "cleaning" tasks such as `Mop` and `Scrub` are defined in a similar manner in the `clean_house_smach.py` script.

3.8.13 Adding the battery check to the house cleaning robot

Using the `patrol_smach_concurrence.py` script as a guide, it is left to the reader to modify the `clean_house_smach.py` script using a Concurrence container so that the robot also checks its battery level and recharges when necessary.

3.8.14 Drawbacks of state machines

If you are not already an experienced state machine programmer, the process of putting one together might appear a little tedious. Large state machines can become somewhat difficult to construct and their state diagrams can sometimes appear difficult to follow such as the one below for [plugging in the PR2](#):



However, some robot tasks are just inherently complex and programming such tasks is never going to be easy. In the next section, we will look at an alternative approach that might seem easier to work with for those without a background in state machines.

3.9 Behavior Trees

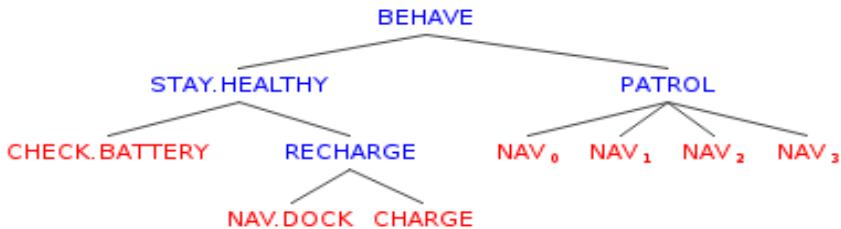
We have seen how to program a complex set of robot behaviors using state machines and SMACH. An alternate approach, called *behavior trees*, has been gaining popularity

among programmers of computer games, and more recently, in robotics as well (e.g. [BART](#)). Perhaps not surprisingly, programming an animated character in a game is not unlike programming a robot to carry out a series of tasks. Both must handle a set of unpredictable inputs and choose from a variety of behaviors appropriate to the task at hand as well as the current situation. (NOTE: The Wikipedia entry for *behavior trees* refers to a unrelated topic related to graphical modeling used in systems and software engineering.)

3.9.1 Behavior Trees versus Hierarchical State Machines

In the early days of game programming, developers tended to use hierarchical state machines similar to SMACH. The main difficulty with state machines is keeping track of all the transitions between states. Adding a new state or behavior to a character (or robot) requires not only coding the new state itself but also adding the transitions between all related states. Before long, the state diagram of the game begins to look a little like tangled spaghetti which can make it difficult to understand and debug.

Although behavior trees are superficially similar to hierarchical state machines, the modularity of their components and the simplicity of their connections make them especially easy to program and understand even when dealing with complex games or behaviors. Here is a behavior tree representation of our Patrol Bot's behavior:



Behavior trees break down complex tasks by branching them into a set of conditions and subtasks. The tree is always executed from top to bottom and left to right. For a given level of the tree, nodes on the left have higher priority than nodes on the right. In the tree shown above, this means that the `STAY_HEALTHY` branch of the tree will always be run before the `PATROL` branch. Similarly, the `CHECK_BATTERY` task will always be executed before the `RECHARGE` behavior.

Behaviors or conditions higher in the tree are more abstract such as "stay healthy" while those at lower levels are more concrete such as "navigate to the docking station". In fact, only the terminal nodes in each branch of the tree result in actual behavior. Terminal nodes are also known as "leaf nodes" and are colored red in the diagram above. We might also refer to them as "action nodes". The blue nodes are called

"interior nodes" and represent sequences or other composite behaviors as we shall see below.

The most important distinction between behavior trees and hierarchical state machines is that there are never direct links between behaviors on the same level of the tree: any connection between such "sibling" behaviors can only be indirect by virtue of shared links to higher-level behaviors. For example, in the SMACH version of the house cleaning robot we programmed earlier, the tasks "clean tub" and "mop floor" in the bathroom state machine connect directly together via transitions such as "when the tub is clean, start mopping the floor". This kind of lateral connection can never occur in a behavior tree. This means that individual behaviors can be treated as independent *modules* and can be moved around the tree without having to worry about breaking lateral connections with other behaviors.

3.9.2 Key properties of behavior trees

The key features of behavior trees can be summarized as follows:

- Nodes represent tasks and conditions rather than states. The terms "task" and "behavior" will be used interchangeably. As will see later on, checking a condition can be thought of as just another kind of task.
- A task or behavior is generally a piece of code that runs for one or more cycles and produces a result of either `SUCCESS` or `FAILURE`. If a task takes more than one cycle to complete, it will have a status of `RUNNING` before returning its result. A task's current status is always passed up to its parent task in the tree.
- Tasks or behaviors can also represent *composite behaviors* whose status depends on two or more child behaviors. The two most commonly used composite behaviors are *selectors* and *sequences*.
 - A *selector* attempts to execute its first child behavior and if it succeeds, the selector also succeeds. Otherwise, the selector attempts to execute the next child behavior and so on. If all child behaviors fail, the selector also fails. In this way, a selector is like a problem solver: first try one solution and if that fails, try the next solution, and so on.
 - A *sequence* attempts to run *all* of its child tasks one after the other. If any child task fails, then the sequence fails. If the sequence is executed without failure all the way to the last child behavior, then the sequence succeeds.
- When a task has more than one subtask, the subtasks are prioritized based on their *list order*; i.e. from left to right in a standard tree diagram or from top to

bottom if using a bulleted list. This property works as a kind of built-in [subsumption](#) architecture with no additional coding necessary.

- The connections between behaviors are always parent-to-child and never sibling-to-sibling. This follows directly from the structure of a tree and it allows us to move a node—or even a whole subtree—from one part of the tree and attach it somewhere else without changing any other connections. Such alterations can even be done at run time. This kind of *modularity* is one of the most valuable features of behavior trees.
- The execution of a behavior tree always begins at the root node and proceeds in *depth-first* order beginning with the left most branch. When a given node is run, the result (`SUCCESS`, `FAILURE` or `RUNNING`) is passed up to its parent. Only leaf nodes result directly in the production of a behavior or the checking of a condition. Interior nodes are used to direct the flow of processing to their child nodes using selectors and sequences. An important property of behavior trees is that execution begins again at the root of the tree on every "tick" of the clock. This ensures that any change in the status of higher priority behaviors will result in the appropriate change in execution flow even if other nodes are already running.
- Behaviors can be augmented with *decorators* that modify the results of the behavior. For example, we might want to execute a sequence of subtasks but ignore failures so that the sequence continues through to the end even when individual subtasks do not succeed. For this we could use an "ignore failure" decorator that turns a result of `FAILURE` into `SUCCESS` for the task it decorates. We will see an example of this in the next section where we revisit the Patrol Bot scenario.
- Many behavior trees make use of a global "black board" that can store data about the environment as well as the results of earlier behaviors. Individual nodes can read and write to the black board.

3.9.3 Building a behavior tree

Constructing a behavior tree can be done either from the top down or from the bottom up or even some combination of the two. It is often easier to begin with the root node and work our way down, beginning with the more abstract composite behaviors and then adding the more concrete conditions and action tasks.

For the Patrol Bot example, the root node will have two composite child behaviors: `STAY_HEALTHY` and `PATROL`. Using a bulleted list, we can represent our tree so far like this:

- BEHAVE (root)
 - STAY_HEALTHY
 - PATROL

where we have used the label BEHAVE for the root node.

Note that the *priority* of a child behavior is determined by its order in the list since we will always run through the child nodes in list order. So in this case, the STAY_HEALTHY behavior has a higher priority than the PATROL behavior. After all, if we let the battery run down, the robot will not be able to continue its patrol.

Next, let's flesh out these two high level behaviors. The STAY_HEALTHY task will consist of two child tasks: CHECK_BATTERY and RECHARGE. It could also include checking servos for overheating, or watching for excessive current to the drive motors (perhaps the robot is stuck on something). The PATROL task will navigate to the four corners of a square using four child tasks. So the behavior tree now looks like this:

- BEHAVE
 - STAY_HEALTHY
 - CHECK_BATTERY
 - RECHARGE
 - PATROL
 - NAV_0
 - NAV_1
 - NAV_2
 - NAV_3

Once again, list order is important. For example, we want to check the battery before we decide to recharge, and we want to patrol the corners of the square in a particular order.

Finally, the RECHARGE task consists of two child behaviors: navigating to the docking station and charging the robot. So our final behavior tree looks like this:

- BEHAVE
 - STAY_HEALTHY
 - CHECK_BATTERY
 - RECHARGE
 - NAV_DOCK
 - CHARGE
 - PATROL
 - NAV_0
 - NAV_1
 - NAV_2
 - NAV_3

With our basic tree structure in place, all that is left is to describe the relationships between layers in the tree; i.e. between parent tasks and their child tasks. Let's turn to that next.

3.9.4 Selectors and sequences

As we learned earlier, behavior trees use two basic types of composite behaviors: *selectors* and *sequences*. A selector tries each of its subtasks in turn until one succeeds whereas a sequence executes each of its child behaviors until one of them fails. So selectors and sequences essentially complement one another. Surprisingly, these two types of parent-child relationships, together with a few variations, are almost all we need to generate very complex robot behaviors. To see how, let's look at them in more detail.

A **selector** starts by executing the first task among its child nodes and, if the task succeeds, the selector's status is also set to `SUCCESS` and all other child tasks are ignored. However, if the first subtask fails, the selector's status is also set to `FAILURE` and the next subtask in its list is executed. In this way, the selector picks a single behavior from its collection of child behaviors, always moving in list order; i.e. from left to right if referring to a tree diagram or top to bottom if using a bulleted list. We can therefore think of a selector as a simple kind of "problem solver". Beginning with its highest priority sub-behavior, it tries each "solution" until one succeeds or until it runs out of behaviors, in which case it passes a final `FAILURE` result up to its parent which is then left to deal with the situation. (More on that later.)

The `STAY_HEALTHY` behavior in our example is a relatively simple selector. First we run the `CHECK_BATTERY` task and if it returns `SUCCESS` (the battery is OK), we pass this result to the parent task which then also has a status of `SUCCESS` and we are done. If however the `CHECK_BATTERY` task returns `FAILURE` (the battery level is low), we move on to the `RECHARGE` task which attempts to remedy the problem by navigating to the docking station and charging the robot. One could also add a `NOTIFY` behavior so that if the `RECHARGE` task fails, the robot could call for help using text-to-speech or alert a human via email.

A **sequence** also starts with the first behavior in its list of subtasks but this time, if the task succeeds, the next subtask in the list is executed. The process continues until either a subtask fails or we run out of subtasks. If the entire sequence is completed, a status of `SUCCESS` is passed to the parent task. However, if the sequence is cut short because a subtask fails, then `FAILURE` is returned up the tree.

Note that the top level `BEHAVE` behavior for our Patrol Bot should be a sequence: if the `STAY_HEALTHY` sub-task fails (which happens when the `CHECK_BATTERY` tasks fails) then the sequence fails and we don't move on to the `PATROL` task.

At first glance, it seems we would use a sequence for the PATROL task since we want the robot to navigate from one waypoint to the next. However, what happens if the robot is unable to get to one of the waypoints for some reason? Perhaps that location is currently inaccessible due to an obstacle. A sequence would fail at this point since one of its subtasks has failed. This is often the desired result. For example, if we are mixing the ingredients for a cake and half way through we find we are out of eggs, we should probably stop mixing until we do some shopping. However, for a Patrol Bot, we might prefer that the robot continue on to the next waypoint rather than aborting the entire patrol. With a slight modification of the sequence composite behavior, we can produce the desired behavior as we shall see next.

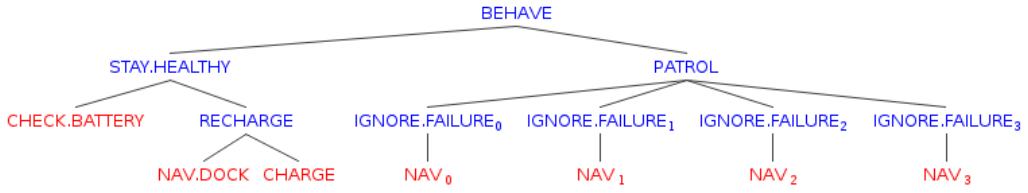
3.9.5 Customizing behaviors using decorators (meta-behaviors)

As we have just seen, selectors and sequences are not always enough to produce the behavior we need from our robot. There are a number of ways we can add greater flexibility to a behavior tree including the use of so-called "decorators". We will use the term "decorator" or "meta-behavior" rather loosely to refer to a behavior or function whose only role is to modify the result of another behavior or task.

For our first example, consider the `IgnoreFailure` meta-behavior which modifies its child behavior(s) by returning `SUCCESS` even if the original result is `FAILURE`. The `IgnoreFailure` meta-behavior is just what we need to make our patrol sequence work the way we want by using it to modify each waypoint `NAV` task. Our behavior tree now looks like this:

- BEHAVE
 - STAY_HEALTHY
 - CHECK_BATTERY
 - RECHARGE
 - NAV.Dock
 - CHARGE
 - PATROL
 - IGNORE_FAILURE
 - NAV_0
 - IGNORE_FAILURE
 - NAV_1
 - IGNORE_FAILURE
 - NAV_2
 - IGNORE_FAILURE
 - NAV_3

Or, in graphical form:



Another way to iterate over a sequence of behaviors while ignoring failures is to simply define a new type of sequence with the ignore failure property built in. The `pi_trees` library includes the `Iterator` composite behavior that does just that.

Another commonly used meta-behavior is the `Loop` decorator which takes a single parameter indicating the number of iterations and executes its child behavior(s) the specified number of times before terminating. We could use a `Loop` decorator around our `PATROL` behavior to limit the number of patrols to a specified number.

There is nothing stopping you from creating any number of meta-behaviors for use in your behavior tree. Some examples include:

- limiting the number of times a behavior can be run
- limiting the frequency at which a behavior can be run by using a timer
- temporarily deactivating a behavior
- a selector that tries the child tasks in random order

3.10 Programming with Behavior Trees and ROS

Since a ready-made behavior tree library for ROS was not available at the time of this writing, a new ROS package called `pi_trees` was created for use with this book. In this section and those that follow, we will install the `pi_trees` package and learn how to use it to program our Patrol Bot and house cleaning robot using behavior trees.

3.10.1 Installing the `pi_trees` library

Before running the examples that follow, we need to install the `pi_trees` ROS package using the following commands:

```

$ sudo apt-get install graphviz-dev libgraphviz-dev \
    python-pygraph python-pygraphviz gv
$ cd ~/catkin_ws/src
$ git clone -b indigo-devel https://github.com/pirobot/pi_trees.git
$ cd ~/catkin_ws
$ catkin_make
$ rospack profile

```

3.10.2 Basic components of the `pi_trees` library

Behavior trees are fairly easy to implement in Python and while there are several different approaches one can take, the methods used in the `pi_trees` package lend themselves well to integrating with ROS topics, services and actions. In fact, the `pi_trees` package was modeled after SMACH so that some of the code might already seem familiar.

The core `pi_trees` library is contained in the file `pi_trees_lib.py` in the `pi_trees/pi_trees_lib/src` directory and the ROS classes can be found in the file `pi_trees_ros.py` under the `pi_trees/pi_trees_ros/src` directory. Let's start with `pi_trees_lib.py`.

Link to source: [pi_trees_lib.py](#)

```

class TaskStatus():
    FAILURE = 0
    SUCCESS = 1
    RUNNING = 2

```

First we define the possible task status values using the class `TaskStatus` as a kind of enum. One can include additional status values such as `ERROR` or `UNKNOWN` but these three seem to be sufficient for most applications.

```

class Task(object):
    """ The base Task class """
    def __init__(self, name, children=None, *args, **kwargs):
        self.name = name
        self.status = None

        if children is None:
            children = []

        self.children = children

    def run(self):
        pass

    def reset(self):
        for c in self.children:
            c.reset()

```

```

def add_child(self, c):
    self.children.append(c)

def remove_child(self, c):
    self.children.remove(c)

def prepend_child(self, c):
    self.children.insert(0, c)

def insert_child(self, c, i):
    self.children.insert(i, c)

def get_status(self):
    return self.status

def set_status(self, s):
    self.status = s

def announce(self):
    print("Executing task " + str(self.name))

# These next two functions allow us to use the 'with' syntax
def __enter__(self):
    return self.name

def __exit__(self, exc_type, exc_val, exc_tb):
    if exc_type is not None:
        return False
    return True

```

The base `Task` class defines the core object of the behavior tree. At a minimum it must have a name and a `run()` function that in general will not only perform some behavior but also return its status. The other key functions are `add_child()` and `remove_child()` that enable us to add or remove sub-tasks to composite tasks such as selectors and sequences (described below). You can also use the `prepend_child()` or `insert_child()` functions to add a sub-task with a specific priority relative to the other tasks already in the list.

When creating your own tasks, you will override the `run()` function with code that performs your task's actions. It will then return an appropriate task status depending on the outcome of the action. This will become clear when we look at the Patrol Bot example later on.

The `reset()` function is useful when we want to zero out any counters or other variables internal to a particular task and its children.

```

class Selector(Task):
    """
        Run each subtask in sequence until one succeeds or we run out of tasks.
    """
    def __init__(self, name, *args, **kwargs):

```

```

super(Selector, self).__init__(name, *args, **kwargs)

def run(self):
    for c in self.children:

        c.status = c.run()

        if c.status != TaskStatus.FAILURE:
            return c.status

    return TaskStatus.FAILURE

```

A Selector runs each child task in list order until one succeeds or until it runs out of subtasks. Note that if a child task returns a status of RUNNING, the selector also returns RUNNING until the child either succeeds or fails.

```

class Sequence(Task):
    """
        Run each subtask in sequence until one fails or we run out of tasks.
    """
    def __init__(self, name, *args, **kwargs):
        super(Sequence, self).__init__(name, *args, **kwargs)

    def run(self):
        for c in self.children:

            c.status = c.run()

            if c.status != TaskStatus.SUCCESS:
                return c.status

        return TaskStatus.SUCCESS

```

A Sequence runs each child task in list order until one succeeds or until it runs out of subtasks. Note that if a child task returns a status of RUNNING, the sequence also returns RUNNING until the child either succeeds or fails.

```

class Iterator(Task):
    """
        Iterate through all child tasks ignoring failure.
    """
    def __init__(self, name, *args, **kwargs):
        super(Iterator, self).__init__(name, *args, **kwargs)

    def run(self):
        for c in self.children:

            c.status = c.run()

            if c.status != TaskStatus.SUCCESS and c.status != TaskStatus.FAILURE:
                return c.status

        return TaskStatus.SUCCESS

```

An Iterator behaves like a Sequence but ignores failures.

```
class ParallelOne(Task):
    """
        A parallel task runs each child task at (roughly) the same time.
        The ParallelOne task returns success as soon as any child succeeds.
    """
    def __init__(self, name, *args, **kwargs):
        super(ParallelOne, self).__init__(name, *args, **kwargs)

    def run(self):
        for c in self.children:
            c.status = c.run()

        if c.status == TaskStatus.SUCCESS:
            return TaskStatus.SUCCESS

    return TaskStatus.FAILURE
```

The key difference between the ParallelOne composite task and a Selector is that the ParallelOne task runs *all* of its tasks on every "tick" of the clock unless (or until) one subtask succeeds. A Selector continues running the *first* subtask until that task either succeeds or fails before moving on to the next subtask or returning altogether.

```
class ParallelAll(Task):
    """
        A parallel task runs each child task at (roughly) the same time.
        The ParallelAll task requires all subtasks to succeed for it to succeed.
    """
    def __init__(self, name, *args, **kwargs):
        super(ParallelAll, self).__init__(name, *args, **kwargs)

    def run(self):
        n_success = 0
        n_children = len(self.children)

        for c in self.children:
            c.status = c.run()
            if c.status == TaskStatus.SUCCESS:
                n_success += 1

        if c.status == TaskStatus.FAILURE:
            return TaskStatus.FAILURE

        if n_success == n_children:
            return TaskStatus.SUCCESS
        else:
            return TaskStatus.RUNNING
```

Similar to the ParallelOne task, the ParallelAll task runs each subtask on each tick of the clock but continues until all subtasks succeed or until one of them fails.

```

class Loop(Task):
    """
        Loop over one or more subtasks for the given number of iterations
        Use the value -1 to indicate a continual loop.
    """
    def __init__(self, name, announce=True, *args, **kwargs):
        super(Loop, self).__init__(name, *args, **kwargs)

        self.iterations = kwargs['iterations']
        self.announce = announce
        self.loop_count = 0
        self.name = name
        print("Loop iterations: " + str(self.iterations))

    def run(self):
        while True:
            if self.iterations != -1 and self.loop_count >= self.iterations:
                return TaskStatus.SUCCESS

            for c in self.children:
                while True:
                    c.status = c.run()

                    if c.status == TaskStatus.SUCCESS:
                        break

                return c.status

            c.reset()

            self.loop_count += 1

            if self.announce:
                print(self.name + " COMPLETED " + str(self.loop_count) + "
LOOP(S)")

```

The `Loop` task simply executes its child task(s) for the given number of iterations. A value of `-1` for the `iterations` parameters means "loop forever". Note that a `Loop` task is still a task in its own right.

```

class IgnoreFailure(Task):
    """
        Always return either RUNNING or SUCCESS.
    """
    def __init__(self, name, *args, **kwargs):
        super(IgnoreFailure, self).__init__(name, *args, **kwargs)

    def run(self):
        for c in self.children:
            c.status = c.run()

```

```

        if c.status != TaskStatus.RUNNING:
            return TaskStatus.SUCCESS
        else:
            return TaskStatus.RUNNING

    return TaskStatus.SUCCESS

```

The `IgnoreFailure` task simply turns a `FAILURE` into a `SUCCESS` for each of its child behaviors. If the status of a child task is `RUNNING`, the `IgnoreFailure` also takes on a status of `RUNNING`.

The `CallbackTask` turns any function into a task. The function name is passed to the constructor as the `cb` argument along with optional arguments. The only constraint on the callback function is that it must return `0` or `False` to represent a `TaskStatus` of `FAILURE` and `1` or `True` to represent `SUCCESS`. Any other return value is interpreted as `RUNNING`.

```

class CallbackTask(Task):
    """
        Turn any callback function (cb) into a task
    """
    def __init__(self, name, cb=None, cb_args=[], cb_kwargs={}, **kwargs):
        super(CallbackTask, self).__init__(name, cb=None, cb_args=[],
        cb_kwargs={}, **kwargs)

        self.name = name
        self.cb = cb
        self.cb_args = cb_args
        self.cb_kwargs = cb_kwargs

    def run(self):
        status = self.cb(*self.cb_args, **self.cb_kwargs)

        if status == 0 or status == False:
            return TaskStatus.FAILURE

        elif status == 1 or status == True:
            return TaskStatus.SUCCESS

        else:
            return TaskStatus.RUNNING

```

3.10.3 ROS-specific behavior tree classes

You can find the ROS-specific behavior tree classes in the file `pi_trees_ros.py` in the directory `pi_trees/pi_trees_ros/src`. This library contains three key ROS tasks: the `MonitorTask` for monitoring a ROS topic; the `ServiceTask` for connecting to a ROS service; and the `SimpleActionTask` for send goals to a ROS action server

and receiving feedback. We will describe these tasks only briefly here as their use will become clear in the programming examples that follow.

Link to source: [pi_trees_ros.py](#)

Let's begin by looking at the MonitorTask class:

```
class MonitorTask(Task):
    """
        Turn a ROS subscriber into a Task.
    """
    def __init__(self, name, topic, msg_type, msg_cb, wait_for_message=True,
                 timeout=5):
        super(MonitorTask, self).__init__(name)

        self.topic = topic
        self.msg_type = msg_type
        self.timeout = timeout
        self.msg_cb = msg_cb

        rospy.loginfo("Subscribing to topic " + topic)

        if wait_for_message:
            try:
                rospy.wait_for_message(topic, msg_type, timeout=self.timeout)
                rospy.loginfo("Connected.")
            except:
                rospy.loginfo("Timed out waiting for " + topic)

    # Subscribe to the given topic with the given callback function executed
    # via run()
    rospy.Subscriber(self.topic, self.msg_type, self._msg_cb)

    def _msg_cb(self, msg):
        self.set_status(self.msg_cb(msg))

    def run(self):
        return self.status

    def reset(self):
        pass
```

The MonitorTask subscribes to a given ROS topic and executes a given callback function. The callback function is defined by the user and is responsible for returning one of the three allowed task status values: SUCCESS, FAILURE or RUNNING.

```
class ServiceTask(Task):
    """
        Turn a ROS service into a Task.
    """
    def __init__(self, name, service, service_type, request, result_cb=None,
                 wait_for_service=True, timeout=5):
```

```

super(ServiceTask, self).__init__(name)

self.result = None
self.request = request
self.timeout = timeout
self.result_cb = result_cb

rospy.loginfo("Connecting to service " + service)

if wait_for_service:
    rospy.loginfo("Waiting for service")
    rospy.wait_for_service(service, timeout=self.timeout)
    rospy.loginfo("Connected.")

# Create a service proxy
self.service_proxy = rospy.ServiceProxy(service, service_type)

def run(self):
    try:
        result = self.service_proxy(self.request)
        if self.result_cb is not None:
            self.result_cb(result)
        return TaskStatus.SUCCESS
    except:
        rospy.logerr(sys.exc_info())
        return TaskStatus.FAILURE

def reset(self):
    pass

```

The ServiceTask wraps a given ROS service and optionally executes a user-defined callback function. By default, a ServiceTask will simply call the corresponding ROS service and return SUCCESS unless the service call itself fails in which case it returns FAILURE. If the user passes in a callback function, this function may simply execute some arbitrary code or it may also return a task status.

```

class SimpleActionTask(Task):
    """
    Turn a ROS action into a Task.
    """

    def __init__(self, name, action, action_type, goal, rate=5,
                 connect_timeout=10, result_timeout=30, reset_after=False, active_cb=None,
                 done_cb=None, feedback_cb=None):
        super(SimpleActionTask, self).__init__(name)

        self.action = action
        self.goal = goal
        self.tick = 1.0 / rate
        self.rate = rospy.Rate(rate)

        self.result = None
        self.connect_timeout = connect_timeout
        self.result_timeout = result_timeout
        self.reset_after = reset_after

```

```

        if done_cb == None:
            done_cb = self.default_done_cb
        self.done_cb = done_cb

        if active_cb == None:
            active_cb = self.default_active_cb
        self.active_cb = active_cb

        if feedback_cb == None:
            feedback_cb = self.default_feedback_cb
        self.feedback_cb = feedback_cb

        self.action_started = False
        self.action_finished = False
        self.goal_status_reported = False
        self.time_so_far = 0.0

        # Goal state return values
        self.goal_states = ['PENDING', 'ACTIVE', 'PREEMPTED',
                            'SUCCEEDED', 'ABORTED', 'REJECTED',
                            'PREEMPTING', 'RECALLING', 'RECALLED',
                            'LOST']

    rospy.loginfo("Connecting to action " + action)

    # Subscribe to the base action server
    self.action_client = actionlib.SimpleActionClient(action, action_type)

    rospy.loginfo("Waiting for move_base action server...")

    # Wait up to timeout seconds for the action server to become available
    try:

        self.action_client.wait_for_server(rospy.Duration(self.connect_timeout))
    except:
        rospy.loginfo("Timed out connecting to the action server " + action)

    rospy.loginfo("Connected to action server")

    def run(self):
        # Send the goal
        if not self.action_started:
            rospy.loginfo("Sending " + str(self.name) + " goal to action
server...")
            self.action_client.send_goal(self.goal, done_cb=self.done_cb,
active_cb=self.active_cb, feedback_cb=self.feedback_cb)
            self.action_started = True

        """ We cannot use the wait_for_result() method here as it will block
            the entire tree so we break it down in time slices of duration
            1 / rate.
        """
        if not self.action_finished:
            self.time_so_far += self.tick
            self.rate.sleep()
            if self.time_so_far > self.result_timeout:
                self.action_client.cancel_goal()
                rospy.loginfo("Timed out achieving goal")
            return TaskStatus.FAILURE

```

```

        else:
            return TaskStatus.RUNNING
    else:
        # Check the final goal status returned by default_done_cb
        if self.goal_status == GoalStatus.SUCCEEDED:
            self.action_finished = True
            if self.reset_after:
                self.reset()
            return TaskStatus.SUCCESS
        elif self.goal_status == GoalStatus.ABORTED:
            self.action_started = False
            self.action_finished = False
            return TaskStatus.FAILURE
        else:
            self.action_started = False
            self.action_finished = False
            self.goal_status_reported = False
            return TaskStatus.RUNNING

    def default_done_cb(self, status, result):
        # Check the final status
        self.goal_status = status
        self.action_finished = True

        if not self.goal_status_reported:
            rospy.loginfo(str(self.name) + " ended with status " +
str(self.goal_states[status]))
            self.goal_status_reported = True

    def default_active_cb(self):
        pass

    def default_feedback_cb(self, msg):
        pass

    def reset(self):
        self.action_started = False
        self.action_finished = False
        self.goal_status_reported = False
        self.time_so_far = 0.0

```

The `SimpleActionTask` mimics the `SimpleActionState` defined in SMACH. Its main function is to wrap a ROS simple action client and therefore takes an action name, action type, and a goal as arguments. It can also take arguments specifying user-defined callback functions for the standard `active_cb`, `done_cb` and `feedback_cb` callbacks that are passed to the ROS simple action client. In particular, the `SimpleActionTask` defines default `done_cb` function reports the final status of the action which is then turned into a corresponding task status to be used in the rest of the behavior tree.

We will examine the `SimpleActionTask` more closely in the context of a number of example programs that we turn to next.

3.10.4 A Patrol Bot example using behavior trees

We have already seen how we can use SMACH to program a robot to patrol a series of waypoints while monitoring its battery level and recharging when necessary. Let's now see how we can do the same using the `pi_trees` package.

Our test program is called `patrol_tree.py` and is located in the `rbx2_tasks/nodes` subdirectory. Before looking at the code, let's try it out.

Begin by bringing up the fake TurtleBot, blank map, and fake battery simulator:

```
$ roslaunch rbtasks fake_turtlebot.launch
```

Next, bring up RViz with the `nav_tasks.rviz` config file:

```
$ rosrun rviz rviz -d `rospack find rbtasks`/nav_tasks.rviz
```

Finally, run the `patrol_tree.py` script:

```
$ rosrun rbtasks patrol_tree.py
```

The robot should make two loops around the square, stopping to recharge when necessary, then stop. Let's now look at the code.

Link to source: [patrol_tree.py](#)

```
1 #!/usr/bin/env python
2
3 import rospy
4 from std_msgs.msg import Float32
5 from geometry_msgs.msg import Twist
6 from rbtasks.srv import *
7 from pi_trees_ros.pi_trees_ros import *
8 from rbtasks.task_setup import *
9
10 class Patrol():
11     def __init__(self):
12         rospy.init_node("patrol_tree")
13
14         # Set the shutdown function (stop the robot)
15         rospy.on_shutdown(self.shutdown)
16
17         # Initialize a number of parameters and variables
18         setup_task_environment(self)
19
20         # Create a list to hold the move_base tasks
21         MOVE_BASE_TASKS = list()
22
```

```

23     n_waypoints = len(self.waypoints)
24
25     # Create simple action navigation task for each waypoint
26     for i in range(n_waypoints + 1):
27         goal = MoveBaseGoal()
28         goal.target_pose.header.frame_id = 'map'
29         goal.target_pose.header.stamp = rospy.Time.now()
30         goal.target_pose.pose = self.waypoints[i % n_waypoints]
31
32         move_base_task = SimpleActionTask("MOVE_BASE_TASK_" + str(i),
33 "move_base", MoveBaseAction, goal)
34
35         MOVE_BASE_TASKS.append(move_base_task)
36
37     # Set the docking station pose
38     goal = MoveBaseGoal()
39     goal.target_pose.header.frame_id = 'map'
40     goal.target_pose.header.stamp = rospy.Time.now()
41     goal.target_pose.pose = self.docking_station_pose
42
43     # Assign the docking station pose to a move_base action task
44     NAV_DOCK_TASK = SimpleActionTask("NAV_DOC_TASK", "move_base",
45 MoveBaseAction, goal, reset_after=True)
45
46     # Create the root node
47     BEHAVE = Sequence("BEHAVE")
48
49     # Create the "stay healthy" selector
50     STAY_HEALTHY = Selector("STAY_HEALTHY")
51
52     # Create the patrol loop decorator
53     LOOP_PATROL = Loop("LOOP_PATROL", announce=True,
54 iterations=self.n_patrols)
55
56     # Add the two subtrees to the root node in order of priority
57     BEHAVE.add_child(STAY_HEALTHY)
58     BEHAVE.add_child(LOOP_PATROL)
59
60     # Create the patrol iterator
61     PATROL = Iterator("PATROL")
62
63     # Add the move_base tasks to the patrol task
64     for task in MOVE_BASE_TASKS:
65         PATROL.add_child(task)
66
67     # Add the patrol to the patrol loop
68     LOOP_PATROL.add_child(PATROL)
69
70     # Add the battery check and recharge tasks to the "stay healthy" task
71     with STAY_HEALTHY:
72         # The check battery condition (uses MonitorTask)
73         CHECK_BATTERY = MonitorTask("CHECK_BATTERY", "battery_level",
74 Float32, self.check_battery)
75
76         # The charge robot task (uses ServiceTask)

```

```

74         CHARGE_ROBOT = ServiceTask("CHARGE_ROBOT",
75             "battery_simulator/set_battery_level", SetBatteryLevel, 100,
76             result_cb=self.recharge_cb)
77
78         # Build the recharge sequence using inline construction
79         RECHARGE = Sequence("RECHARGE", [NAV_DOCK_TASK, CHARGE_ROBOT])
80
81         # Add the check battery and recharge tasks to the stay healthy
82         selector
83             STAY_HEALTHY.add_child(CHECK_BATTERY)
84             STAY_HEALTHY.add_child(RECHARGE)
85
86         # Display the tree before beginning execution
87         print "Patrol Behavior Tree"
88         print_tree(BEHAVE)
89
90         # Run the tree
91         while not rospy.is_shutdown():
92             BEHAVE.run()
93             rospy.sleep(0.1)
94
95     def check_battery(self, msg):
96         if msg.data is None:
97             return TaskStatus.RUNNING
98         else:
99             if msg.data < self.low_battery_threshold:
100                 rospy.loginfo("LOW BATTERY - level: " + str(int(msg.data)))
101                 return TaskStatus.FAILURE
102             else:
103                 return TaskStatus.SUCCESS
104
105     def recharge_cb(self, result):
106         rospy.loginfo("BATTERY CHARGED!")
107
108     def shutdown(self):
109         rospy.loginfo("Stopping the robot...")
110         self.move_base.cancel_all_goals()
111         self.cmd_vel_pub.publish(Twist())
112         rospy.sleep(1)
113
114 if __name__ == '__main__':
115     tree = Patrol()

```

Let's take a look at the key lines of the script:

```
7 from pi_trees_ros.pi_trees_ros import *
```

We begin by importing the `pi_trees_ros` library which in turn imports the core `pi_trees` classes from the `pi_trees_lib` library. The first key block of code involves the creating of the navigation tasks shown below:

```

26     for i in range(n_waypoints + 1):
27         goal = MoveBaseGoal()
28         goal.target_pose.header.frame_id = 'map'
29         goal.target_pose.header.stamp = rospy.Time.now()
30         goal.target_pose.pose = self.waypoints[i % n_waypoints]
31
32         move_base_task = SimpleActionTask("MOVE_BASE_TASK_" + str(i),
33                                         "move_base", MoveBaseAction, goal, reset_after=False)
34
35         MOVE_BASE_TASKS.append(move_base_task)
36
37     # Set the docking station pose
38     goal = MoveBaseGoal()
39     goal.target_pose.header.frame_id = 'map'
40     goal.target_pose.header.stamp = rospy.Time.now()
41     goal.target_pose.pose = self.docking_station_pose
42
43     # Assign the docking station pose to a move_base action task
44     NAV_DOCK_TASK = SimpleActionTask("NAV_DOC_TASK", "move_base",
45                                     MoveBaseAction, goal, reset_after=True)

```

Here we see nearly the same procedure as we used with SMACH although now we are using the `SimpleActionTask` from the `pi_trees` library instead of the `SimpleActionState` from the SMACH library.

Note the parameter called `reset_after` in the construction of a `SimpleActionTask`. We set this to `False` for the `move_base` tasks assigned to waypoints but we set it to `True` for the docking `move_base` task for the following reason. Recall that when a behavior or task in a behavior tree succeeds or fails, it retains that status indefinitely unless it is reset. This "memory" property is essential because on every execution cycle, we poll the status of every node in the tree. This enables us to continually check condition nodes whose status might have changed since the last cycle. However, if the robot has just successfully reached a waypoint, we want that status to be retained on the next pass through the tree so that the parent node will advance the sequence to the next waypoint. On the other hand, when it comes to recharging, we need to reset the navigation task once the robot is docked so that it can be executed again the next time the battery runs low.

Once we have the navigation and docking tasks created, we move on to building the rest of the behavior tree. The order in which we create the nodes in the script is somewhat flexible since it is the parent-child relations that determine the actual structure of the tree. If we start at the root of the tree, our first behavior nodes would look like this:

```

46     BEHAVE = Sequence("BEHAVE")
47
48     # Create the "stay healthy" selector
49     STAY_HEALTHY = Selector("STAY_HEALTHY")
50

```

```

51      # Create the patrol loop decorator
52      LOOP_PATROL = Loop("LOOP_PATROL", iterations=self.n_patrols)
53
54      # Build the full tree from the two subtrees
55      BEHAVE.add_child(STAY_HEALTHY)
56      BEHAVE.add_child(LOOP_PATROL)

```

The root behavior is a Sequence labeled BEHAVE that will have two child branches; one that starts with the Selector labeled STAY_HEALTHY and a second branch labeled LOOP_PATROL that uses the Loop decorator to loop over the patrol task. We then add the two child branches to the root node in the order that defines their priority. In this case, the STAY_HEALTHY branch has higher priority than LOOP_PATROL.

```

58      # Create the patrol iterator
59      PATROL = Iterator("PATROL")
60
61      # Add the move_base tasks to the patrol task
62      for task in MOVE_BASE_TASKS:
63          PATROL.add_child(task)
64
65      # Add the patrol to the patrol loop
66      LOOP_PATROL.add_child(PATROL)

```

Next we take care of the rest of the patrol nodes. The patrol sequence itself is constructed as an Iterator called PATROL. We then add each move_base task to the iterator. Finally, we add the entire patrol to the LOOP_PATROL task.

```

68      # Add the battery check and recharge tasks to the "stay healthy" task
69      with STAY_HEALTHY:
70          # The check battery condition (uses MonitorTask)
71          CHECK_BATTERY = MonitorTask("CHECK_BATTERY", "battery_level",
Float32, self.check_battery)
72
73          # The charge robot task (uses ServiceTask)
74          CHARGE_ROBOT = ServiceTask("CHARGE_ROBOT",
"battery_simulator/set_battery_level", SetBatteryLevel, 100,
result_cb=self.recharge_cb)
75
76          # Build the recharge sequence using inline construction
77          RECHARGE = Sequence("RECHARGE", [NAV_DOCK_TASK, CHARGE_ROBOT])

```

Here we flesh out the STAY_HEALTHY branch of the tree. First we define the CHECK_BATTERY task as a MonitorTask on the ROS topic battery_level using the callback function self.check_battery (described below). Next we define the CHARGE_ROBOT behavior as a ServiceTask that connects to the ROS service battery_simulator/set_battery_level and sends a value of 100 to recharge the fake battery.

We then construct the RECHARGE task as a Sequence whose child tasks are NAV.Dock_Task and CHARGE_ROBOT. Note how we have used the inline syntax to illustrate how we can add child tasks at the same time that we construct the parent. Equivalently, we could have used the three lines:

```
RECHARGE = Sequence("RECHARGE")
RECHARGE.add_child(NAV.Dock_Task)
RECHARGE.add_child(CHARGE_ROBOT)
```

You can use whichever syntax you prefer.

```
80     STAY_HEALTHY.add_child(CHECK_BATTERY)
81     STAY_HEALTHY.add_child(RECHARGE)
```

We complete the STAY_HEALTHY branch of the tree by adding the CHECK_BATTERY and RECHARGE tasks . Note again that the order is important since we want to check the battery first to see if we need to recharge.

```
83     # Display the tree before beginning execution
84     print "Patrol Behavior Tree"
85     print_tree(BEHAVE)
86
87     # Run the tree
88     while not rospy.is_shutdown():
89         BEHAVE.run()
90         rospy.sleep(0.1)
```

Before starting execution, we use the print_tree() function from the pi_trees library to display a representation of the behavior tree on the screen. The tree itself is executed by calling the run() function on the root node. The run() function makes one pass through the nodes of the tree so we need to place it in a loop.

Finally, we have the check_battery() callback:

```
92     def check_battery(self, msg):
93         if msg.data is None:
94             return TaskStatus.RUNNING
95         else:
```

```

96         if msg.data < self.low_battery_threshold:
97             rospy.loginfo("LOW BATTERY - level: " + str(int(msg.data)))
98             return TaskStatus.FAILURE
99         else:
100             return TaskStatus.SUCCESS

```

Recall that this function was assigned to the `CHECK_BATTERY` MonitorTask which monitors the `battery_level` topic. We therefore check the battery level against the `low_battery_threshold` parameter. If the level is below threshold, we return a task status of `FAILURE`. Otherwise we return `SUCCESS`. Because the `CHECK_BATTERY` task is the highest priority task in the `STAY_HEALTHY` selector, if it returns `FAILURE`, then the selector moves on to its next subtask which is the `RECHARGE` task.

The `patrol_tree.py` script illustrates an important property of behavior trees that helps distinguish them from ordinary hierarchical state machines like SMACH. You'll notice that after a recharge, the robot continues its patrol where it left off even though nowhere in the script did we explicitly save the last waypoint reached. Remember that in the SMACH example (`patrol_smach_concurrence.py`), we had to save the last state just before a recharge so that the robot would know where to continue after being charged. Behavior trees inherently store their state by virtue of each node's `status` property. In particular, if the robot is on its way to a waypoint, the navigation task doing the work of moving the robot has a status of `RUNNING`. If the robot is then diverted to the docking station for a recharge, the status of the previously active navigation status is still `RUNNING`. This means that when the robot is fully charged and the `CHECK_BATTERY` task returns `SUCCESS`, control returns automatically to the running navigation node.

3.10.5 A housing cleaning robot using behavior trees

Earlier in the chapter we used SMACH to simulate a house cleaning robot. Let us now do the same using behavior trees. Our new script is called `clean_house_tree.py` and is found in the `rbx2_tasks/nodes` subdirectory. The program is similar to the `patrol_tree.py` script but this time we will add a few tasks that simulate vacuuming, scrubbing and mopping just as we did with the SMACH example. We will also include battery checking and recharge behavior.

Before describing the code, let's try it out. If you don't already have the `fake_turtlebot.launch` file running, bring it up now:

```
$ roslaunch rbx2_tasks fake_turtlebot.launch
```

Recall that this launch file also runs a `move_base` node, the map server with a blank map, and the fake battery node with a default runtime of 60 seconds.

Next, bring up RViz with the `nav_tasks.rviz` config file:

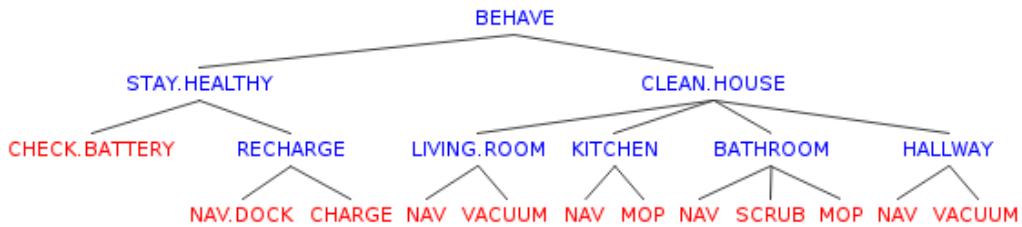
```
$ rosrun rviz rviz -d `rospack find rbx2_tasks`/nav_tasks.rviz
```

Finally, run the `clean_house_tree.py` script:

```
$ rosrun rbx2_tasks clean_house_tree.py
```

The robot should make one circuit of the square, performing cleaning tasks in each room and recharging when necessary.

The overall behavior tree implemented by the `clean_house_tree.py` script looks like this:



In addition to the main tasks shown above, we also require a few condition nodes such as "is the room already clean" and "are we at the desired location?" The need for these condition checks arises from the recharge behavior of the robot. For example, suppose the robot is in the middle of mopping the kitchen floor when its battery level falls below threshold. The robot will navigate out of the kitchen and over to the docking station. Once the robot is recharged, control will return to the last running task—mopping the kitchen floor—but now the robot is no longer in the kitchen. If we don't check for this, the robot will start mopping the docking station! So to get back to the kitchen, we include a task that checks the robot's current location and compares it to where it is supposed to be. If not, the robot navigates back to that location.

A good way to understand the behavior tree we will create is to imagine that you are asked to clean a room yourself. If you were asked to clean the bathroom, you might use a strategy like the following:

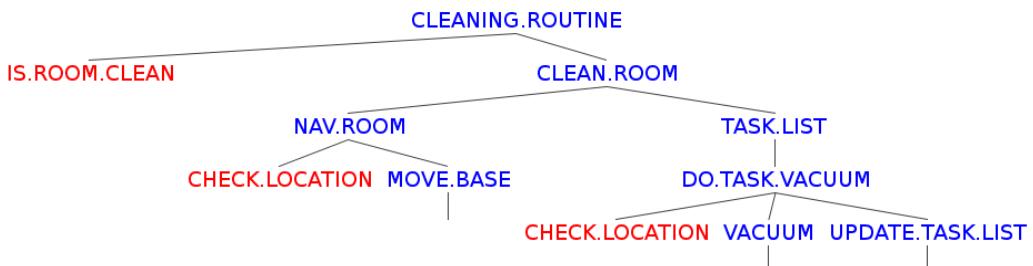
- First find out if the bathroom even needs cleaning. Perhaps your roommate felt energetic and took care of it already. If there is a task checklist somewhere such as on the refrigerator, make sure the bathroom isn't already checked off.

- If the bathroom does need cleaning, you need to be in the bathroom to clean it. If you are already in the bathroom, then you can start cleaning. If not, you have to navigate your way through the house to the bathroom.
- Once you are in the bathroom, check the list of tasks to perform. After each task is completed, put a check mark beside the task on the list.

We can mimic this very same process in a behavior tree. For a given room, the subtree will look something like the following. In parenthesis beside each task, we have indicated the type of task it is: selector, sequence, iterator, condition, or action.

- CLEANING_ROUTINE (selector)
 - IS_ROOM_CLEAN (condition)
 - CLEAN_ROOM (sequence)
 - NAV_ROOM (selector)
 - CHECK_LOCATION (condition)
 - MOVE_BASE (action)
 - TASK_LIST (iterator)
 - DO_TASK_1 (sequence)
 - CHECK_LOCATION (condition)
 - EXECUTE_TASK (action)
 - UPDATE_TASK_LIST (action)
 - DO_TASK_2 (sequence)
 - CHECK_LOCATION (condition)
 - EXECUTE_TASK (action)
 - UPDATE_TASK_LIST (action)
 - ETC

Here's how the tree would look if the only task was to vacuum the living room and we omit the battery checking subtree:



We interpret this tree as follows. The top level node (`CLEANING_ROUTINE`) is a selector so if the condition node `IS_ROOM_CLEAN` returns `SUCCESS`, then we are done. Otherwise we move to selector's next child task, `CLEAN_ROOM`.

The `CLEAN_ROOM` task is a sequence whose first sub-task is `NAV_ROOM` which in turn is a selector. The first sub-task in the `NAV_ROOM` selector is the condition `CHECK_LOCATION`. If this check returns `SUCCESS`, then `NAV_ROOM` also returns `SUCCESS` and the `CLEAN_ROOM` sequence can move to the next behavior in its sequence which is the `TASK_LIST` iterator. If the `CHECK_LOCATION` task returns `FAILURE`, then we execute the `MOVE_BASE` behavior. This continues until `CHECK_LOCATION` returns `SUCCESS`.

Once we are at the target room, the `TASK_LIST` iterator begins. First we check that we are still at the correct location, then we execute each task in the iterator and update the task list.

To make our script more readable, the simulated cleaning tasks can be found in the file `cleaning_tasks_tree.py` under the `rbx2_tasks/src` folder. We then import this file at the top of the `clean_house_tree.py` script. Let's look at the definition of one of these simulated tasks:

```
class VacuumFloor(Task):
    def __init__(self, name, room, timer, *args):
        super(VacuumFloor, self).__init__(self, name, *args)
        self.name = name
        self.room = room
        self.counter = timer
        self.finished = False
        self.cmd_vel_pub = rospy.Publisher('cmd_vel', Twist)
        self.cmd_vel_msg = Twist()
        self.cmd_vel_msg.linear.x = 0.05

    def run(self):
        if self.finished:
            return TaskStatus.SUCCESS
        else:
            rospy.loginfo('Vacuuming the floor in the ' + str(self.room))

            while self.counter > 0:
                self.cmd_vel_pub.publish(self.cmd_vel_msg)
                self.cmd_vel_msg.linear.x *= -1
                rospy.loginfo(self.counter)
                self.counter -= 1
                rospy.sleep(1)
            return TaskStatus.RUNNING

        self.finished = True
        self.cmd_vel_pub.publish(Twist())
        message = "Finished vacuuming the " + str(self.room) + "!"
        rospy.loginfo(message)
```

The `VacuumFloor` class extends the basic `Task` class. Since we want to move the robot back and forth in a simulated vacuuming motion, we create a ROS publisher to send `Twist` message to the `cmd_vel` topic. We then override the `run()` function which creates the desired motion. Since the `run()` function is visited on every pass through the behavior tree, we return a status of `RUNNING` until the motion is complete at which time we return a status of `SUCCESS`.

The `clean_house_tree.py` script is similar to the `patrol_tree.py` program we have already described in detail earlier. Let's therefore focus only on the key differences.

```
class BlackBoard():
    def __init__(self):
        # A list to store rooms and tasks
        self.task_list = list()

        # The robot's current position on the map
        self.robot_position = Point()
```

Recall that some behavior trees use an object called the global "black board" for tracking certain properties of the tree and the world. In Python, the black board can be a simple class with a number of variables to hold the data. At the top of the `clean_house_tree.py` script we define the `BlackBoard()` class shown above with a `list` variable to store the task list and a ROS `Point` variable to track the robot's current coordinates on the map.

```
black_board = BlackBoard()

# Create a task list mapping rooms to tasks
black_board.task_list = OrderedDict([
    ('living_room', [Vacuum(room="living_room", timer=5)]),
    ('kitchen', [Mop(room="kitchen", timer=7)]),
    ('bathroom', [Scrub(room="bathroom", timer=9), Mop(room="bathroom",
timer=5)]),
    ('hallway', [Vacuum(room="hallway", timer=5)])
])
```

Next we create an instance of the `BlackBoard` class and create an ordered list of cleaning tasks using the task definitions from the file `clean_house_tasks_tree.py` in the `src/rbx2_tasks` subdirectory. This task list will be convenient for iterating through all the tasks assigned to the robot. It also means that we can add or remove tasks by simply editing the list here at the top of the script.

The heart of the script involves creating the desired behavior tree from this task list. Here is the relevant block in its entirety.

```

1   for room in black_board.task_list.keys():
2       # Convert the room name to upper case for consistency
3       ROOM = room.upper()
4
5       # Initialize the CLEANING_ROUTINE selector for this room
6       CLEANING_ROUTINE[room] = Selector("CLEANING_ROUTINE_" + ROOM)
7
8       # Initialize the CHECK_ROOM_CLEAN condition
9       CHECK_ROOM_CLEAN[room] = CheckRoomCleaned(room)
10
11      # Add the CHECK_ROOM_CLEAN condition to the CLEANING_ROUTINE selector
12      CLEANING_ROUTINE[room].add_child(CHECK_ROOM_CLEAN[room])
13
14      # Initialize the CLEAN_ROOM sequence for this room
15      CLEAN_ROOM[room] = Sequence("CLEAN_" + ROOM)
16
17      # Initialize the NAV_ROOM selector for this room
18      NAV_ROOM[room] = Selector("NAV_ROOM_" + ROOM)
19
20      # Initialize the CHECK_LOCATION condition for this room
21      CHECK_LOCATION[room] = CheckLocation(room, self.room_locations)
22
23      # Add the CHECK_LOCATION condition to the NAV_ROOM selector
24      NAV_ROOM[room].add_child(CHECK_LOCATION[room])
25
26      # Add the MOVE_BASE task for this room to the NAV_ROOM selector
27      NAV_ROOM[room].add_child(MOVE_BASE[room])
28
29      # Add the NAV_ROOM selector to the CLEAN_ROOM sequence
30      CLEAN_ROOM[room].add_child(NAV_ROOM[room])
31
32      # Initialize the TASK_LIST iterator for this room
33      TASK_LIST[room] = Iterator("TASK_LIST_" + ROOM)
34
35      # Add the tasks assigned to this room
36      for task in black_board.task_list[room]:
37          # Initialize the DO_TASK sequence for this room and task
38          DO_TASK = Sequence("DO_TASK_" + ROOM + " " + task.name)
39
40          # Add a CHECK_LOCATION condition to the DO_TASK sequence
41          DO_TASK.add_child(CHECK_LOCATION[room])
42
43          # Add the task itself to the DO_TASK sequence
44          DO_TASK.add_child(task)
45
46          # Create an UPDATE_TASK_LIST task for this room and task
47          UPDATE_TASK_LIST[room + " " + task.name] = UpdateTaskList(room,
task)
48
49          # Add the UPDATE_TASK_LIST task to the DO_TASK sequence
50          DO_TASK.add_child(UPDATE_TASK_LIST[room + " " + task.name])
51
52          # Add the DO_TASK sequence to the TASK_LIST iterator
53          TASK_LIST[room].add_child(DO_TASK)
54

```

```

55      # Add the room TASK_LIST iterator to the CLEAN_ROOM sequence
56      CLEAN_ROOM[room].add_child(TASK_LIST[room])
57
58      # Add the CLEAN_ROOM sequence to the CLEANING_ROUTINE selector
59      CLEANING_ROUTINE[room].add_child(CLEAN_ROOM[room])
60
61      # Add the CLEANING_ROUTINE for this room to the CLEAN_HOUSE sequence
62      CLEAN_HOUSE.add_child(CLEANING_ROUTINE[room])

```

As you can see, the behavior tree is built by looping over all the tasks in the task list stored on the black board. The inline comments should make clear how we build a subtree for each room and its tasks. We then add each subtree to the overall CLEAN_HOUSE task.

3.10.6 Parallel tasks

Some times we want the robot to work on two or more tasks simultaneously. The `pi_trees` library includes the `Parallel` task type to handle these situations. There are two flavors of `Parallel` task. The `ParallelAll` type returns `SUCCESS` if *all* the simultaneously running tasks succeed. The `ParallelOne` type returns `SUCCESS` as soon as any *one* of the tasks succeeds.

The sample script called `parallel_tree.py` in the `rbx2_tasks/nodes` directory illustrates the `ParallelAll` task type. In this script, the first task prints the message "Take me to your leader" one word at a time. The second task counts to 10. Try out the script now:

```
$ rosrun rbtasks parallel_tree.py
```

You should see the following output:

```

Behavior Tree Structure
--> PRINT_AND_COUNT
    --> PRINT_MESSAGE
    --> COUNT_TO_10
Take
1
me
2
to
3
your
4
leader!
5
6
7
8
9
10

```

Notice how both the message task and the counting task run to completion before the script exits but that the output alternates between the two tasks since they are running in parallel.

Let's take a look at the core part of the code:

```

1  class ParallelExample():
2      def __init__(self):
3          # The root node
4          BEHAVE = Sequence("behave")
5
6          # The message to print
7          message = "Take me to your leader!"
8
9          # How high the counting task should count
10         n_count = 10
11
12         # Create a PrintMessage() task as defined later in the script
13         PRINT_MESSAGE = PrintMessage("PRINT_MESSAGE", message)
14
15         # Create a Count() task, also defined later in the script
16         COUNT_TO_10 = Count("COUNT_TO_10", n_count)
17
18         # Initialize the ParallelAll task
19         PARALLEL_DEMO = ParallelAll("PRINT_AND_COUNT")
20
21         # Add the two subtasks to the Parallel task
22         PARALLEL_DEMO.add_child(PRINT_MESSAGE)
23         PARALLEL_DEMO.add_child(COUNT_TO_10)
24
25         # Add the Parallel task to the root task
26         BEHAVE.add_child(PARALLEL_DEMO)

```

```

27     # Display the behavior tree
28     print "Behavior Tree Structure"
29     print_tree(BEHAVE)
30
31     # Initialize the overall status
32     status = None
33
34     # Run the tree
35     while not status == TaskStatus.SUCCESS:
36         status = BEHAVE.run()
37         time.sleep(0.1)
38

```

The construction of the behavior tree shown above should be fairly self explanatory from the inline comments. Note that the `PrintMessage()` and `Count()` tasks are defined later in the script and are fairly straightforward so we will not display them here.

If you modify the `parallel_tree.py` script so that the `ParallelAll` task on line 19 above is replaced with a `ParallelOne` task instead, the output should look like this:

```

Behavior Tree Structure
--> PRINT_AND_COUNT
    --> PRINT_MESSAGE
    --> COUNT_TO_10
Take
1
me
2
to
3
your
4
leader!

```

Now the script exits as soon as one of the tasks finishes. In this case, the `PRINT_MESSAGE` task completes before the `COUNT_TO_10` task so we do not see the numbers 5-10.

3.10.7 Adding and removing tasks

One of the key features of behavior trees is the ability to add or remove behaviors in a modular fashion. For example, suppose we want to add a "dish washing" task to the house cleaning robot when it is in the kitchen. All we need to do is define this new task and add it to our task list at the top of our script and we are done. There is no need to worry about how this new task interacts with other tasks in the behavior tree. It simply becomes another child task in the `TASK_LIST` iterator for the kitchen.

Conversely, suppose your robot does not have a docking station and you want to eliminate the recharging task from the behavior tree but you still want to be notified when the battery is low. Then we can simply comment out the line that adds the RECHARGE behavior to the STAY_HEALTHY task, then add a new task that sends us an email or cries for help.

The addition or removal of nodes or even whole branches of the behavior tree can even be done dynamically at run time. The sample script `add_remove_tree.py` demonstrates the concept. The script is identical to the `parallel_tree.py` script described in the previous section except that at the end of alternate cycles through the tree, we remove the counting task so that only the words are displayed. On the next cycle, we add back the counting task. Here is the key block of code that does the work:

```
remove = True

while True:
    status = BEHAVE.run()
    time.sleep(0.1)

    if status == TaskStatus.SUCCESS:
        BEHAVE.reset()

        if remove:
            PARALLEL_DEMO.remove_child(COUNT_WORDS)
        else:
            PARALLEL_DEMO.add_child(COUNT_WORDS)

    remove = not remove
```

Try out the script with the command:

```
$ rosrun rbx2_tasks add_remove_tree.py
```

The output should start out the same as the `parallel_tree.py` script except that on successive loops through the script, the counting task will be omitted then reappear, and so on.

It's not hard to imagine a behavior tree where nodes or branches are added or pruned based a robot's experience to better adapt to the conditions on hand. For example, one could add or remove entire branches of the tree depending on which room the robot is in or switch its behavior from cleaning to patrolling by simply snipping one branch of the tree and adding the other.

4. CREATING A URDF MODEL FOR YOUR ROBOT

If you have built your own robot and you want to use it with ROS, you will need to create a URDF model that accurately reflects the robot's dimensions as well as the placement of any cameras, servos, laser scanners or other sensors. The model is used by the ROS `robot_state_publisher` to publish the `tf` transform tree for the robot. The transform tree is then used by other ROS components such as the navigation stack, the `openni_camera` package, and `MoveIt!` (to name a few) to accurately track the relative positions and orientations of the robot's parts relative to each other and to the world.

The best place to start when learning about URDF models for the first time is with the [URDF tutorials](#) on the ROS Wiki. Once you understand the basic concepts and syntax, it sometimes helps to have a few templates that you can customize for your particular robot. This chapter describes a number of URDF/Xacro models that are included in the `rbx2_description` package and explains how you can modify them for your robot.

URDF models can use both mesh objects (STL or Collada) or simple box and cylinder components. We will cover both cases in the sections that follow. Our general strategy will be as follows:

- Create individual URDF/Xacro files for the main components including a base with wheels, a torso, camera, laser scanner, pan-and-tilt head, and arm(s). All dimensions and offsets will be stored as properties at the top of each file so that modifications can be made easily at any time.
- Create separate files for materials (e.g. colors) and hardware components (e.g. Dynamixel servos and brackets).
- Construct our final URDF model by including the relevant files created earlier and attaching the various components in the right places for our particular robot.
- View the model in `RViz` to make sure everything looks OK.
- Test the operation of the model using the ArbotiX simulator.

The modular approach taken here makes it very easy to add or remove components like an arm or to make changes by simply modifying parameters that specify dimensions like torso height or offsets like the placement of the wheels. Most of the files in this chapter started off as copies of Michael Ferguson's [Maxwell](#) robot.

4.1 Start with the Base and Wheels

All robot models in ROS need to start with the base and the simplest robots are nothing more than a base. If you are using a mesh model (STL or Collada), the wheels can be a fixed part of the model or separate meshes. There is no real need to make the wheels rotate around a joint but some ROS packages (like MoveIt!) do assume that the wheels are separate components attached to the base at joints (which can be fixed or continuous).

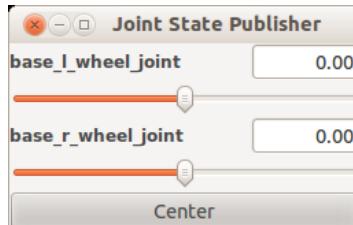
If your robot's torso is fixed to the base and has no moving parts, it can also be included in the same mesh file as the base. However, in the examples that follow, we will treat it as a separate component. This way you can modify the torso at a later time; for example, you might want to add a linear actuator to make it move up and down.

Before we look at the URDF files, let's make sure we can view them in `RViz`. We will use this procedure throughout the chapter for verifying the appearance of our model.

First run the file `box_robot_base_only.launch` in the `rbx2_description` package:

```
$ roslaunch rbx2_description box_robot_base_only.launch
```

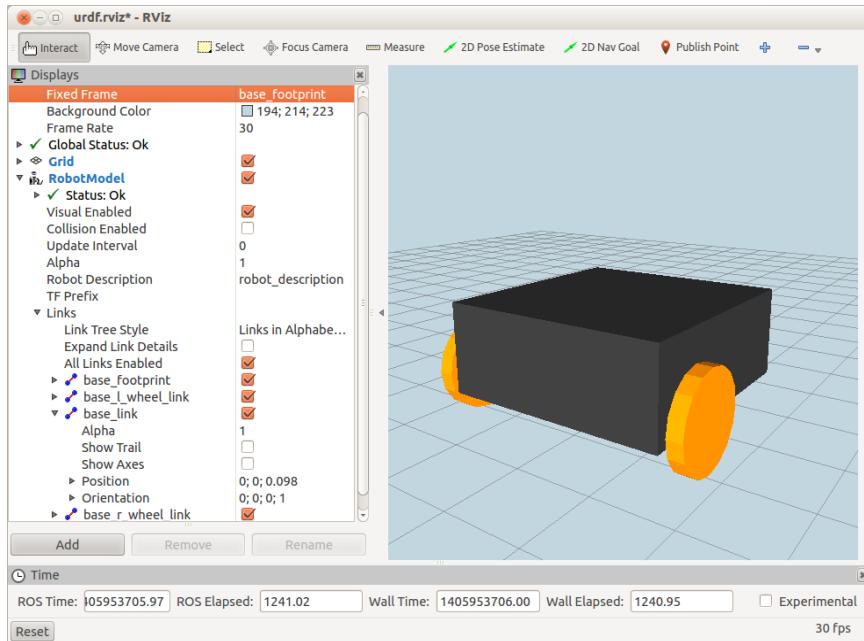
You should see a small window appear called "Joint State Publisher" with two slider controls, one for each drive wheel:



Next, bring up `RViz` with the included `urdf.rviz` config file:

```
$ rosrun rviz rviz -d `rospack find rbx2_description`/urdf.rviz
```

The view in `RViz` should look something like this:



Note how the **Fixed Frame** is set to `/base_footprint`. Keep this in mind as we look at the URDF definition of the base later on.

NOTE: As we load different URDF models throughout this chapter, there is (usually) no need to restart `RViz` to see the changes. Instead, simply toggle the checkbox beside the **RobotModel** display. (By "toggle the checkbox" we mean un-check the box then check it again.) This should refresh the model to the latest version loaded on the parameter server. If a model is already loaded, changes in joint definitions usually get detected by `RViz` automatically whereas modifications to link parameters require the checkbox to be toggled. Occasionally, especially when working with meshes, `RViz` might get stuck on a previously loaded model or display something in the wrong color. On these occasions, restarting `RViz` should clear the problem.

4.1.1 *The robot_state_publisher and joint_state_publisher nodes*

Let's take a look at the launch file we used above, [`box_robot_base_only.launch`](#):

```
<launch>
    <!-- Load the URDF/Xacro model of our robot -->
    <arg name="urdf_file" default="$(find xacro)/xacro.py '$(find
rbx2_description)/urdf/box_robot/box_robot_base_only.urdf.xacro'" />

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

    <!-- Publish the robot state -->
```

```

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

<!-- Provide simulated control of the robot joint angles -->
<node name="joint_state_publisher" pkg="joint_state_publisher"
type="joint_state_publisher">
    <param name="use_gui" value="True" />
    <param name="rate" value="20.0"/>
</node>

</launch>

```

The `<arg>` line near the top points to our robot's URDF/Xacro file and the `<param>` line loads that file onto the ROS parameter server as the parameter named `robot_description`. In fact, you can view the XML of the loaded model using the command:

```
$ rosparam get /robot_description
```

though there is generally little need to do this except perhaps during debugging to verify that the parameter has been set.

The next few lines in the launch file bring up the `robot_state_publisher` node. This node reads the geometry defined in the robot model and publishes a set of transforms that make up the robot's `tf` tree. Without the `tf` tree, very little else could be done with the robot including navigation, 3D vision, or controlling an arm.

The next set of the lines launches the `joint_state_publisher` node. This optional node is used mostly when testing a URDF model in `RViz` as we are doing now. Running the node with the `use_gui` parameter set to `True` brings up the slider control window we saw earlier for setting the positions of any movable joints defined in the robot model. At the moment we only have the two wheel joints but when we add a multi-jointed arm later in the chapter, we will be able to set its configuration as well.

NOTE: When the robot's joints are under the control of an actual hardware driver, the `joint_state_publisher` node is no longer used and in fact, it would conflict with the real joint positions being published by the driver.

4.1.2 The base URDF/Xacro file

Let's now take a look at the URDF/Xacro file that was used above to load the box model of the robot base. The file is called [base.urdf.xacro](#) and it is located in the

`rbx2_description/urdf/box_robot` directory. We will review the file in sections beginning at the top:

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

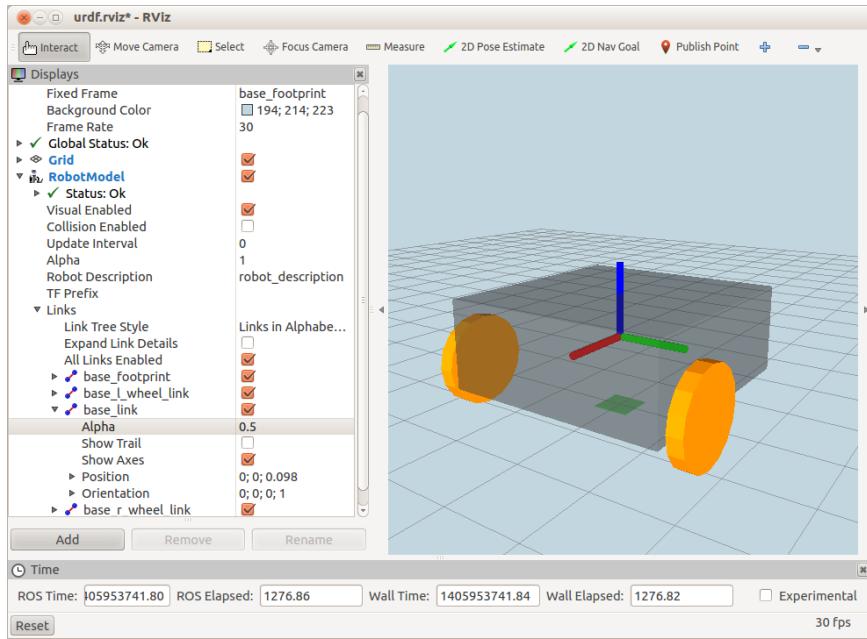
All URDF/Xacro files will begin with these two opening tags. Everything after the `<robot>` tag will define our component and we will close the whole file with an ending `</robot>` tag.

```
4 <!-- Define a number of dimensions using properties -->
5   <property name="base_size_x" value="0.30" />
6   <property name="base_size_y" value="0.30" />
7   <property name="base_size_z" value="0.12" />
8   <property name="wheel_length" value="0.02032" />
9   <property name="wheel_radius" value="0.06191" />
10  <property name="wheel_offset_x" value="0.09" />
11  <property name="wheel_offset_y" value="0.17" />
12  <property name="wheel_offset_z" value="-0.038" />
13
14  <property name="PI" value="3.1415" />
```

The properties section allows us to assign all dimensions and offsets to variables that can then be used throughout the rest of the file. If changes are made to the robot at a later time, simply tweak these values accordingly. Keep in mind the following points when setting property values:

- linear dimensions are specified in meters
- angular values are given in radians
- when specifying `xyz` parameters, the coordinate axes are aligned with `x` pointing in the forward direction of the robot, `y` pointing to its left and `z` pointing upward. For example, the property `wheel_offset_y` above indicates the distance that each wheel is mounted to the left or right of the robot's center line..
- when assigning rotation parameters `rpy` (roll, pitch, yaw), the roll parameter (`r`) rotates around the `x` axis, pitch (`p`) is around the `y` axis and yaw (`y`) rotates about the `z` axis.

To see the axes attached to the base in RViz, set the **Alpha** value of the `base_link` to something like 0.5 like the following:



Here we have set the **Alpha** value to **0 . 5** for the **base_link** and checked the box beside **Show Axes**. RViz always uses the same color code for the frame axes attached to a link: red for the **x-axis**, green for **y** and blue for **z**.

Now back to our discussion of the `base.urdf.xacro` file. Here is the block for a wheel:

```

16  <!-- define a wheel -->
17  <macro name="wheel" params="suffix parent reflect color">
18      <joint name="${parent}_${suffix}_wheel_joint" type="continuous">
19          <axis xyz="0 0 1" />
20          <limit effort="100" velocity="100"/>
21          <safety_controller k_velocity="10" />
22          <origin xyz="${wheel_offset_x} ${reflect*wheel_offset_y} ${wheel_offset_z}" rpy="${reflect*PI/2} 0 0" />
23          <parent link="${parent}_link"/>
24          <child link="${parent}_${suffix}_wheel_link"/>
25      </joint>
26      <link name="${parent}_${suffix}_wheel_link">
27          <visual>
28              <origin xyz="0 0 0" rpy="0 0 0" />
29              <geometry>
30                  <cylinder radius="${wheel_radius}" length="${wheel_length}" />
31              </geometry>
32              <material name="${color}" />
33          </visual>
34      </link>
```

```
35    </macro>
```

A wheel is defined as a macro so that we can use it for each drive wheel without having to repeat the XML. We won't describe the syntax in detail as it is already covered in the [URDF tutorials](#). However, note the use of the `reflect` parameter which takes on the values 1 or -1 for the left and right sides respectively and allows us to flip both the wheel and the sign of the `y` offset.

```
37  <!-- The base xacro macro -->
38  <macro name="base" params="name color">
39      <link name="${name}_link">
40          <visual>
41              <origin xyz="0 0 0" rpy="0 0 0" />
42              <geometry>
43                  <box size="${base_size_x} ${base_size_y} ${base_size_z}" />
44              </geometry>
45              <material name="${color}" />
46          </visual>
47          <collision>
48              <origin xyz="0 0 0" rpy="0 0 0" />
49              <geometry>
50                  <box size="${base_size_x} ${wheel_offset_y*2 + wheel_length} ${base_size_z}" />
51              </geometry>
52          </collision>
53      </link>
54  </macro>
```

Next we define the macro for the base itself. In this case we are using a simple box geometry for the `<visual>` component incorporating the parameters defined at the top of the file. For the `<collision>` block, we have defined a wider box to encompass the wheels. This provides a safety margin around the robot to help prevent the wheels getting caught against obstacles. (Note: if your robot's wheels lie inside the perimeter of the base, then you could simply use the same box used for the visual component.)

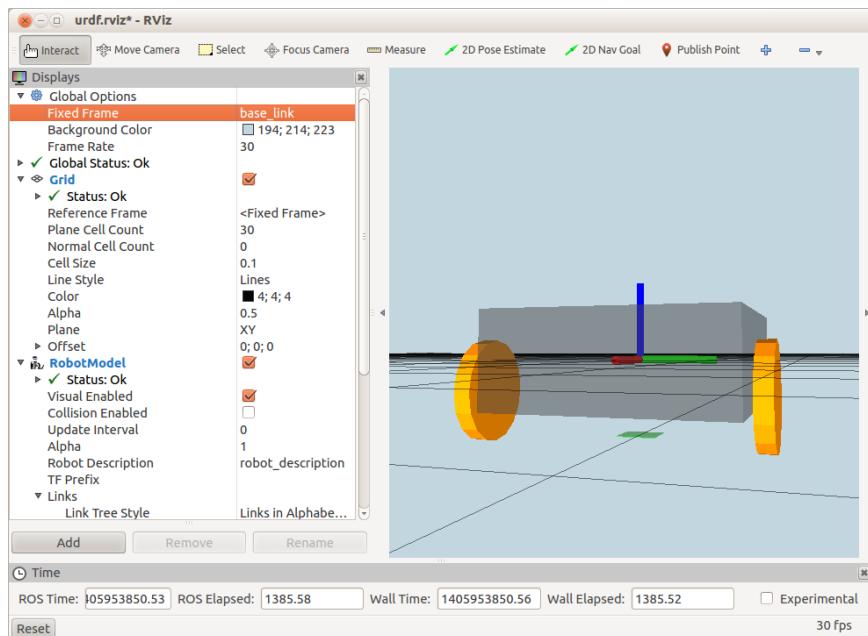
```
56  <link name="base_footprint">
57      <visual>
58          <origin xyz="0 0 0" rpy="0 0 0" />
59          <geometry>
60              <box size="0.05 0.05 0.001" />
61          </geometry>
62          <material name="TransparentGreen" />
63      </visual>
64  </link>
65
66  <joint name="base_joint" type="fixed">
67      <origin xyz="0 0 ${base_size_z/2 - wheel_offset_z}" rpy="0 0 0" />
68      <parent link="base_footprint"/>
```

```

69      <child link="base_link" />
70    </joint>

```

Here we define the `base_footprint` link and a fixed joint that defines its relationship to the `base_link`. As explained in the next section, the role of the `base_footprint` is essentially to define the elevation of the base above the ground. As you can see above, the joint between the footprint and the base raises the base by an amount calculated from the base height and the z-offset of the wheels. To understand why we have to elevate the robot by an amount `base_size_z/2`, change the **Fixed Frame** in RViz to `/base_link`. If you use your mouse to shift the viewpoint so that you are looking at the ground nearly edge-on, the image should look like this:



Note how the origin of a URDF box component is placed at the center of the box; i.e. at a height of `base_size_z/2` meters above the bottom of the robot. Consequently, the fixed joint between the `/base_footprint` frame (which rests on the ground) and the `/base_link` frame must include a translation in the z-direction a distance of `base_size_z/2` meters just to get the bottom of the robot up to the ground plane.

Finally, let's add the two drive wheels to the robot:

```

72    <!-- Add the drive wheels -->
73    <wheel parent="base" suffix="l" reflect="1" color="Orange"/>

```

```
74     <wheel parent="base" suffix="r" reflect="-1" color="Orange"/>
```

Here we call the `<wheel>` macro twice with the `reflect` parameter set first to 1 for the left wheel and -1 for the right to make sure the wheels get mounted on opposite sides of the base. The suffix parameter causes the `<wheel>` macro to give each wheel link a unique name.

4.1.3 Alternatives to using the /base_footprint frame

Not everyone uses the `/base_footprint` frame method for elevating the robot to the proper height above the ground. Another method simply adds the ground clearance to the `<origin>` tag of the `/base_link` frame in the URDF model like this:

```
<parameter name="ground_clearance" value="0.025" />

<!-- The base xacro macro -->
<macro name="base" params="name color">
  <link name="${name}_link">
    <visual>
      <origin xyz="0 0 ${ground_clearance}" rpy="0 0 0" />
      <geometry>
        <box size="${base_size_x} ${base_size_y} ${base_size_z}" />
      </geometry>
      <material name="${color}" />
    </visual>
    <collision>
      <origin xyz="0 0 ${ground_clearance}" rpy="0 0 0" />
      <geometry>
        <box size="${base_size_x} ${wheel_offset_y*2 + wheel_length} ${base_size_z}" />
      </geometry>
    </collision>
  </link>
</macro>
```

The only problem with this approach is that the `ground_clearance` variable then has to be added to the origin of **all** the components in your model, including wheels, torso, camera, arms, etc. If you do use this approach, then remember that other ROS packages like the Navigation stack will require using the `/base_link` frame rather than `/base_footprint` in a number of configuration files.

4.1.4 Adding the base to the robot model

So far we have only defined the base and wheels as a macro. We now need to create a URDF/Xacro file to define the robot as a whole to which we will add the base as a component.

The file we need is called [`box_robot_base_only.xacro`](#) in the `rbx2_templates/urdf/box_robot` directory. Let's take a look at it now:

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

    <!-- Include all component files -->
    <xacro:include filename="$(find
rbx2_description)/urdf/materials.urdf.xacro" />

    <xacro:include filename="$(find
rbx2_description)/urdf/box_robot/base.urdf.xacro" />

    <!-- Add the base and wheels -->
    <base name="base" color="Black"/>

</robot>
```

As you can see, the file is quite simple. First we add a name for our robot to the opening `<robot>` tag (highlighted in bold above). Then we include two macro files—the materials file that defines various colors, and the base macro file we just created. Finally, we add the base to the robot by calling the `<base>` macro with a name and color.

As we will see, adding a torso, camera, laser scanner and arm are nearly as straightforward.

4.1.5 Viewing the robot's transform tree

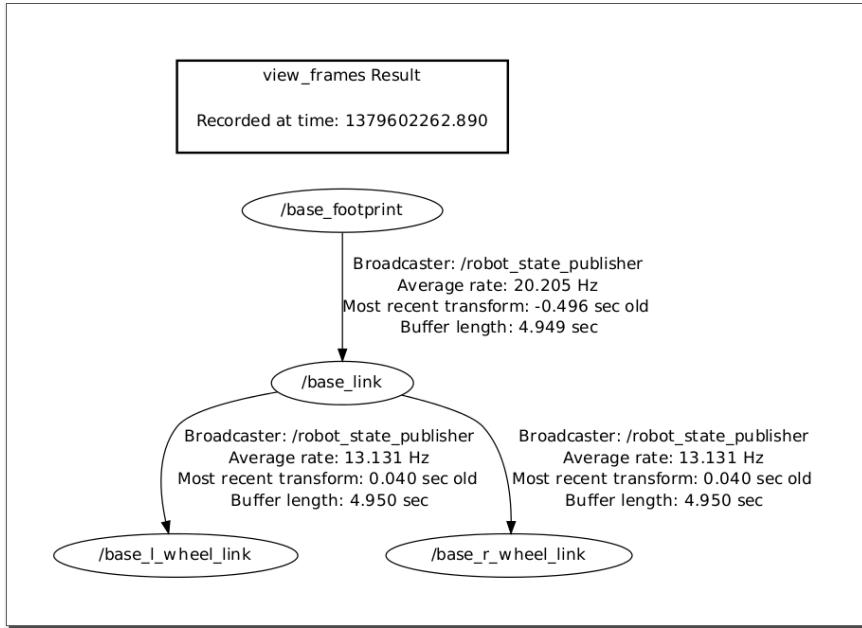
To view an image of the robot's `tf` tree, use the `view_frames` utility:

```
$ cd ~
$ rosrun tf view_frames
```

This will create a PDF file called `frames.pdf` in the current directory. It can be viewed using any PDF reader such as `evince`:

```
$ evince frames.pdf
```

which should display an image like the following:



For our base-only robot, we see that the transform tree is quite simple. Each link in our URDF model appears as a node in the tree while the arcs represent the coordinate transformations from one link to the next. The root of the tree is the `/base_footprint` link which in turn connects to the `/base_link`. The `/base_link` then branches to the left and right drive wheels.

4.1.6 Using a mesh for the base

If you have a 3D mesh for your base and/or wheels, you can use them instead of a box-and-cylinder model. The `rbx2_description` package includes STL meshes for Pi Robot's components to be used as an example. You can find the STL files in the directory `rbx2_description/meshes/pi_robot`. The mesh files were created in Google Sketchup and cleaned up using Meshlab. (See the next section on simplifying meshes.) The Sketchup files are located in the directory `rbx2_description/sketchup/pirobot`.

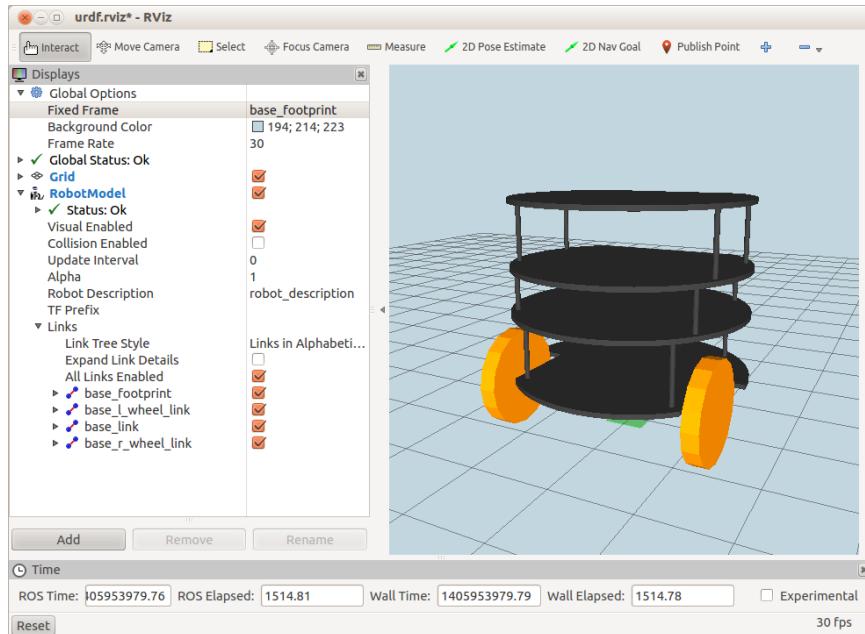
To bring up just the base, first terminate any other URDF launch file you might have running from an earlier section, then run the following:

```
$ roslaunch rbx2_description pi_robot_base_only.launch
```

If `RViz` is not still running, bring it up now:

```
$ rosrun rviz rviz -d `rospack find rbx2_description`/urdf.rviz
```

The view in RViz should look something like this:



(Remember to toggle the checkbox beside the **RobotModel** display if you are still seeing a previously loaded model.)

To see how the mesh was included in the URDF model, take a look at the file [pi_base.urdf.xacro](#) in the directory `rbx2_description/urdf/pi_robot`. The file is nearly identical to the box model of the base with a few key differences. Here are the key property lines and the block defining the base macro:

```
<property name="base_radius" value="0.152" />
<property name="base_height" value="0.241" />
<property name="ground_clearance" value="0.065" />

<property name="base_mesh_scale" value="0.0254" />

<property name="PI" value="3.1415" />

<!-- The base xacro macro -->
<macro name="base" params="name color">
  <link name="${name}_link">
    <visual>
      <origin xyz="0 0 0" rpy="0 0 ${PI/2}" />
    <geometry>
```

```

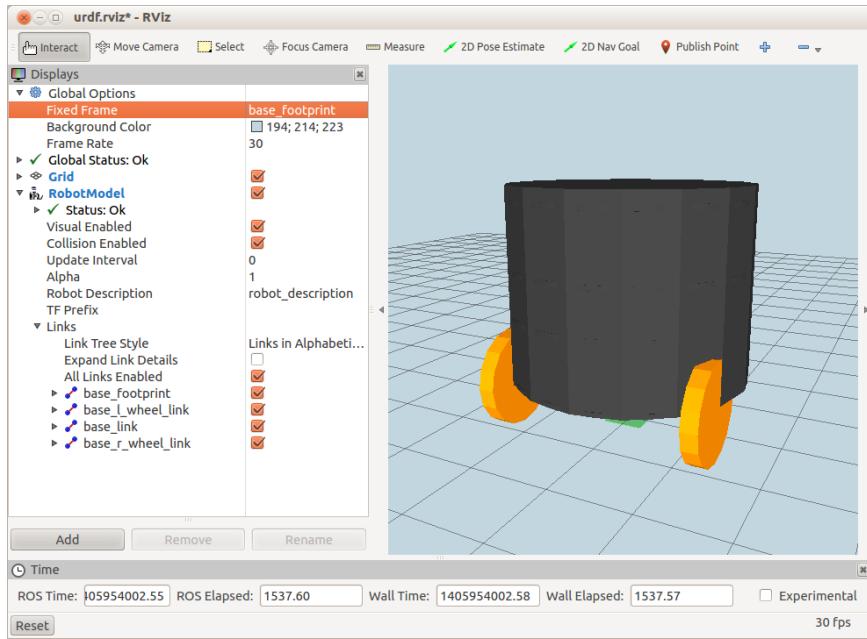
<mesh
filename="package://rbx2_description/meshes/pi_robot/pi_robot_base.stl" scale="$
{base_mesh_scale} ${base_mesh_scale} ${base_mesh_scale}" />
    </geometry>
    <material name="${color}" />
</visual>
<collision>
    <origin xyz="0 0 ${base_height/2}" rpy="0 0 0" />
    <geometry>
        <cylinder radius="${base_radius}" length="${base_height}" />
    </geometry>
</collision>
</link>
</macro>
```

Since Pi Robot's base is cylindrical rather than a box, we set its radius and height as properties. We also set the ground clearance and base mesh scale values. (More on the mesh scale below.)

The key line is highlighted in bold above. Instead of defining a simple box (or in this case a cylinder) for the visual component, we load the 3D model of the base using the `<mesh>` tag. The `filename` parameter is set to the ROS package path of the appropriate file, in this case, the file `pi_robot_base.stl` in the directory `rbx2_description/meshes/pi_robot`. Since meshes created in CAD programs will not necessarily use meters as the base unit, we also use a `scale` parameter that takes three values for the `x`, `y` and `z` scale multipliers. (The scale will almost always be the same in all three directions.) Pi's base mesh was created in Google Sketchup and it turns out that the appropriate scale factor is `0.0254`. This is the value set for the `base_mesh_scale` property near the top of the file.

Note that we use a simple cylinder rather than the mesh in the `<collision>` block. The reason is that the collision parameters are used by other ROS packages like the Navigation stack for checking that the robot is not about to run into an obstacle. It is much quicker to do collision checking with simple geometric shapes like boxes and cylinders than to use a 3D mesh which might have thousands of faces. As long as the simpler shape envelops the visual mesh, it serves just as well for collision checking. Note also that we have set the `z` origin component for the collision cylinder to `base_height/2`. This is because the coordinate frame of the base mesh is actually attached to the bottom of the mesh rather than the middle as will will explain below.

To see the collision cylinder in `RViz`, simply check the box beside **Collision Enabled** under the **RobotDisplay** and the view should look like this:

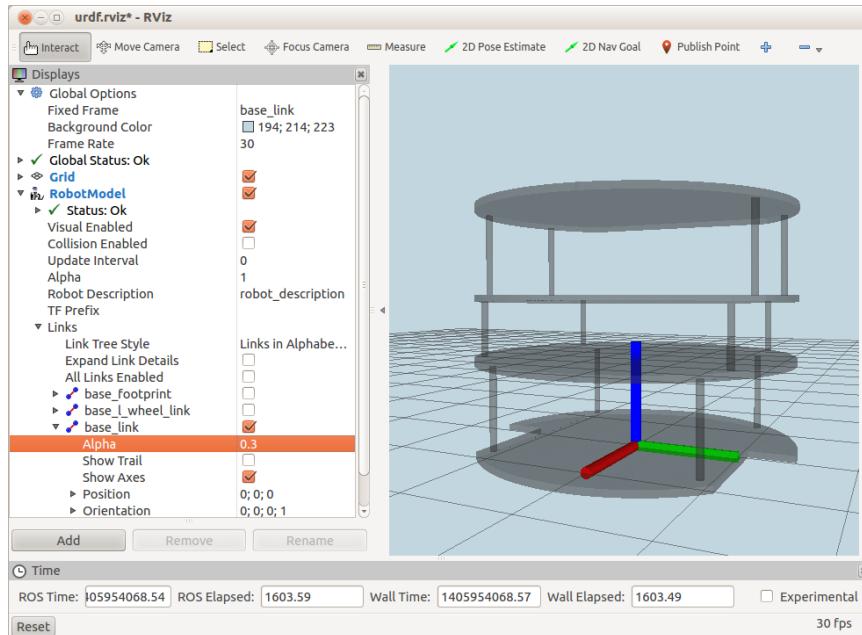


Note how the collision cylinder completely envelops the base mesh. Toggling the **Collision Enabled** checkbox is a good way to verify that your collision shapes are positioned and sized correctly. In this case, you might be wondering if the wheels will present a collision hazard since they stick out from the base cylinder. Remember that each wheel has its own collision envelope which we defined to be the same shape as the wheel itself. So in that sense the wheels are covered. If for some reason you wanted to be especially cautious, you could increase the size of the base collision cylinder to envelop the wheels as well. However, this might also prevent the robot from navigating through narrow gaps.

Computing the ground clearance for the robot depends on where the mesh defines its origin. To see the axes for the Pi Robot base mesh, set the parameters in RViz as follows:

- un-check the **Collision Enabled** checkbox
- set the **Fixed Frame** to `/base_link` under **Global Options**
- open the **Links** list under the **RobotModel** display set the **Alpha** value for the `base_link` to something like `0.3` and check the checkbox labeled **Show Axes**
- un-check the checkboxes under the Links section for all the links except the `base_link`

The resulting view in `RViz` should look something like this:



As you can see, this particular mesh defines its origin in the center of the bottom of the base. (Recall that a box or cylinder component has the origin in the middle of the component.) Fortunately the axes are oriented the way we want with the `x`-axis pointing in the robot's forward direction.

The ground clearance of the robot can now be seen to be simply the difference between the wheel radius and the wheels z-offset. So the `<joint>` definition for the `base_footprint` link looks like this:

```
<joint name="base_joint" type="fixed">
  <origin xyz="0 0 ${wheel_offset_z - wheel_radius}" rpy="0 0 0" />
  <parent link="base_link"/>
  <child link="base_footprint" />
</joint>
```

The top-level URDF/Xacro model that includes the mesh base is stored in the file [pi_robot_base_only.xacro](#) in the `rbx2_templates/urdf/pi_robot` directory and it looks like this:

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

```

<!-- Include all component files -->
<xacro:include filename="$(find
rbx2_description)/urdf/materials.urdf.xacro" />

<xacro:include filename="$(find rbox2_description)/urdf/pi_base.urdf.xacro" />

<!-- Add the base and wheels -->
<base name="base" color="Black"/>

</robot>

```

As you can see, the file is quite simple. First we add a name for our robot to the opening `<robot>` tag. Then we include two macro files, the materials file that defines various colors and the base macro file we just created. Finally, we add the base to the robot by calling the `<base>` macro and a desired color.

As we will see, adding a torso, camera and arm are just as straightforward.

4.2 Simplifying Your Meshes

Using 3D models for your robot components makes for nicer looking graphics in `RViz` but it can also put a load on your computer's CPU. It therefore pays to simplify your meshes as much as possible using a tool such as MeshLab. The biggest gains are achieved by reducing the number of vertices and faces in the mesh. In MeshLab, this can be done by running the three filters **Merge Close Vertices**, **Remove Duplicated Vertex**, and **Remove Duplicate Faces**. All three functions can be found under the menu **Filters→Cleaning and Repairing**.

Of course, if you are already familiar with another CAD program such as SolidWorks or AutoCad, you can probably find similar functions to simplify a mesh. If you can also find a function to place the coordinate axes in the center of the component, even better.

4.3 Adding a Torso

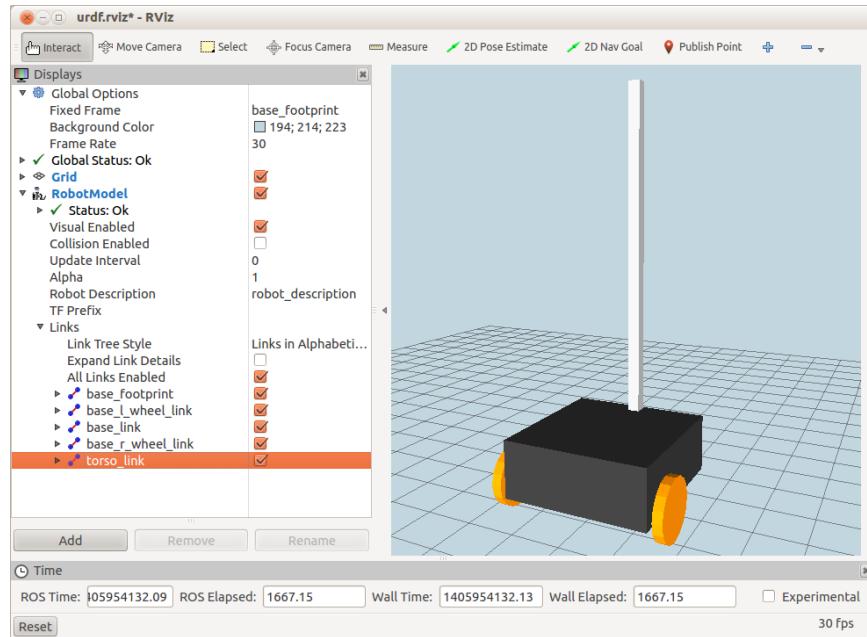
Now that we have laid the ground work for the base and wheels, adding a torso is relatively straightforward. Let's begin with the box-and-cylinder version. In this case, we will add a vertical 1"x1" post resembling a section of 8020 T-slot. To see how it will look, terminate any running URDF launch files and run the following launch file instead:

```
$ rosrun rbox2_description box_robot_with_torso.launch
```

If `RViz` is not still running, bring it up now:

```
$ rosrun rviz rviz -d `rospack find rbox2_description`/urdf.rviz
```

The view in `RViz` should look something like the image below:



(Remember to toggle the checkbox beside the **RobotModel** display if you are still seeing a previously loaded model.)

Change the **Fixed Frame** back to `/base_footprint` if you still have it set to `/base_link` from the previous section.

Note how the new `torso_link` is listed under the **Links** section of the **RobotModel** display in the left hand panel. You can turn off the display of the torso by un-checking the checkbox beside the link name.

4.3.1 Modeling the torso

Let's look at the URDF/Xacro file [`torso.urdf.xacro`](#) defining the torso:

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

    <!-- Define a number of dimensions using properties -->
    <property name="torso_size_x" value="0.0254" />
    <property name="torso_size_y" value="0.0254" />
    <property name="torso_size_z" value="0.7" />

    <!-- Define the torso -->
    <macro name="torso" params="parent name color *origin">
```

```

<joint name="${parent}_${name}_joint" type="fixed">
  <xacro:insert_block name="origin" />
  <parent link="${parent}_link"/>
  <child link="${name}_link"/>
</joint>
<link name="${name}_link">
  <visual>
    <origin xyz="0 0 0" rpy="0 0 0" />
    <geometry>
      <box size="${torso_size_x} ${torso_size_y} ${torso_size_z}" />
    </geometry>
    <material name="${color}" />
  </visual>
  <collision>
    <origin xyz="0 0 0" rpy="0 0 0" />
    <geometry>
      <box size="${torso_size_x} ${torso_size_y} ${torso_size_z}" />
    </geometry>
  </collision>
</link>
</macro>

</robot>

```

At the top of the file we define the dimensions of the torso as a post 0.7 meters tall and 1" x 1" in cross-section. We then define the torso macro as a `<joint>` and a `<link>`. The joint block connects the torso to the parent link which is passed as the `parent` parameter to the macro. The point of attachment is determined by the `*origin` block parameter which is also passed to the macro. Recall from the URDF [Xacro tutorial](#) that a block parameter is inserted using the `<xacro:insert_block>` tag as we see above. We'll see how to pass the `*origin` parameter in the next section.

4.3.2 Attaching the torso to the base

Attaching the torso to the base occurs in the file [`box_robot_with_torso.xacro`](#) found in the `rbx2_description/urdf/box_robot` directory. This is our new top-level robot model and it looks like this:

```

<?xml version="1.0"?>
<robot name="box_robot">

  <!-- Define a number of dimensions using properties -->
  <property name="torso_offset_x" value="-0.13" />
  <property name="torso_offset_y" value="0.0" />
  <property name="torso_offset_z" value="0.41" />

  <!-- Include all component files -->
  <xacro:include filename="$(find
rbx2_description)/urdf/materials.urdf.xacro" />

  <xacro:include filename="$(find rbt2_description)/urdf/base.urdf.xacro" />

```

```

<xacro:include filename="$(find rbx2_description)/urdf/torso.urdf.xacro" />

<!-- Add the base and wheels -->
<base name="base" color="Black"/>

<!-- Add the torso -->
<torso name="torso" parent="base" color="Grey">
    <origin xyz="${torso_offset_x} ${torso_offset_y} ${torso_offset_z}" rpy="0
0 0" />
</torso>

</robot>

```

As you can see, the file is similar to our base-only robot model. First we add the three torso offset parameters that determine where the torso is attached to the base. We also include the `torso.urdf.xacro` file. Finally, we attach the torso to the base by calling the `<torso>` macro with the `parent` parameter set to "base" and the `xyz` part of the `<origin>` block parameter set to the torso offset values. Note that we could also assign orientation values to the `rpy` parameter but in this case the default vertical orientation of the torso is what we want.

4.3.3 Using a mesh for the torso

Pi Robot uses an 8020 T-slot post for a torso. As it turns out, the 3D Warehouse for Google Sketchup has a model for just such an item so we can use it as a mesh. Pi's mesh torso is defined in the file `pi_torso.urdf.xacro` in the directory `rbx2_description/meshes/pi_robot`. The file looks like this:

```

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

<!-- Define a number of dimensions using properties -->
<property name="torso_size_x" value="0.025" />
<property name="torso_size_y" value="0.025" />
<property name="torso_size_z" value="0.885" />

<property name="torso_mesh_scale" value="0.0128" />

<!-- Define the torso -->
<macro name="torso" params="parent name color *origin">
    <joint name="${parent}_${name}_joint" type="fixed">
        <xacro:insert_block name="origin" />
        <parent link="${parent}_link"/>
        <child link="${name}_link"/>
    </joint>
    <link name="${name}_link">
        <visual>
            <origin xyz="0 0 0" rpy="0 0 0" />
            <geometry>
                <mesh
filename="package://rbx2_description/meshes/pi_robot/t_slot.stl" scale="$
{torso_mesh_scale} ${torso_mesh_scale} ${torso_mesh_scale}" />
            </geometry>
    
```

```

        <material name="${color}" />
    </visual>
    <collision>
        <origin xyz="0 0 0" rpy="0 0 0" />
        <geometry>
            <box size="${torso_size_x} ${torso_size_y} ${torso_size_z}" />
        </geometry>
    </collision>
</link>
</macro>
</robot>

```

The `<torso>` macro takes the parent and `*origin` as parameters so we know how to attach it to the robot. As with Pi Robot's base mesh, we include a `<mesh>` tag in the geometry section of the `<link>` block. In this case, the mesh filename points to the package location of the `t_slot.stl` file. We also scale the mesh by a factor stored in the property `torso_mesh_scale`. As it turns out, we need a value of `0.0127` to make the scale the T-slot mesh the same as the rest of the robot.

4.3.4 Adding the mesh torso to the mesh base

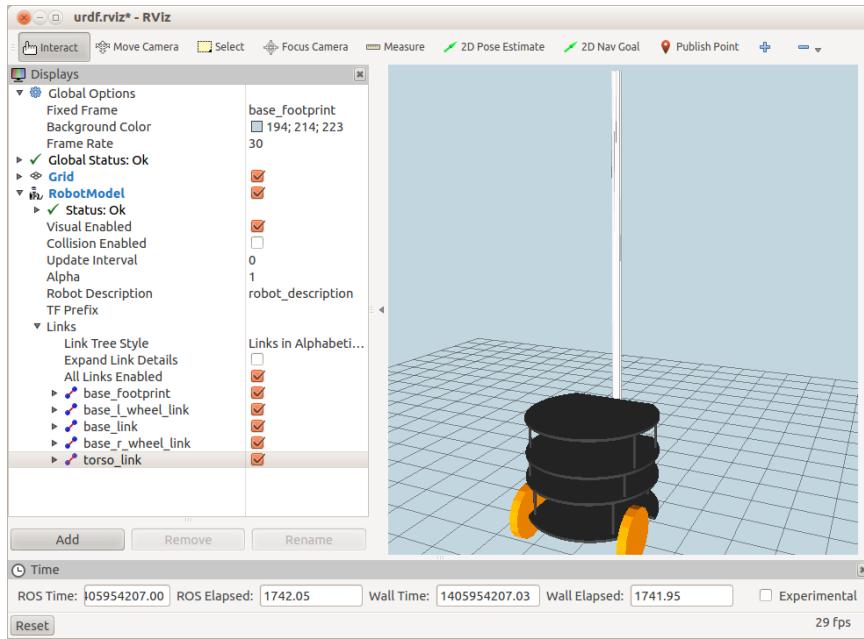
Adding the mesh torso to the rest of Pi Robot is the same as we did for the Box Robot. To see how it looks, run the following launch file:

```
$ rosrun urdf_create pi_robot_with_torso.launch
```

If `RViz` is not still running, bring it up now:

```
$ rosrun rviz rviz -d `rospack find rbx2_description`/urdf.rviz
```

The view in `RViz` should look like the following:



(Remember to toggle the checkbox beside the **RobotModel** display if RViz is still showing a previously loaded model.)

The top-level URDF/Xacro file for this version of Pi Robot is called [pi_robot_with_torso.xacro](#) in the directory `rbx2_description/urdf/pi_robot` and looks like this:

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

    <!-- Define a number of dimensions using properties -->
    <property name="torso_offset_x" value="-0.13" />
    <property name="torso_offset_y" value="0.0" />
    <property name="torso_offset_z" value="0.088" />

    <!-- Include all component files -->
    <xacro:include filename="$(find
rbx2_description)/urdf/materials.urdf.xacro" />

    <xacro:include filename="$(find
rbx2_description)/urdf/pi_robot/pi_base.urdf.xacro" />

    <xacro:include filename="$(find
rbx2_description)/urdf/pi_robot/pi_torso.urdf.xacro" />

    <!-- Add the base and wheels -->
    <base name="base" color="Black"/>
```

```

<!-- Add the torso -->
<torso name="torso" parent="base" color="Grey">
    <origin xyz="${torso_offset_x} ${torso_offset_y} ${torso_offset_z}" rpy="0
0 0" />
</torso>

</robot>

```

As with the Box Robot model, first we define the torso offsets relative to the base. Then we include the materials file and the two mesh macro files, one for the base and one for the torso. Finally, we call the base macro followed by the torso macro with the `origin` block parameters set to the desired point of attachment.

4.4 Measure, Calculate and Tweak

Getting model components to line up correctly can sometimes be a little tricky. Unless you are really good at visualizing objects in 3D, figuring out offsets and rotation angles sometimes takes a bit of trial and error.

If you are using meshes for your parts, get out a ruler and measure them as well. Set these dimensions as properties in your URDF/Xacro files so that you can use simple boxes and cylinders for the `<collision>` blocks. You can also use the dimensions in your offsets. For example, if you know that part A is displaced in the `x`-direction from part B by half the width of part A, then use an expression such as `${A_WIDTH/2}` as the `x`-component of the joint connecting the two parts.

Even with meshes and careful measurements, you may see gaps or overlaps of the robot parts in `RViz`. This is the "tweaking" part of the process. Change your offsets by small amounts and check `RViz` for the result. It is often helpful to set the `Alpha` values for a component to something less than 1 (e.g `0.5`) so that you can see if another component is actually penetrating it. Then adjust the offset until the parts no longer overlap.

4.5 Adding a Camera

The procedure for adding a camera to the robot is nearly identical to adding the torso. We will add a camera to the torso but you could use the same method to add it directly to the base if your robot does not have a torso. For our Box Robot, we will start with a model that uses the same dimensions as the Kinect for the camera.

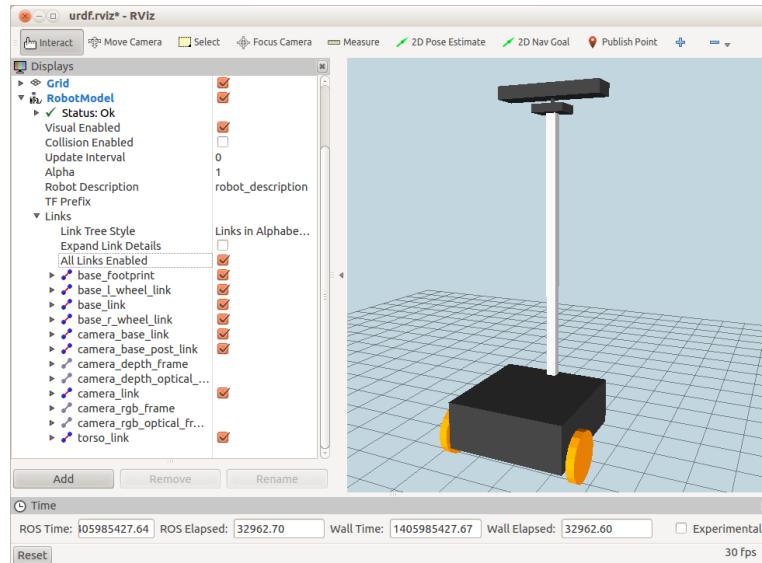
Before looking at the URDF files, let's view the final result using the following command:

```
$ rosrun rviz box_robot.launch
```

If `RViz` is not still running, bring it up now:

```
$ rosrun rviz rviz -d `rospack find rbx2_description`/urdf.rviz
```

The view in RViz should look like this:



As you can see, we have modeled the Kinect using three pieces: a box for the base, a cylinder for the supporting post and another box for the camera body. Let's now look at the details.

4.5.1 Placement of the camera

Bear in mind that current depth cameras like the Kinect and Xtion Pro cannot compute the distance to an object that lies within about 0.5 meters (about 2 feet) of the camera. If you ultimately plan for your robot to use an arm and gripper to pick up objects, then it is generally a good idea to mount the camera fairly high up on the robot. This is one of the reasons we are using a long torso on our robot model. By placing the camera high above the base, the robot can look down at a table surface or the ground and objects will be outside of its blind spot. The camera's depth image can then be used to compute the 3D pose of objects and guide the motion of an arm during a pick and place task.

Another reason to mount the camera higher up on the robot is so that it will be at a better height for face detection and recognition if you plan for your robot to interact with people.

4.5.2 Modeling the camera

The box Kinect URDF/Xacro file is called [kinect_box.urdf.xacro](#) in the directory `rbx2_description/urdf/box_robot` and the first part of the file looks like this:

```
<property name="kinect_body_x" value="0.07271" />
<property name="kinect_body_y" value="0.27794" />
<property name="kinect_body_z" value="0.033" />

<property name="kinect_base_x" value="0.072" />
<property name="kinect_base_y" value="0.085" />
<property name="kinect_base_z" value="0.021" />

<property name="kinect_base_post_height" value="0.016" />
<property name="kinect_base_post_radius" value="0.005" />

<property name="PI" value="3.1415" />

<!-- Define a box-shaped camera link for the Kinect -->
<macro name="camera" params="parent name color *origin">
  <joint name="${parent}_${name}_joint" type="fixed">
    <xacro:insert_block name="origin" />
    <parent link="${parent}_link"/>
    <child link="${name}_base_link"/>
  </joint>

  <link name="${name}_base_link">
    <visual>
      <origin xyz="0 0.0 0.0" rpy="0 0 0" />
      <geometry>
        <box size="${kinect_base_x} ${kinect_base_y} ${kinect_base_z}" />
      </geometry>
      <material name="${color}" />
    </visual>
    <collision>
      <origin xyz="0 0 0" rpy="0 0 0" />
      <geometry>
        <box size="${kinect_base_x} ${kinect_base_y} ${kinect_base_z}" />
      </geometry>
    </collision>
  </link>
</macro>
```

First we store the dimensions of the Kinect in a number of properties separating out the three pieces: the base, the supporting post, and the camera body.

Next we define a box-shaped link for the camera base and a joint that connects it to the parent link (which will be the torso as we will see in the next section). Note how we use the variable `${name}` instead of the fixed string "camera" for the prefix for the link and joint names. This allows us to attach more than one camera if desired and give each camera a different name which then propagates through its set of links and joints.

```
<joint name="camera_base_post_joint" type="fixed">
  <origin xyz="0 0 ${kinect_base_z + kinect_base_post_height}/2" rpy="0
0 0" />
```

```

<parent link="${name}_base_link" />
<child link="${name}_base_post_link" />
</joint>

<link name="${name}_base_post_link">
    <visual>
        <origin xyz="0 0 0" rpy="0 0 0" />
        <geometry>
            <cylinder radius="${kinect_base_post_radius}" length="$
{kinect_base_post_height}" />
        </geometry>
        <material name="${color}" />
    </visual>
    <collision>
        <origin xyz="0 0 0" rpy="0 0 0" />
        <geometry>
            <cylinder radius="${kinect_base_post_radius}" length="$
{kinect_base_post_height}" />
        </geometry>
    </collision>
</link>

<joint name="${name}_base_joint" type="fixed">
    <origin xyz="0 0 ${kinect_base_post_height + kinect_body_z}/2" rpy="0
0 0" />
    <parent link="${name}_base_post_link" />
    <child link="${name}_link" />
</joint>

<link name="${name}_link">
    <visual>
        <origin xyz="0 0 0" rpy="0 0 0" />
        <geometry>
            <box size="${kinect_body_x} ${kinect_body_y} ${kinect_body_z}" />
        </geometry>
        <material name="${color}" />
    </visual>
    <collision>
        <origin xyz="0 0 0" rpy="0 0 0" />
        <geometry>
            <box size="${kinect_body_x} ${kinect_body_y} ${kinect_body_z}" />
        </geometry>
    </collision>
</link>

```

Here the construction process continues as we attach the cylindrical post to the camera base at the bottom and the camera body at the top.

The rest of the URDF/Xacro file defines the relationship between the various optical and depth frames of the camera. The offsets were copied from the "official" Kinect URDF file found in the `turtlebot_description` package. For example, the relation between the camera body (`camera_link` frame) and the depth frame is given by the block:

```

<joint name="${name}_depth_joint" type="fixed">
  <origin xyz="0 0.0125 0" rpy="0 0 0" />
  <parent link="${name}_link" />
  <child link="${name}_depth_frame" />
</joint>

```

Here we see that the depth frame is laterally displaced from the center of the camera body by 1.25 cm. This relation as well as the others specified in the camera file enable the `robot_state_publisher` to include the various camera reference frames in the `tf` tree. This in turn connects the camera frames to the rest of the robot which allows the robot to view an object in the depth frame but know where it is relative to the robot's base (for example). All the complicated frame transformations are done for you by ROS thanks to the `tf` library.

4.5.3 Adding the camera to the torso and base

Attaching the Kinect to the torso occurs in the file `box_robot_with_kinect.xacro` found in the `rbx2_description/urdf/box_robot` directory. This is our new top-level robot model and it looks like this:

```

<?xml version="1.0"?>
<robot name="box_robot">

  <!-- Define a number of dimensions using properties -->
  <property name="torso_offset_x" value="-0.13" />
  <property name="torso_offset_y" value="0.0" />
  <property name="torso_offset_z" value="0.41" />

  <property name="camera_offset_x" value="0.0" />
  <property name="camera_offset_y" value="0.0" />
  <property name="camera_offset_z" value="0.3605" />

  <!-- Include all component files -->
  <xacro:include filename="$(find
rbx2_description)/urdf/materials.urdf.xacro" />

  <xacro:include filename="$(find
rbx2_description)/urdf/box_robot/base.urdf.xacro" />

  <xacro:include filename="$(find
rbx2_description)/urdf/box_robot/torso.urdf.xacro" />

  <xacro:include filename="$(find
rbx2_description)/urdf/box_robot/kinect_box.urdf.xacro" />

  <!-- Add the base and wheels -->
  <base name="base" color="Black"/>

  <!-- Add the torso -->
  <torso name="torso" parent="base" color="Grey">
    <origin xyz="${torso_offset_x} ${torso_offset_y} ${torso_offset_z}" rpy="0
0 0" />

```

```

</torso>

<!-- Add the camera -->
<camera name="camera" parent="torso" color="Black">
  <origin xyz="${camera_offset_x} ${camera_offset_y} ${camera_offset_z}"
rpy="0 0 0" />
</camera>

</robot>
```

As you can see, the file is similar to our base-plus-torso robot model. First we add the three camera offset parameters that determine where the camera is attached to the torso. We also include the `kinect_box.urdf.xacro` file. Next we attach the torso to the base using the `<torso>` macro and finally, we attach the camera to the torso by calling the `<camera>` macro with the `parent` parameter set to the torso and the `origin` parameters set to the camera offset values.

To use an Asus Xtion Pro camera instead of a Kinect, terminate the previous launch file and run the command:

```
$ rosrun rviz rviz -d `rospack find rbx2_description`/urdf.rviz
```

If RViz is not still running, bring it up now:

```
$ rosrun rviz rviz -d `rospack find rbx2_description`/urdf.rviz
```

This launch file calls up the `box_robot_with_xtion.xacro` model which is nearly identical to the Kinect version but uses different box dimensions for the Xtion.

4.5.4 Viewing the transform tree with torso and camera

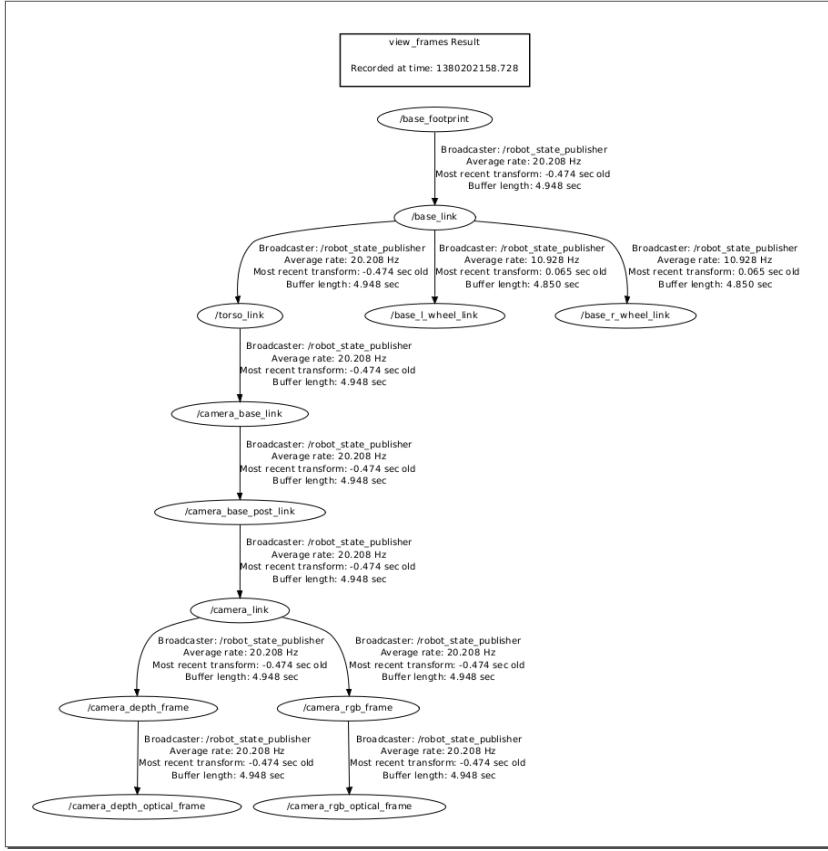
The transform tree of our robot should now be a little more interesting. Let's use `view_frames` to take a look at it:

```
$ cd /tmp
$ rosrun tf view_frames
```

This will create a PDF file called `frames.pdf` in the current directory. It can be viewed using using any PDF reader such as `evince`:

```
$ evince frames.pdf
```

And the `tf` tree should look like the following:



Note how the `/base_link` frame connects to the `/torso_link` which in turn connects to the `/camera_link`. The `/camera_link` then branches into depth frames on the left and RGB frames on the right. Each arc represents the coordinate transformation between adjacent links and not only does the `robot_state_publisher` continually update these transformations should any of the links move, it also computes the transformations between non-adjacent links by combining the transforms along the path that connects them. For example, if a ROS vision node detects an object in the camera depth frame, we can use the `tf` library to get its coordinates in the base frame.

4.5.5 Using a mesh for the camera

An 3D model for the Kinect has been available in the TurtleBot ROS packages for quite some time and we have copied a simplified version of it into the `rbx2_description` package under the `meshes` subdirectory. The macro using the mesh can be found in the file [`kinect.urdf.xacro`](#) in the directory `rbx2_description/urdf/pi_robot`.

And the model for Pi Robot that uses the mesh along with the base and torso camera is called `pi_robot_with_kinect.xacro`.

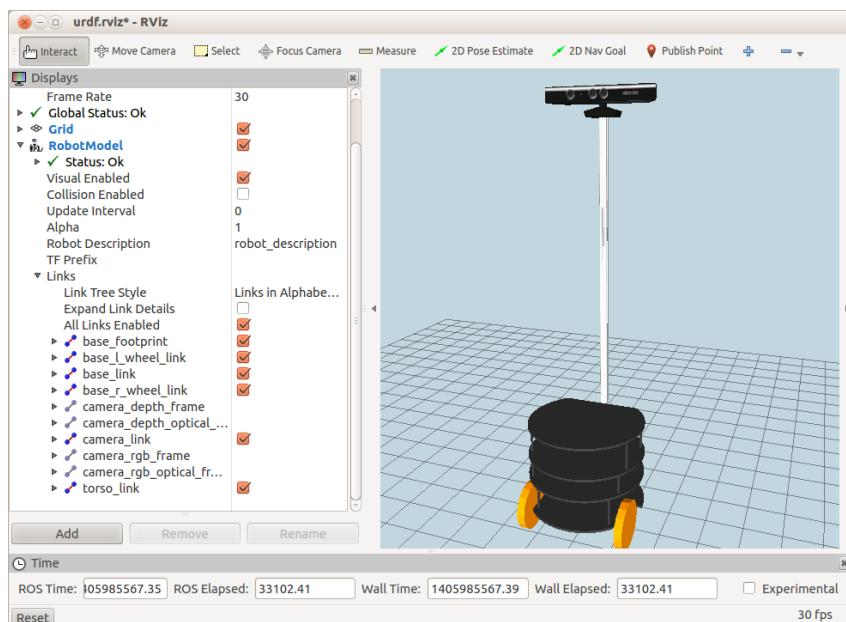
The description of these files is essentially the same as we have already covered so let us simply fire up the model to see how it looks:

```
$ rosrun rviz rviz -d `rospack find rbx2_description`/urdf.rviz
```

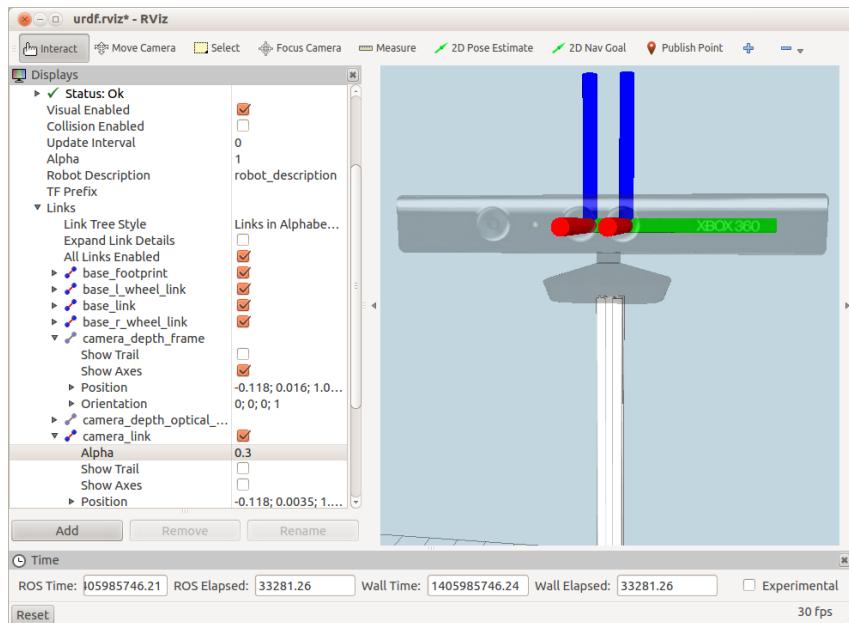
If RViz is not still running, bring it up now:

```
$ rosrun rviz rviz -d `rospack find rbx2_description`/urdf.rviz
```

And the view in RViz should look like the following:



And here is closer view of the mesh including the coordinate axes for the depth and RGB camera frames:



4.5.6 Using an Asus Xtion Pro instead of a Kinect

The `rbx2_description` package includes launch files and URDF/Xacro models for both the Box Robot and Pi Robot using an Asus Xtion Pro camera instead of a Kinect. To launch the Box Robot model with the Xtion, use the command:

```
$ rosrun rviz rviz -d `rospack find rbx2_description`/urdf.rviz
```

If `RViz` is not still running, bring it up now:

```
$ rosrun rviz rviz -d `rospack find rbx2_description`/urdf.rviz
```

And to launch the Pi Robot model with a mesh version of the Xtion Pro, run the command:

```
$ rosrun rviz rviz -d `rospack find rbx2_description`/urdf.rviz
```

4.6 Adding a Laser Scanner (or other Sensors)

Adding a laser scanner or another type of sensor to your robot is relatively straightforward. Just as we did for the torso or camera, first we need a model of the

sensor itself which could just be a box or cylinder of a certain size. Then we need to attach it to the robot at the appropriate location.

4.6.1 Modeling the laser scanner

A box-and-cylinder model of a Hokuyo laser scanner can be found in the file [laser.urdf.xacro](#) under `rbx2_description/urdf/sensors`. The model is fairly simple so we won't list it out here. The file [box_robot_with_laser.xacro](#) shows how to attach the laser to the base:

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

    <!-- Define a number of dimensions using properties -->
    <property name="laser_offset_x" value="0.123" />
    <property name="laser_offset_y" value="0.0" />
    <property name="laser_offset_z" value="0.08" />

    <!-- Include all component files -->
    <xacro:include filename="$(find
rbx2_description)/urdf/materials.urdf.xacro" />

    <xacro:include filename="$(find
rbx2_description)/urdf/box_robot/base.urdf.xacro" />

    <xacro:include filename="$(find
rbx2_description)/urdf/sensors/laser.urdf.xacro" />

    <!-- Add the base and wheels -->
    <base name="base" color="Black"/>

    <!-- Add the laser -->
    <laser parent="base" color="DarkGrey">
        <origin xyz="${laser_offset_x} ${laser_offset_y} ${laser_offset_z}" rpy="0
0 0" />
    </laser>

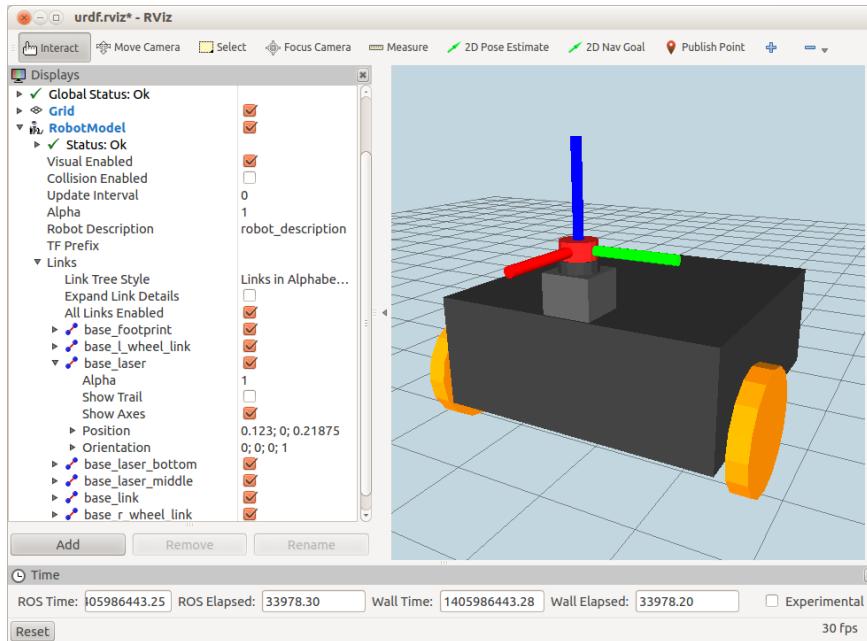
</robot>
```

To see how it looks in RViz, terminate any current URDF launch files and run:

```
$ roslaunch rbx2_description box_robot_base_with_laser.launch
```

If RViz is not still running, bring it up now:

```
$ rosrun rviz rviz -d `rospack find rbx2_description`/urdf.rviz
```



In the image above, we have checked the **Show Axes** checkbox for the `base_laser` link. This allows us to see that the base laser frame is located in the middle of the upper cylinder and that the axes are oriented in the same direction as the base. (Remember that the red axis is in the x-direction and that we want it pointing forward.)

Other sensors can be added in the same way. For example, if you have a Ping(TM) sonar sensor mounted on the robot, you could model it as a single cylinder and then attach that model to the base the same way we did for the laser scanner.

4.6.2 Attaching a laser scanner (or other sensor) to a mesh base

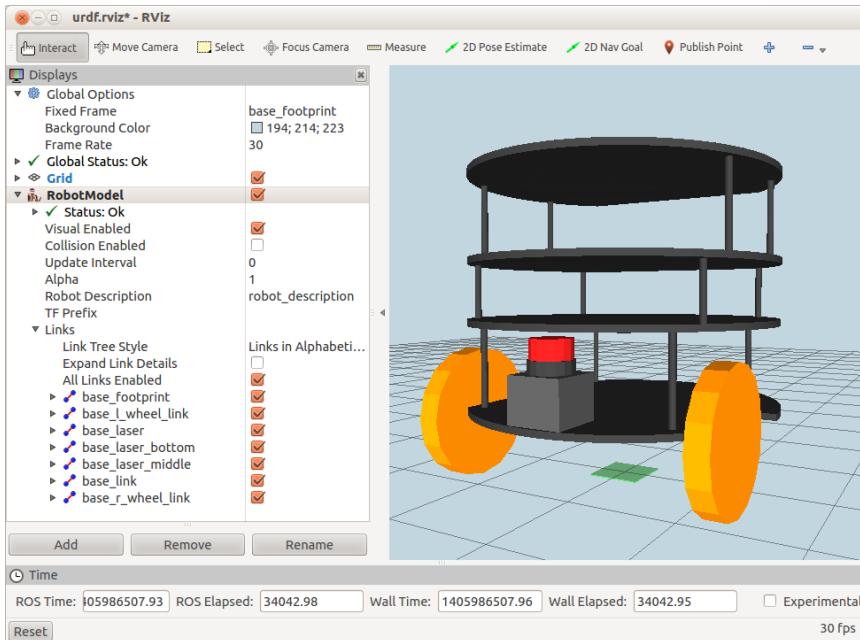
We can use the same `laser.urdf.xacro` file to add the scanner to Pi Robot's mesh base. The file that does the trick is `pi_robot_base_with_laser.xacro` in the directory `rbx2_description/urdf/pi_robot`. The file is nearly identical to the one we used for Box Robot but now we include the mesh for the base and adjust the offsets for the laser scanner to place it in the right place. To see the result, terminate the box model launch file and run the command:

```
$ rosrun rbx2_description pi_robot_base_with_laser.launch
```

If `RViz` is not still running, bring it up now:

```
$ rosrun rviz rviz -d `rospack find rbx2_description`/urdf.rviz
```

The view in RViz should look like this:



4.6.3 Configuring the laser node launch file

Once you have added your laser scanner or other sensor to your URDF model, you still need to launch a node for the device itself. In the case of a Hokuyo laser scanner, you can use the Hokuyo node that has been available in most versions of ROS since the beginning. A sample launch file called `hokuyo.launch` can be found in the `rbx2_bringup/nodes` directory and looks like this:

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

  <!-- Run the Hokuyo laser scanner node -->
  <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>
```

The `min_ang` and `max_ang` parameters (given in radians) should match your scanner's specs* and the `frame_id` parameter needs to be set to the value you used in your URDF model. The `frame_id` is what allows `tf` to locate the points in the laser scan in proper geometric relation to the rest of your robot.

* Note: you might actually need to set the `min_ang` and `max_ang` values smaller than the full range of your scanner if parts of the robot get in the way of the full range as in the image above of Pi Robot's base.

4.7 Adding a Pan and Tilt Head

Placing the camera on a pair of pan and tilt servos can add a lot of functionality to your robot. However, modeling the servos and the joints between them can be a little tricky. Fortunately, the folks at Willow Garage long ago created a URDF/Xacro model of the TurtleBot arm which uses the same Dynamixel servos and brackets that we will use for our pan-and-tilt head.

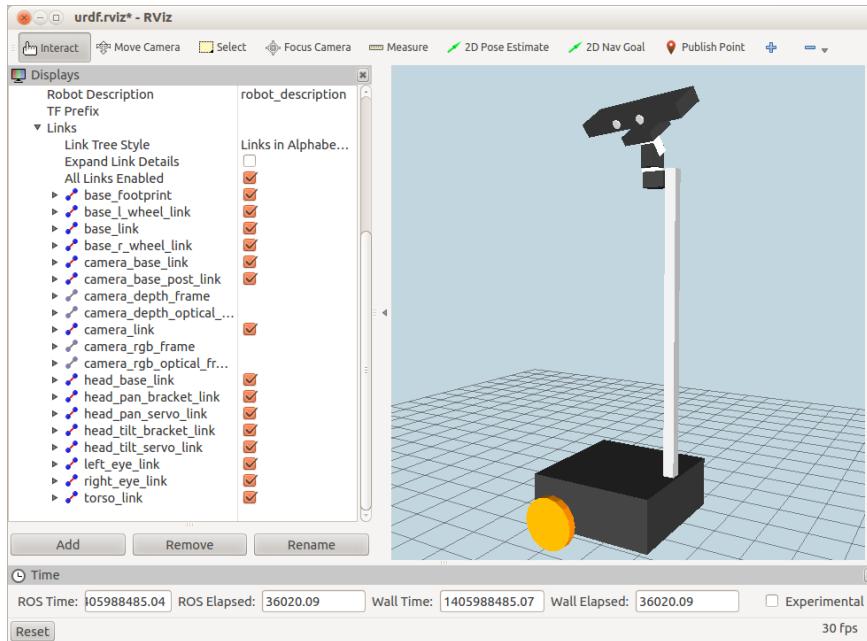
For our Box Robot, we will use simple boxes to model the servos and brackets. You can try it out using the following command:

```
$ roslaunch rbx2_description box_robot_with_pan_tilt_head.launch
```

If `RViz` is not still running, bring it up now:

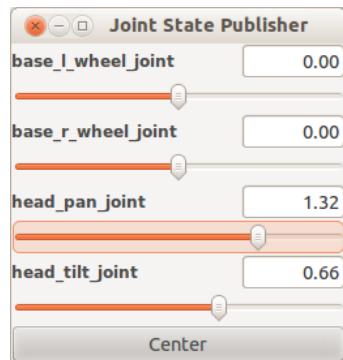
```
$ rosrun rviz rviz -d `rospack find rbx2_description`/urdf.rviz
```

After a toggling the Robot Model display, the view should look like the following:



In the meantime, the Joint State Publisher window will now include two new slider controls for the pan and tilt joints as shown on the right. For the image above, the controls were used to pan the head to the left and tilt it downward. With the **Links** sub-menu expanded under the **RobotModel** display, we can see the links that make up the brackets and servos of the pan-and-tilt head.

NOTE: Recall that ROS uses the right-hand rule for coordinate frames and rotations with the base frame oriented with the x-axis pointing forward, the y-axis pointing to the left, and the z-axis pointing straight up. Since we have mounted the torso and camera directly onto the base without any explicit rotational offsets, the camera's base frame and pan servo also have their z-axis pointing straight upward. Using the right-hand rule with the thumb pointing along the z-axis and the fingers curling in the direction of the rotation, we see that a positive panning of the camera should rotate it to the left which you can confirm with the slide controls. Similarly, the y-axis of the head-tilt servo body points to the left when the camera is pointing forward. Pointing your thumb in this direction, the fingers curl downward indicating that a positive tilt rotation should rotate the camera downward which you can again verify with the slide controls.



4.7.1 Using an Asus Xtion Pro instead of a Kinect

To use an Asus Xtion Pro camera instead of a Kinect on the pan-and-tilt head, run the following launch file:

```
$ roslaunch rbx2_description box_robot_with_pan_tilt_head_xtion.launch
```

If RViz is not still running, bring it up now:

```
$ rosrun rviz rviz -d `rospack find rbx2_description`/urdf.rviz
```

If RViz is still running from a previous session, don't forget to toggle the checkbox beside the **RobotModel** display to update the model.

4.7.2 Modeling the pan-and-tilt head

The overall model for this robot is found in the file [box_robot_with_pan_tilt_head.xacro](#) in the directory `rbx2_description/urdf/box_robot`. This file is nearly the same as the `box_robot_with_kinect.xacro` file we examined earlier although this time we include the file [box_pan_tilt_head.urdf.xacro](#) instead of the `kinect.urdf.xacro` file. The `box_pan_tilt_head.urdf.xacro` itself includes the `kinect.urdf.xacro` as well as the link and joint definitions for the servos and brackets. This way we can attach the camera to the head tilt bracket. The `box_pan_tilt_head.urdf.xacro` file also includes another file called [dynamixel_box_hardware.xacro](#). This file defines a number of properties to hold the dimensions of the servos and various types of Dynamixel brackets. It also defines a collection of macros that models the servos and brackets using simple boxes. (We'll use meshes for Pi Robot below.)

We won't go through the servo URDF/Xacro files line-by-line as we have done with the other components as it would take too much space. The basic idea is that we define a link for each servo and bracket and joints in between them. However, let's at least look at a couple of XML blocks that illustrate how to model revolute joints. Referring to the file [box_pan_tilt_head.urdf.xacro](#), the following lines define the `head_pan` joint and link in terms of an AX12 servo and F3 bracket:

```
<dynamixel_AX12_fixed parent="head_base" name="head_pan_servo">
  <origin xyz="-0.012 0 ${-AX12_WIDTH/2}" rpy="${M_PI/2} 0 ${-M_PI/2}" />
</dynamixel_AX12_fixed>

<bioloid_F3_head_revolute parent="head_pan_servo" name="head_pan_bracket"
  joint_name="head_pan" ulimit="2.53" llimit="-2.53" vlimit="1.571" color="$
  {color}">
  <origin xyz="0 ${AX12_WIDTH/2 + 0.005} 0.012" rpy="${-PI/2} ${PI/2} $
  {PI}" />
```

```

<axis xyz="0 0 -1" />
</bioloid_F3_head_revolute>
```

The macro called `dynamixel_AX12_fixed` can be found in the file `dynamixel_box_hardware.xacro` and looks like this:

```

<macro name="dynamixel_AX12_fixed" params="parent name *origin">
  <joint name="${name}_joint" type="fixed">
    <xacro:insert_block name="origin" />
    <parent link="${parent}_link"/>
    <child link="${name}_link" />
  </joint>

  <link name="${name}_link">
    <inertial>
      <mass value="0.00001" />
      <origin xyz="0 0 0" />
      <inertia ixx="1.0" ixy="0.0" ixz="0.0"
               iyy="1.0" iyz="0.0"
               izz="1.0" />
    </inertial>

    <visual>
      <origin xyz="0 0 0" rpy="0 0 0" />
      <geometry>
        <box size="${AX12_HEIGHT} ${AX12_WIDTH} ${AX12_DEPTH}" />
      </geometry>
      <material name="Black"/>
    </visual>

    <collision>
      <origin xyz="0 0 -0.01241" rpy="0 0 0" />
      <geometry>
        <box size="${AX12_HEIGHT} ${AX12_WIDTH} ${AX12_DEPTH}" />
      </geometry>
    </collision>
  </link>
</macro>
```

This macro is fairly straightforward—it defines the link as a box with the same dimensions of an AX-12 servo and it defines a fixed joint between itself and the parent which in this case is the `head_base_link`.

The second macro above, `bioloid_F3_head_revolute`, does the work of actually panning the head. This macro is also defined in the `dynamixel_box_head_hardware.xacro` file and looks like this:

```

<macro name="bioloid_F3_head_revolute" params="parent name joint_name llimit
  ulimit color *origin *axis">
  <joint name="${joint_name}_joint" type="revolute">
    <insert block name="origin"/>
    <insert block name="axis"/>
```

```

<parent link="${parent}_link"/>
<limit lower="${llimit}" upper="${ulimit}" velocity="${vlimit}"
effort="1.0" />
<child link="${name}_link" />
</joint>

<link name="${name}_link">
<inertial>
<mass value="0.00001" />
<origin xyz="0 0 0" />
<inertia ixx="1.0" ixy="0.0" ixz="0.0"
iyy="1.0" iyz="0.0"
izz="1.0" />
</inertial>

<visual>
<origin xyz="0 0 0" rpy="0 0 0" />
<geometry>
<box size="${F3_DEPTH} ${F3_WIDTH} ${F3_HEIGHT}" />
</geometry>
<material name="${color}" />
</visual>

<collision>
<origin xyz="0 0 0" rpy="0 0 0" />
<geometry>
<box size="${F3_DEPTH} ${F3_WIDTH} ${F3_HEIGHT}" />
</geometry>
</collision>
</link>
</macro>

```

Note how the joint type in this case (highlighted in bold above) is **revolute** instead of fixed. A revolute joint takes parameters `lower` and `upper` for the lower and upper rotation limits, specified in radians. These are set for a particular joint by passing the values `${llimit}` and `${ulimit}` in the code above. An AX-12 servo has an operating range of 300 degrees, so 150 degrees to either side of center. The value of 2.53 radians that is passed in for the lower and upper limits in the `bioloid_F3_head_revolute` definition above is about 145 degrees—just a little less than the full range so we don't force the servo against its own limits.

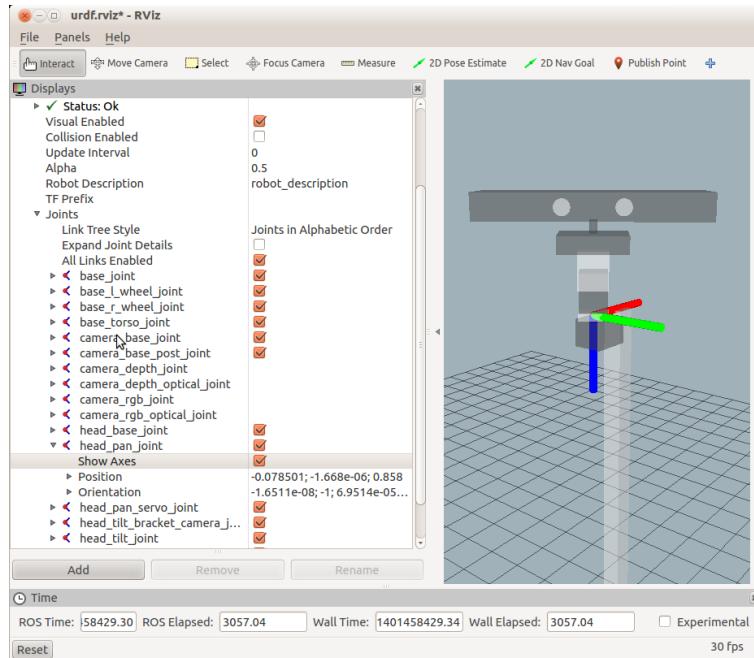
The joint also takes parameters `velocity` and `effort` which are limits on its dynamics that should be respected by a real servo controller. A velocity limit of 1.571 radians per second is equivalent to about 90 degrees per second which is moderately fast, especially if the servo is panning a camera. Although we have set the effort to 1.0 to show the syntax, neither the `dynamixel_motor` nor the `arbotix` package makes use of this value.

Another key property of a revolute joint is this `<axis>` tag, also highlighted in bold above. The `xyz` components of the `axis` tag are typically either 0 or 1 and define the

rotation axis of the joint. These parameters are passed as a block argument called **axis** similar to the way we pass the **origin** parameters. In the case of the head pan joint, we pass the values "`0 0 -1`" which defines a rotation in the negative z-direction. In the next section we'll show how these axis components are determined.

4.7.3 Figuring out rotation axes

How did we know to choose the negative z-direction for the head pan rotation? Take a look at the image below:



Here we have set the **Alpha** value for the **RobotModel** display to `0.5` so that we can see the link and joint frames more easily. Next, we have set the **Link Tree Style** to *Joints in Alphabetical Order* instead of the default *Links in Alphabetical Order*. Looking down the list of joints, we find the `head_pan_joint` and check the box beside **Show Axes**. Referring back to the image above, we see that panning the head requires a rotation around the z-axis (shown in blue in `RViz`) but that the reference frame is inverted with the z-axis pointing downward. Therefore, we need to specify the rotation axis as the negative z-axis ("`0 0 -1`" in the URDF model) which would then be pointing upward. If you use the right-hand rule and point your thumb upward, your fingers should curl in the direction of counter-clockwise rotation of the head which is the convention ROS uses to define a positive rotation. You can verify that this is the

case by returning to the joint state GUI and moving the `head_pan_joint` slider control toward increasing values (i.e. to the right). The head should then pan counter-clockwise as a result.

You can apply this same analysis to any joint for which you need to figure out the rotation axis. First, turn on the axes display for the joint in question and use the right-hand rule to determine the sign of the axis the joint rotates around. If you do the same analysis for the `head_tilt_joint`, you will see that the rotation is around the positive y-axis (shown in green in RViz) so we pass the values "0 1 0" for the axis parameter for the head tilt joint in our URDF model.

4.7.4 A pan and tilt head using meshes on Pi Robot

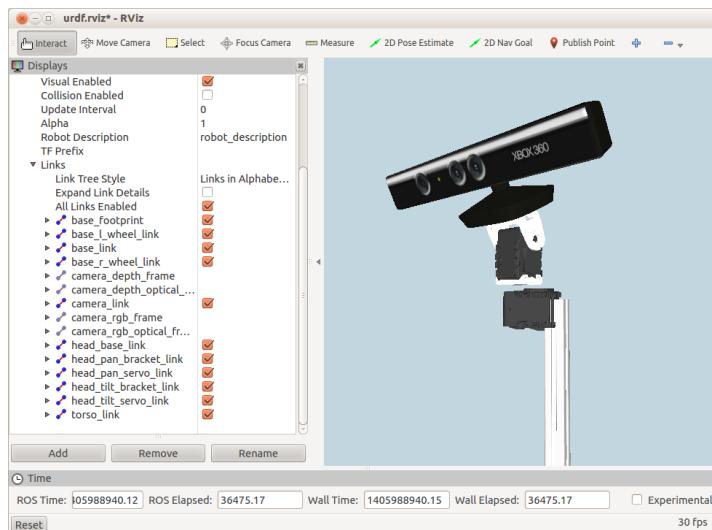
We can also use meshes for the Dynamixel servos and brackets thanks to craftsmanship of [Michael Overstreet](#), aka [I-Bioloid at Thingverse](#). Mike's mesh files were used in the URDF model for the original TurtleBot arm and with Mike's permission, they have been copied to the directory `rbx2_description/meshes` to be used here as well. To see how they look, run the following launch file:

```
$ rosrun rbt2_description pi_robot_with_pan_tilt_head.launch
```

If RViz is not still running, bring it up now:

```
$ rosrun rviz rviz -d `rospack find rbt2_description`/urdf.rviz
```

The view in RViz should then look something like the following:



4.7.5 Using an Asus Xtion Pro mesh instead of a Kinect on Pi Robot

To use a mesh for an Asus Xtion Pro camera instead of a Kinect on the pan-and-tilt head on Pi Robot, run the following launch file:

```
$ roslaunch rbx2_description pi_robot_with_pan_tilt_head_xtion.launch
```

If RViz is not still running, bring it up now:

```
$ rosrun rviz rviz -d `rospack find rbx2_description`/urdf.rviz
```

If RViz is still running from a previous session, don't forget to toggle the checkbox beside the **RobotModel** display to update the model.

4.8 Adding One or Two Arms

The final component we will add to the robot is a multi-jointed arm made up of revolute joints modeled as Dynamixel servos and brackets. We will use six joints thereby giving the arm six *degrees of freedom* (DOF). This is the minimum number of joints required for a general purpose arm if the goal is to grasp objects in space. This is because a 3-dimensional object requires 6 numbers to specify both its position and orientation. So to place a gripper with the correct position and orientation to grasp an object, the arm needs at least 6 degrees of freedom. (More on this in the chapter on Arm Navigation.)

4.8.1 Placement of the arm(s)

If you plan for your robot to use depth information from its camera to guide its arm and gripper to pick up objects, then keep in mind the fact that the Kinect and Xtion Pro are unable to compute depth within about 0.5 meters (2 feet) of the camera. You will therefore need to place the arm so that the typical target location for the gripper is outside of this blind spot. If the camera is mounted high up on the torso, then a good mount point for the arm might be a couple of feet below that. If you can also include a telescoping joint for the torso or a movable arm like on [Maxwell](#), then your arm will be that much more versatile when it comes to reaching objects on counter tops and tables at different heights.

4.8.2 Modeling the arm

Creating a URDF/Xacro for an arm is similar to what we have already done for the pan-and-tilt head. The only real difference is that we have more joints and the arm assembly is attached to a different location on the torso. To see the result for a one-arm Box Robot, run the following launch file:

```
$ roslaunch rbx2_description box_robot_with_arm.launch
```

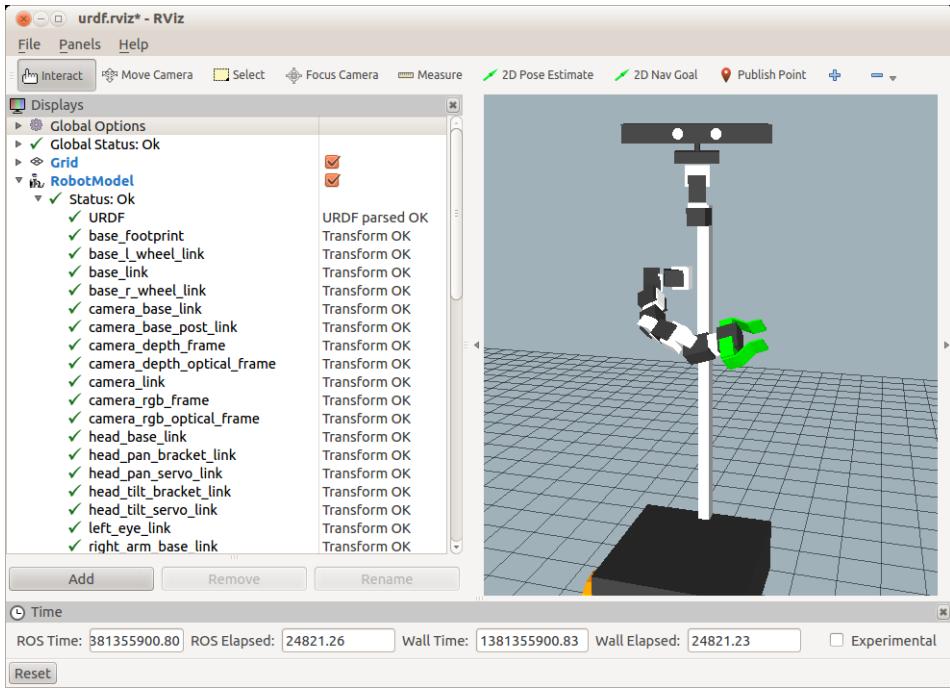
If `RViz` is not still running, bring it up now:

```
$ rosrun rviz rviz -d `rospack find rbx2_description`/urdf.rviz
```

If `RViz` is still running, toggle the **Robot Display** checkbox to refresh the model. The joint state GUI should now appear as follows:



And the view in `RViz` should appear something like the following:



Here we have positioned the joints of the arm as well as the gripper using the slider controls in the joint state GUI.

To conserve space, we won't list out the entire URDF/Xacro files for the arm. The joint definitions are nearly the same as with the pan-and-tilt head covered earlier. The key files are as follows:

- the launch file [box_robot_with_arm.launch](#) loads the Xacro file [box_robot_with_arm.xacro](#) from the `urdf/box_robot` subdirectory
- the `box_robot_with_arm.xacro` file in turn includes Xacro files for the base, torso, pan-and-tilt head, arm, and gripper
- the arm itself is defined in the file [box_arm.urdf.xacro](#)
- the gripper is defined in the file [box_gripper.urdf.xacro](#)

In the file `box_robot_with_arm.xacro` we attach the arm to the torso and the gripper to the arm with the following two code blocks:

```
<!-- Attach the right arm -->
<arm side="right" reflect="-1" parent="torso" color="White">
```

```

<origin xyz="${arm_offset_x} ${arm_offset_y} ${arm_offset_z}" rpy="0 0 ${PI/2}" />
</arm>

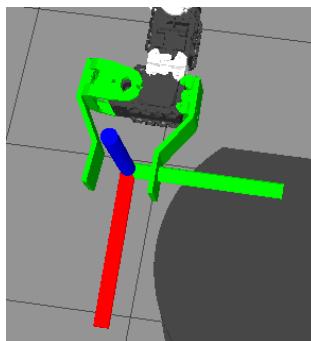
<!-- Attach the right gripper -->
<gripper side="right" reflect="-1" parent="right_arm_gripper_attach"
color="Green">
  <origin xyz="0 0 0" rpy="0 0 0" />
</gripper>
```

The key parameters to note here are the `side` and `reflect` parameters. For the right arm, we set the `side` parameter to "right" and the `reflect` parameter to "-1". These parameters are then passed to macros in the `box_arm.urdf.xacro` macro to give each joint a unique name (e.g. `right_arm_shoulder_pan_joint` versus `left_arm_shoulder_pan_joint`) and (optionally) reflect any displacements from right to left. (As it turns out, our current models do not require the `reflect` parameter but is a good idea to know about it in case it is needed for a different model.) As we shall see in the next section, adding a second arm can be done using the same `box_arm.urdf.xacro` file with parameters `side="left"` and `reflect="1"`.

Don't forget to check the rotations of the arm joints using the method described section 4.7.3. If you find that a particular joint rotates in the wrong direction, simply change the sign of the axis component in the arm's URDF/Xacro file.

4.8.3 Adding a gripper frame for planning

The arm model defined above includes a model for a simple gripper which in turn is defined in the file `box_gripper.urdf.xacro`. This particular gripper has one fixed finger plate and one movable finger controlled by a single servo. When it comes to grasping an object later in the book, we will want to know where within the gripper is the best place to actually do the grasping. For this particular gripper, a good grasping point is located in between the two fingers at the midpoint of the two parallel sections of the finger plates as shown below:



The red, green and blue axes are centered at the desired grasping point and are oriented with the red x-axis parallel to the fingers, the blue z-axis pointing upward from the plane of the gripper, and the green y-axis pointing to the gripper's left.

This gripper frame is defined in the gripper URDF model ([box_gripper.urdf.xacro](#)) by the following block:

```
<!-- Planning link and joint for the right gripper -->
<joint name="${side}_gripper_joint" type="fixed">
  <origin xyz="0.05 0.0 -0.0375" rpy="${PI/2} 0 0"/>
  <axis xyz="0 0 1" />
  <parent link="${side}_gripper_static_finger_link"/>
  <child link="${side}_gripper_link"/>
</joint>

<link name="${side}_gripper_link">
  <visual>
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <geometry>
      <box size="0.001 0.0005 0.0005"/>
    </geometry>
  </visual>
</link>
```

The link itself is defined as a very small box called `right_gripper_link` and the location of this small box is defined by the `right_gripper_joint` which displaces and rotates the origin of the link relative to the parent (`right_gripper_static_finger_link` in this case). The result is a gripper frame located and oriented in a good place for grasping. We will use this gripper frame in the chapter on arm navigation when it comes to picking up objects.

NOTE: Using a virtual gripper frame like this is optional and not a standard ROS practice. It is more common to use the last arm link (e.g wrist roll or wrist flex) attached to the gripper as the reference frame for planning. That way one can swap out different grippers without having much of an effect on existing planning software. However, adding a virtual planning frame can make it easier to visualize and describe grasping goals relative to the gripper.

4.8.4 Adding a second arm

Adding a second arm is almost as easy as adding one arm. The URDF/Xacro file for the arm ([box_arm.urdf.xacro](#)) uses the parameters `side` and `reflect` to determine which side and orientation to place the arm on the torso.

The Xacro file [box_robot_with_two_arms.xacro](#) does the trick by including the following pair of macro blocks, one for the right arm and one for the left:

```
<!-- Attach the right arm -->
<arm side="right" reflect="-1" parent="torso" color="White">
  <origin xyz="${arm_offset_x} ${arm_offset_y} ${arm_offset_z}" rpy="0 0 ${-PI/2}" />
</arm>

<!-- Attach the right gripper -->
<gripper side="right" reflect="-1" parent="right_arm_gripper_attach" color="Green">
  <origin xyz="0 0 0" rpy="0 0 0" />
</gripper>

<!-- Attach the left arm -->
<arm side="left" reflect="1" parent="torso" color="White">
  <origin xyz="${arm_offset_x} ${-arm_offset_y} ${arm_offset_z}" rpy="0 0 ${-PI/2}" />
</arm>

<!-- Attach the left gripper -->
<gripper side="left" reflect="1" parent="left_arm_gripper_attach" color="Green">
  <origin xyz="0 0 0" rpy="0 0 0" />
</gripper>
```

Note how the two arms and grippers use the same macros `<arm>` and `<gripper>` but with opposite values for the parameters `side` and `reflect`. The `side` parameter gives the links and joints of each arm different prefixes (e.g.

`right_arm_shoulder_lift_joint` versus `left_arm_shoulder_lift_joint`) and the `reflect` parameter can be used to mirror displacements or rotation directions although this is not necessary in the case of the current arm models. As for how each arm is mounted, note that the y-offset of the left arm is the opposite sign of the right arm and it is rotated around the z-axis in the by 90 degrees in the opposite direction.

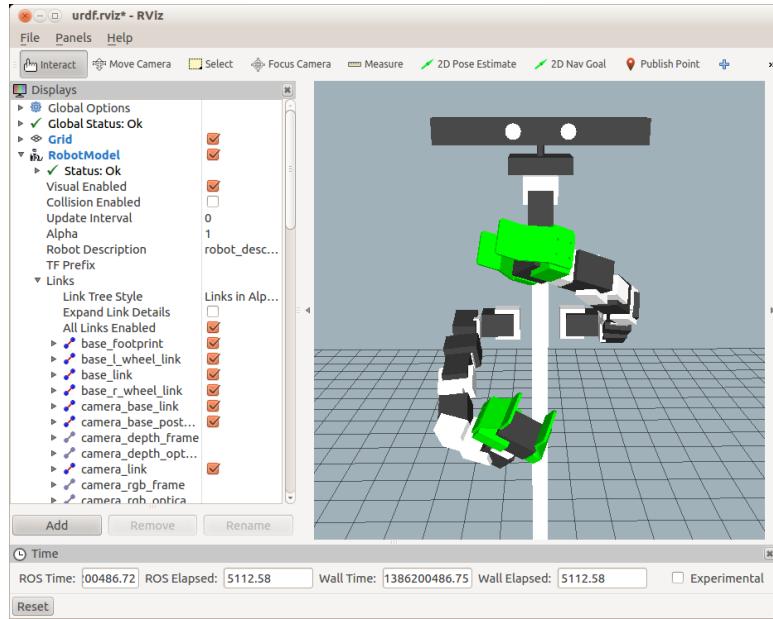
To view the two-armed Box Robot, run the launch file:

```
$ rosrun roslaunch rbx2_description box_robot_with_two_arms.launch
```

If RViz is not still running, bring it up now:

```
$ rosrun rviz rviz -d `rospack find rbx2_description`/urdf.rviz
```

After toggling the **RobotModel** display in RViz and adjusting the joint slider controls, the model should look something like the following:



4.8.5 Using meshes for the arm servos and brackets

Fortunately for us, a number of people have already made STL meshes for Dynamixel servos and brackets. So we can add some detail to our robot arm by using meshes instead of boxes. This has been done for the model of Pi Robot using the following files:

- the launch file [pi_robot_with_arm.launch](#) loads the Xacro file [pi_robot_with_arm.xacro](#) from the urdf/pi_robot subdirectory
- the [pi_tobot_with_arm.xacro](#) file in turn includes Xacro files for the base, torso, pan-and-tilt head, arm, and gripper
- the arm itself is defined in the file [pi_arm.urdf.xacro](#)
- the gripper is defined in the file [pi_gripper.urdf.xacro](#)

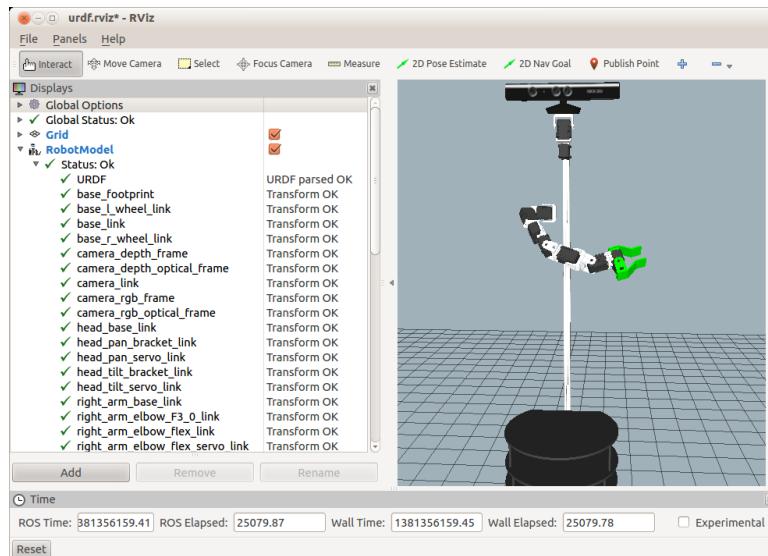
To see the resulting model, terminate any currently running URDF launch files and run:

```
$ rosrun rbx2_description pi_robot_with_arm.launch
```

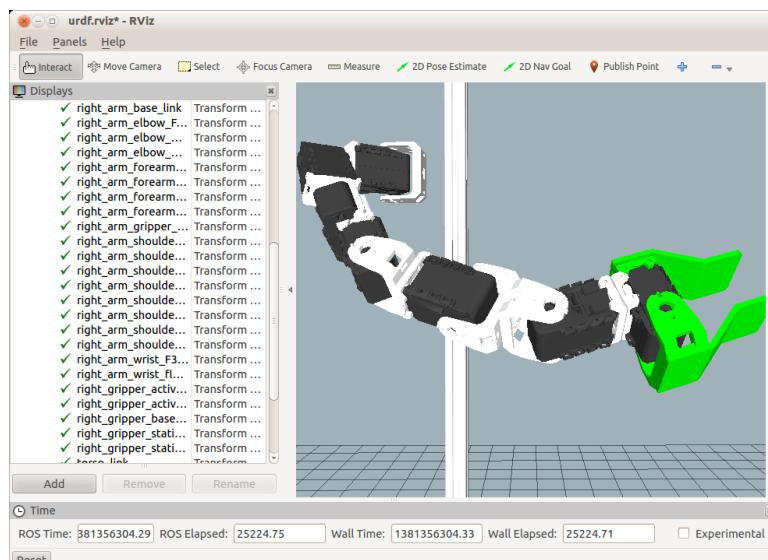
If `RViz` is not still running, bring it up now:

```
$ rosrun rviz rviz -d `rospack find rbx2_description`/urdf.rviz
```

(If RViz is still running from a previous session, toggle the checkbox beside the **RobotModel** display to reload the model.) The image should look like the following:



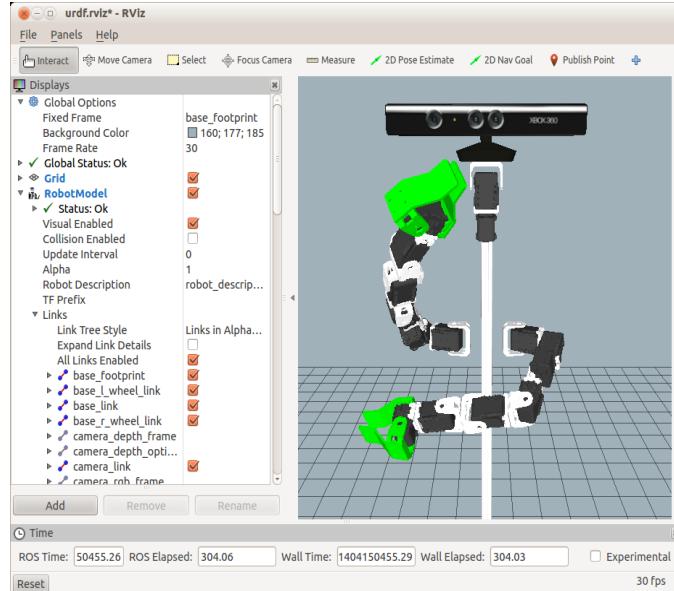
And a close-up of the Dynamixel mesh components is shown below:



You can also view a two-armed version of Pi, using the launch file:

```
$ roslaunch rbx2_description pi_robot_with_two_arms.launch
```

Which should result in the following model in RViz:



4.9 Adding a Telescoping Torso to the Box Robot

A number of commercial robots such as the Willow Garage [PR2](#) and the [UBR1](#) use an adjustable height torso that can move vertically to raise or lower the robot. This increases the reachability of the arm(s) and allows the head camera to be positioned over a greater range of heights. (Michael Ferguson created the hobby level [Maxwell](#) with a customized linear actuator that allows the single arm to move vertically. The head and torso actually remain at a fixed height.)

It is relatively straightforward to add a linear joint to the model of our robot's torso so that we can experiment with adjusting the robot's height on the fly. To try it out with our box robot, run the launch file:

```
$ roslaunch rbx2_description box_robot_tele_torso_with_arm.launch
```

If RViz is not still running, bring it up now:

```
$ rosrun rviz rviz -d `rospack find rbx2_description`/urdf.rviz
```

(If RViz is still running from a previous session, toggle the checkbox beside the **RobotModel** display to reload the model.)

Note that the joint control panel now has a slider for the torso joint. Moving this control will adjust the torso height up and down.

The URDF that makes this happen can be found in the file [tele_torso.urdf.xacro](#) in the directory `rbx2_description/urdf/box_robot`. The key element is the addition of a prismatic (linear) joint that splits the torso into upper and lower segments like this:

```
<joint name="${name}_joint" type="prismatic">
  <parent link="lower_${name}_link"/>
  <child link="upper_${name}_link"/>
  <axis xyz="0 0 1" />
  <origin xyz="0 0 0.1" rpy="0 0 0"/>
  <limit lower="-0.15" upper="0.15" velocity="0.1" effort="1.0" />
</joint>
```

(ROS uses the keyword "prismatic" to indicate a linear joint.) We define the axis to be in the positive *z* direction and we define the origin to have a 10 cm (0.1 meter) offset so that the starting position places the robot at a neutral height. We then give the joint a 15 cm range in both the up and down directions using the `lower` and `upper` parameters.

4.10 Adding a Telescoping Torso to Pi Robot

We can use the same technique with Pi Robot using meshes for the torso rather than boxes. You can try it out using the launch file:

```
$ roslaunch rbx2_description pi_robot_tele_torso_with_arm.launch
```

The URDF model for the telescoping torso can be found in the file [pi_tele_torso.urdf.xacro](#) found in the directory `rbx2_description/urdf/pi_robot`. The code is nearly identical to the box robot as described in the previous section.

4.11 A Tabletop One-Arm Pi Robot

Sometimes it is nice to have a robot, or at least a model of a robot, that just sits on a tabletop and enables you to focus on the head and arm(s). You can find a model of just

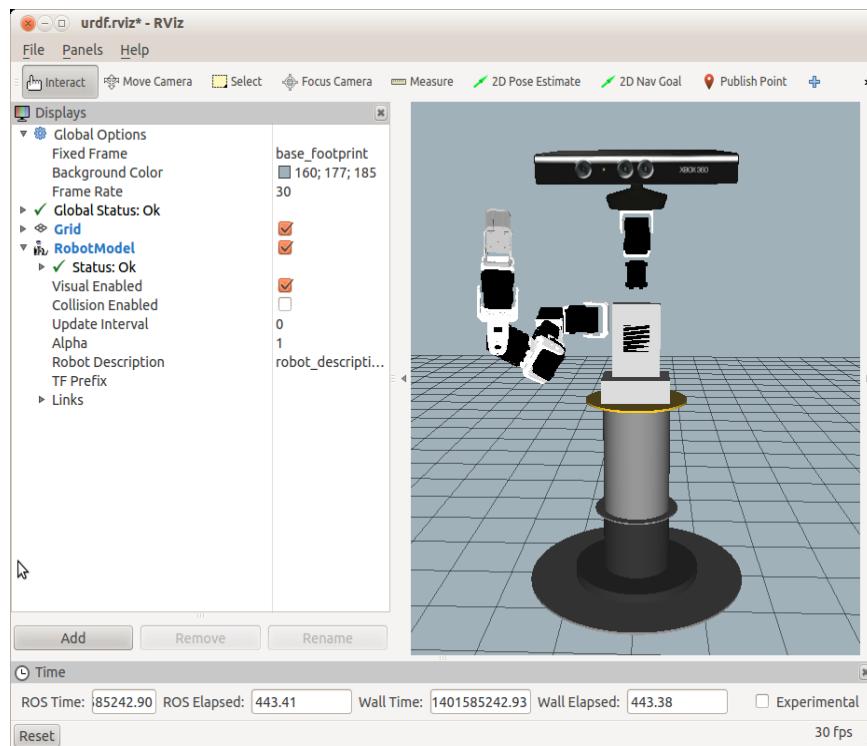
such a robot in the `rbx2_description/urdf/pedestal_pi` directory. There are two versions of the robot, one with a gripper ([pedestal_pi_with_gripper.xacro](#)) and one with a simple paddle-like hand with no moving fingers ([pedestal_pi_no_gripper.xacro](#)). To view the model without the gripper, run the command:

```
$ rosrun rbt2_description pedestal_pi_no_gripper.launch
```

If RViz is not still running, bring it up now:

```
$ rosrun rviz rviz -d `rospack find rbt2_description`/urdf.rviz
```

(If RViz is still running from a previous session, toggle the checkbox beside the **RobotModel** display to reload the model.) The image should look like this:



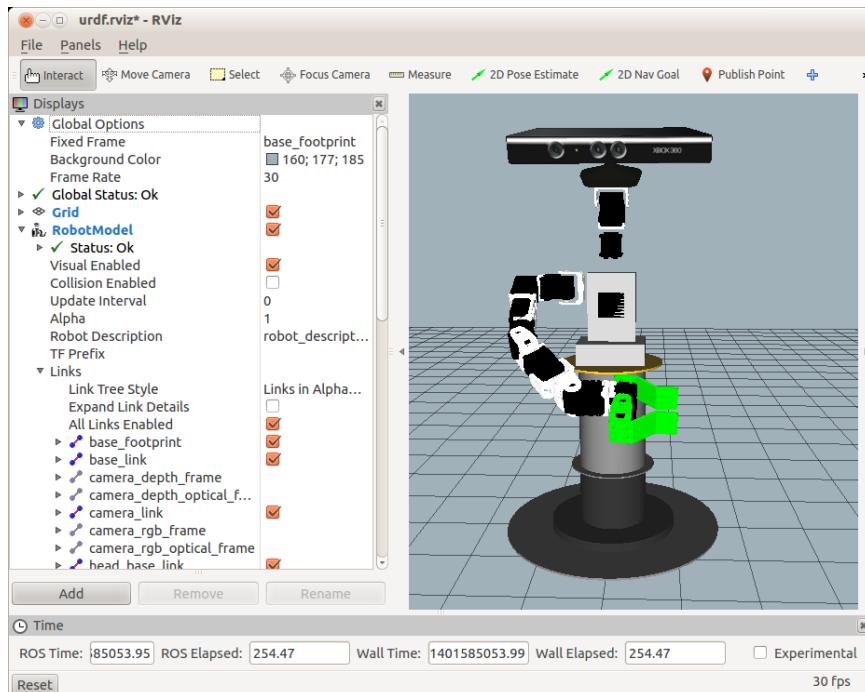
To load the model with the gripper, terminate the earlier launch file and run the command:

```
$ rosrun rviz rviz -d `rospack find rbx2_description`/urdf.rviz
```

If RViz is not still running, bring it up now:

```
$ rosrun rviz rviz -d `rospack find rbx2_description`/urdf.rviz
```

(If RViz is still running from a previous session, toggle the checkbox beside the **RobotModel** display to reload the model.) The image should look like this:



It is instructive to note that the very same URDF/Xacro files for the arm and gripper are used with this "Pedestal Pi" model as with the mobile version of Pi Robot. We are simply attaching the arm and gripper to different base models for the two robots.

4.12 Testing your Model with the ArbotiX Simulator

In *Volume 1* we used the [arbotiX python](#) package to provide a fake base controller that allowed us to control a simulated TurtleBot and view the results in RViz. The complete [arbotiX package](#) also includes controllers for Dynamixel servos, multi-jointed arms, and grippers with various geometries. These controllers can also be run in

fake mode which enables us to test a robot's arm(s) or pan-and-tilt head while monitoring the motions in `RViz`.

The `rbx2_bringup` package includes a number of launch files for running either the Box Robot or Pi Robot using the `arbotix` controllers. We will examine the launch files and configuration files in detail in the next chapter. For now, let's simply try it out with our robot models.

If you haven't already installed the `arbotix` package, do it now with the following command:

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

4.12.1 A fake Box Robot

To launch the ArbotiX simulator using the one-arm Box Robot, use the launch file `fake_box_robot_with_gripper.launch` in the `rbx2_bringup/launch`:

```
$ roslaunch rbx2_bringup fake_box_robot_with_gripper.launch
```

The output on the screen should look like something like this:

```
process[arbotix-1]: started with pid [31013]
process[right_gripper_controller-2]: started with pid [31016]
process[robot_state_publisher-3]: started with pid [31022]
[INFO] [WallTime: 1424531572.432293] ArbotiX being simulated.
[INFO] [WallTime: 1424531573.023166] Started FollowController
(right_arm_controller). Joints: ['right_arm_shoulder_pan_joint',
'right_arm_shoulder_lift_joint', 'right_arm_shoulder_roll_joint',
'right_arm_elbow_flex_joint', 'right_arm_forearm_flex_joint',
'right_arm_wrist_flex_joint'] on C1
[INFO] [WallTime: 1424531573.101172] Started DiffController
(base_controller). Geometry: 0.26m wide, 4100.0 ticks/m.
```

Here we see that we have started three nodes: the main `arbotix` driver, a driver for the right gripper, and the `robot_state_publisher`. We also see `INFO` lines for the two joint trajectory controllers (called `FollowControllers` in the `arbotix` package) one for the right arm and one for the gripper) and the `DiffController` for the differential drive base. We will postpone discussion of these nodes and controllers until the chapter on MoveIt! and Arm Navigation. For now, let us simply verify that we can move the joints in an appropriate manner.

If `RViz` is still running from a previous session, terminate it now and bring it up with the `sim.rviz` config file as follows:

```
$ rosrun rviz rviz -d `rospack find rbx2_bringup`/sim.rviz
```

Note that under the **Global Options** category, we have set the **Fixed Frame** to `/odom`. Recall from *Volume 1* that we need to be in the `/odom` frame in order to see the robot move when publishing `Twist` commands on the `/cmd_vel` topic.

Now open up a new terminal and bring up the ArbotiX GUI utility:

```
$ arbotix_gui
```

A new window should pop up that looks like this:



Use the simulated track pad on the left to drive the robot around in `RViz` by grabbing the little red dot and dragging it around. (If the robot does not appear to move, double-check that you have set the **Fixed Frame** to `/odom` under the **Global Options** category in `RViz`.) To move a joint, click the checkbox beside a servo name then use the slide control to rotate it.

The `arbotix_gui` is a nice way to test the basic functionality of your robot. Moving the red dot on the track pad publishes `Twist` messages on the `/cmd_vel` topic. If the base controller parameters are set properly, the fake robot should move in the expected manner.

Similarly, the servo slider controls can be used to test the motion of individual joints. These controls publish a `Float64` message representing the joint position in radians on the topic `/joint_name/command`. For example, the slider labeled

`right_arm_shoulder_lift` publishes its position value on the `/right_arm_shoulder_lift_joint/command` topic. If you move the slider, the arm should move up or down at the shoulder joint.

Since the `arbotix` nodes are true controllers, unlike the joint state publisher that we have been using in this chapter up until now, we can publish commands on the relevant control topics to move the joints. In *Volume 1* we already saw how to move the fake robot's base by publishing `Twist` commands on the `/cmd_vel` topic. Now we can also move the joints in a similar fashion.

Terminate the `arbotix_gui` if you still have it running, then try the following commands to move the head and arm.

For example to pan the head 90 degrees (1.57 radians) to the left, run the command:

```
$ rostopic pub -1 /head_pan_joint/command std_msgs/Float64 -- 1.57
```

To lift the arm to a horizontal position using the `right_arm_shoulder_lift_joint`, try the command:

```
$ rostopic pub -1 /right_arm_shoulder_lift_joint/command \
std_msgs/Float64 -- -1.57
```

And to move it back to its resting position:

```
$ rostopic pub -1 /right_arm_shoulder_lift_joint/command \
std_msgs/Float64 -- 0.0
```

In later chapters, we will learn how to control the joints programmatically from within our own nodes.

4.12.2 A fake Pi Robot

To test the one-arm Pi Robot mesh model, terminate the launch file for the fake Box Robot if it is still running, run the `pi_robot_with_gripper.launch` file with the `sim` argument set to `true`:

```
$ rosrun rbt2_bringup pi_robot_with_gripper.launch sim:=true
```

If `RViz` is not still running, bring it up now with the `sim.rviz` config file:

```
$ rosrun rviz rviz -d `rospack find rbt2_bringup`/sim.rviz
```

If RViz was already running, toggle the checkbox beside the **RobotModel** display to make sure the display is updated with the model of Pi Robot.

If the ArbotiX GUI is not still running, bring it up now:

```
$ arbotix_gui
```

Now you can drive Pi Robot's base and control his servos as we did with the Box Robot.

4.13 Creating your own Robot Description Package

You now have all the tools you need to model a robot with almost any configuration. But before you start working on the design of your own robot, it is usually a good idea to create a package to hold all of the files. Traditionally in ROS, the package holding the model for a robot named `mybot` would be called `mybot_description`. To create your robot description package using `catkin`, follow these steps. (If you haven't already set up your `catkin` workspace, see *Volume 1 Chapter 4* for instructions or refer to the [online tutorial](#) on the ROS Wiki.)

```
$ cd ~/catkin_ws/src  
$ catkin_create_pkg mybot_description roscpp rospy urdf
```

Then create a few standard subdirectories:

```
$ cd mybot_description  
$ mkdir urdf  
$ mkdir meshes  
$ mkdir launch
```

You will now need to edit the `package.xml` file to include your name and email address as the package maintainer as well as the license type. You can edit the `description` field as you like.

To make sure everything is OK, run `catkin_make`:

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

And to ensure your new package is added to your `ROS_PACKAGE_PATH` for the current terminal session, run:

```
$ rospack profile
```

4.13.1 Copying files from the rbx2_description package

If you want to start with some of the URDF/Xacro files from the `rbx2_description` package and then make modifications, copy them into your own package directory under the appropriate sub-directory. Note however that the Xacro files in the `rbx2_description` package refer to other files in that same package. So you will have to edit the package name and probably the path and file name to match your own filenames. For example, the following `<xacro:include>` line is used in the file `pi_robot_with_arm.xacro` in the `rbx2_description/urdf/pi_robot` directory:

```
<xacro:include filename="$(find  
rbx2_description)/urdf/pi_robot/pi_base.urdf.xacro" />
```

This line pulls in Pi's base model from the file `pi_base.urdf.xacro` found in the same package directory. Simply edit this line in your copy of the file to match your package name and the location of your robot's base file.

4.13.2 Creating a test launch file

To test your URDF/Xacro model, copy and paste the following lines into a launch file called `test_urdf.launch` (or whatever you like) and place it in the `launch` subdirectory of your package directory:

```
<launch>  
    <!-- Load the URDF/Xacro model of your robot -->  
    <arg name="urdf_file" default="$(find xacro)/xacro.py '$(find  
mybot_description)/urdf/mybot.xacro'" />  
  
    <param name="robot_description" command="$(arg urdf_file)" />  
  
    <!-- Publish the robot state -->  
    <node name="robot_state_publisher" pkg="robot_state_publisher"  
type="state_publisher">  
        <param name="publish_frequency" value="20.0"/>  
    </node>  
  
    <!-- Provide simulated control of the robot joint angles -->  
    <node name="joint_state_publisher" pkg="joint_state_publisher"  
type="joint_state_publisher">  
        <param name="use_gui" value="True" />  
        <param name="rate" value="20.0"/>  
    </node>  
</launch>
```

Be sure to change the name of the package and the `.xacro` file highlighted above in bold to match your package name and robot model file.

You can then launch this file any time you want to test your model:

```
$ rosrun mybot_description test_urdf.launch
```

To view the model in RViz, use the config file in the `rbx2_description` package as we have done in the rest of this chapter:

```
$ rosrun rviz rviz -d `rospack find rbx2_description`/urdf.rviz
```

Alternatively, copy the config file into your new package directory:

```
$ roscd mybot_description  
$ rosclp rbx2_description urdf.rviz .
```

You can then run RViz with your own config file and any changes you make to the settings will be saved to your copy:

```
$ rosrun rviz rviz -d `rospack find mybot_description`/urdf.rviz
```

5. CONTROLLING DYNAMIXEL SERVOS: TAKE 2

Readers of *Volume 1* will recall that we used the [dynamixel_motor](#) package to control a Dynamixel pan-and-tilt head. At that time, we had a choice between using the `dynamixel_motor` package or the [arbotix](#) package, and we still do. Both packages provide robust drivers for Dynamixel servos and the primary reason we chose the `dynamixel_motor` package in *Volume 1* was that we needed servo speed control for head tracking that was missing in the `arbotix` package at the time. Since then, Michael Ferguson has added the speed control function to the `arbotix` package. Furthermore, when it comes to arm navigation, the `arbotix` package offers us a number of additional advantages:

- The `arbotix` package includes gripper controllers that are not available in the `dynamixel_motor` package.
- The `arbotix` package provides a fake mode that uses essentially the same configuration file used with real servos. This allows us to test the operation of our scripts and nodes using `RViz` as a kind of fake simulator before trying them out on a real robot.
- The `arbotix` package can be used with either the [ArbotiX controller](#) or the [USB2Dynamixel controller](#) while the `dynamixel_motor` package only runs with the USB2Dynamixel device.
- As we know from *Volume 1*, the `arbotix` package also includes a base controller for a differential drive robot.
- The `arbotix` package is currently more actively developed than the `dynamixel_motor` package.

5.1 Installing the ArbotiX Packages

To install the `arbotix` packages, run the command:

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

That's all there is to it.

5.2 Launching the ArbotiX Nodes

At the end of the chapter on URDF models, we briefly explained how we can use the [arbotix](#) package in fake mode to test the function of our robot's joints. In this chapter we will learn how to set up the ArbotiX configuration file and create a launch file that will bring up the controllers for the arm and head joints as well as the gripper. We can then use these controllers later in the book for head tracking and arm navigation, both in simulation and with real Dynamixel servos.

The `rbx2 Bringup` package includes launch files for different versions of Pi Robot.

The launch file for the one-arm version of Pi Robot is called

`pi_robot_with_gripper.launch`. This file brings up the `arbotix` base controller, individual controllers for each joint (servo), trajectory controllers for groups of joints such as the arm, and a controller for Pi's single-servo style gripper.

Link to source: [pi_robot_with_gripper.launch](#)

```
1 <launch>
2   <!-- Make sure we are not using simulated time -->
3   <param name="/use_sim_time" value="false" />
4
5   <!-- Launch the arbotix driver in fake mode by default -->
6   <arg name="sim" default="true" />
7
8   <!-- If a real controller, look on /dev/ttyUSB0 by default -->
9   <arg name="port" default="/dev/ttyUSB0" />
10
11  <!-- Load the URDF/Xacro model of our robot -->
12  <param name="robot_description" command="$(find xacro)/xacro.py '$(find
rbx2_description)/urdf/pi_robot/pi_robot_with_gripper.xacro'" />
13
14  <node name="arbotix" pkg="arbotix_python" type="arbotix_driver"
clear_params="true" output="screen">
15    <rosparam file="$(find
rbx2_dynamixels)/config/arbotix/pi_robot_with_gripper.yaml" command="load" />
16    <param name="port" value="$(arg port)" />
17    <param name="sim" value="$(arg sim)" />
18  </node>
19
20  <!-- Run a separate controller for the one sided gripper -->
21  <node name="right_gripper_controller" pkg="arbotix_controllers"
type="gripper_controller" output="screen">
22    <rosparam>
23      model: singlesided
24      invert: true
25      center: 0.0
26      pad_width: 0.01
27      finger_length: 0.1653
28      joint: right_gripper_finger_joint
29    </rosparam>
30  </node>
```

```

31      <!-- Publish the robot state -->
32      <node name="robot_state_publisher" pkg="robot_state_publisher"
33      type="state_publisher">
34          <param name="publish_frequency" type="double" value="20.0" />
35      </node>
36
37      <!-- Start all servos in a relaxed state -->
38      <node pkg="rbx2_dynamixels" type="arbotix_relax_all_servos.py"
39      name="relax_all_servos" unless="$(arg sim)" />
40
41      <!-- Load diagnostics -->
42      <node pkg="diagnostic_aggregator" type="aggregator_node"
43      name="diagnostic_aggregator" clear_params="true" unless="$(arg sim)">
44          <rosparam command="load" file="$(find
45          rbx2_dynamixels)/config/dynamixel_diagnostics.yaml" />
46      </node>
47
48      <node pkg="rqt_robot_monitor" type="rqt_robot_monitor"
49      name="rqt_robot_monitor" unless="$(arg sim)" />
50
51  </launch>

```

Let's break this down line by line.

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

ROS uses the `use_sim_time` parameter when running `bag` files or the Gazebo simulator. It is therefore a good idea to explicitly set this parameter to `false` at the top of a launch file designed for a real controller in case you have inadvertently left it set to `true` from an earlier session.

```
6    <arg name="sim" default="true" />
```

The `arbotix` driver uses a parameter called `sim` to determine whether it runs in fake mode or controls real servos. Here we define an argument by the same name so that we can pass a value of `true` or `false` for the parameter when we run the launch file. This allows us to use the same launch file for running the fake robot or the real robot. We also set a default value of `true` so that if we run the launch file without an argument, it will run in fake mode. So to run the one-arm version Pi Robot in fake mode, we would run the launch file like this:

```
$ roslaunch rbx2_bringup pi_robot_with_gripper.launch sim:=true
```

To connect to the real Pi Robot, we would run the launch file with the `sim` argument set to `false` like this:

```
$ roslaunch rbx2_bringup pi_robot_with_gripper.launch sim:=false
```

Let's now continue with the launch file:

```
9    <arg name="port" default="/dev/ttyUSB0" />
```

Here we define an argument for the serial port that will be used if we are connecting to a real controller. We also set the default to the most likely USB port so that if your controller is on /dev/ttyUSB0, you do not have to supply the port argument when running the launch file. So if Pi Robot is using a USB2Dynamixel on port /dev/ttyUSB0, and we want control real servos rather than running in fake mode, we would run the launch file like this:

```
$ roslaunch rbx2_bringup pi_robot_with_gripper.launch sim:=false
```

where we are using the default port of /dev/ttyUSB0. If the controller is on /dev/ttyUSB1, then we supply the port value explicitly:

```
$ roslaunch rbx2_bringup pi_robot_with_gripper.launch sim:=false  
port:=/dev/ttyUSB1
```

If your controller is always on USB1, you could modify the launch file or make a copy of the original and substitute /dev/ttyUSB1 for the default. See also the Appendix on configuring plug and play devices under Ubuntu so that your controller is always assigned the same device name.

```
12    <param name="robot_description" command="$(find xacro)/xacro.py '$(find  
rbx2_description)/urdf/pi_robot/pi_robot_with_gripper.xacro'" />
```

By now you should be familiar with this line that loads the URDF model we want to use. In this case, we are loading the model for the one-arm version of Pi Robot including a gripper as defined by the URDF/Xacro file `pi_robot_with_gripper.xacro` in the `rbx2_description` package under the subdirectory `urdf/pi_robot`.

```
14    <node name="arbotix" pkg="arbotix_python" type="arbotix_driver"  
clear_params="true" output="screen">  
15        <rosparam file="$(find  
rbx2_dynamixels)/config/arbotix/pi_robot_with_gripper.yaml" command="load" />  
16        <param name="sim" value="$(arg sim)" />  
17        <param name="port" value="$(arg port)" />  
18    </node>
```

Lines 14–18 bring up the main `arbotix_driver` node located in the [arbotix_python](#) package. The `clear_params="true"` argument in Line 14 deletes any left over parameters in the `/arbotix` namespace that might be set by an earlier session. Line 15 loads the arbotix configuration file for the robot we are using. In this case, the configuration is loaded from the file `pi_robot_with_gripper.yaml` found in the `rbx2_dynamixels` package under the `config/arbotix` subdirectory. We will examine this file in detail in the next section. Lines 16 and 17 set the `sim` and `port` parameters from the arguments defined earlier in the launch file.

```

21   <node name="right_gripper_controller" pkg="arbotix_controllers"
22     type="gripper_controller" output="screen">
23     <rosparam>
24       model: singlesided
25       invert: true
26       center: 0.0
27       pad_width: 0.004
28       finger_length: 0.065
29       joint: right_gripper_finger_joint
30     </rosparam>
31   </node>
```

Here we launch the arbotix gripper controller for Pi's one-sided gripper. The [arbotix_controllers](#) package includes the `gripper_controller` node that can control a number of different gripper types including the one-sided single-servo model like Pi's, dual servo models like those used on [Maxwell](#), or single-servo parallel grippers like the [PhantomX](#) gripper from Trossen Robotics. For Pi's gripper, we set the `model` parameter to `singlesided` as well as the `pad_width` and `finger_length` given in meters. The single joint name must match Pi's gripper URDF model and is set to `right_gripper_finger_joint` since that joint corresponds to the servo that rotates the movable finger plate.

```

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

Next we run the `robot_state_publisher` node that takes care of mapping the robot's URDF model and joint states into the corresponding `tf` tree.

```

38   <node pkg="rbx2_dynamixels" type="arbotix_relax_all_servos.py"
39     name="relax_all_servos" unless="$(arg sim)" />
```

When using real servos, it is a good idea to initialize each servo in a relaxed state so that the arm or head can be positioned by hand if necessary. The script called `arbotix_relax_all_servos.py` does the job by turning off the torque for each joint. It also sets the initial movement speed to a moderate value so that sending a position command to a servo won't result in a surprisingly fast motion. The `arbotix_relax_all_servos.py` script can be found in the `rbx2_dynamixels/script` directory and will be examined more closely later in the chapter.

Note how we make use of the `unless` keyword inside the `<node>` tag in Line 38. This tells `roslaunch` not to run this node if the `sim` parameter is `true`; i.e. if we are running in fake mode. (There is no need to relax a fake servo!)

```
41   <node pkg="diagnostic_aggregator" type="aggregator_node"
42     name="diagnostic_aggregator" clear_params="true" unless="$(arg sim)">
43       <rosparam command="load" file="$(find
44         rbx2_dynamixels)/config/dynamixel_diagnostics.yaml" />
45     </node>
```

The `arbotix` node publishes servo diagnostics automatically so here we fire up the `diagnostic_aggregator` that will enable us to keep an eye on servo loads and temperatures. Our diagnostics configuration is stored in the `dynamixel_diagnostics.yaml` file located in the `rbx2_dynamixels/config` directory. (ROS diagnostics is explored fully in the next chapter.) This configuration file instructs the aggregator to summarize diagnostic information for joints and controllers. Note that we again use the `unless` keyword inside the `<node>` tag so we don't run the node in fake mode.

```
45   <node pkg="rqt_robot_monitor" type="rqt_robot_monitor"
46     name="rqt_robot_monitor" unless="$(arg sim)" />
```

Finally, we run the `rqt_robot_monitor` (unless `sim=true`) which brings up the diagnostic GUI allowing us to visually monitor the status of the servos. More on `rqt_robot_monitor` in the next chapter on ROS diagnostics.

5.3 The ArbotiX Configuration File

The launch file above loads the configuration file `pi_robot_with_gripper.yaml` located in the directory `rbx2_dynamixels/config/arbotix`. Let's take a look at that file now. (Clicking the link below will bring up a nicely formatted view of the file that might be easier to read than the listing included here.)

Link to source: [pi_robot_with_gripper.yaml](#)

```

1 port: /dev/ttyUSB0
2 baud: 1000000
3 rate: 100
4 sync_write: True
5 sync_read: False
6 read_rate: 10
7 write_rate: 10
8
9 joints: {
10     head_pan_joint: {id: 1, neutral: 512, min_angle: -145, max_angle: 145},
11     head_tilt_joint: {id: 2, neutral: 512, min_angle: -90, max_angle: 90},
12     right_arm_shoulder_roll_joint: {id: 3, neutral: 512, invert: True},
13     right_arm_elbow_flex_joint: {id: 4, neutral: 512, min_angle: -90,
max_angle: 90},
14     right_arm_forearm_flex_joint: {id: 5, neutral: 512, min_angle: -90,
max_angle: 90},
15     right_arm_wrist_flex_joint: {id: 6, neutral: 512, min_angle: -90,
max_angle: 90},
16     right_arm_shoulder_pan_joint: {id: 7, ticks: 4096, neutral: 2048,
min_angle: -100, max_angle: 100},
17     right_arm_shoulder_lift_joint: {id: 8, ticks: 4096, neutral: 1024, invert:
True},
18     right_gripper_finger_joint: {id: 9, neutral: 512, min_angle: -20,
max_angle: 25}
19 }
20
21 controllers: {
22     base_controller: {type: diff_controller, base_frame_id: base_footprint,
base_width: 0.26, ticks_meter: 4100, Kp: 12, Kd: 12, Ki: 0, Ko: 50, accel_limit:
1.0 },
23     right_arm_controller: {onboard: False, action_name:
follow_joint_trajectory, type: follow_controller, joints:
[right_arm_shoulder_pan_joint, right_arm_shoulder_lift_joint,
right_arm_shoulder_roll_joint, right_arm_elbow_flex_joint,
right_arm_forearm_flex_joint, right_arm_wrist_flex_joint]},
24     head_controller: {onboard: False, action_name:
head_controller/follow_joint_trajectory, type: follow_controller, joints:
[head_pan_joint, head_tilt_joint]}
25 }
```

The official description of all available arbotix parameters can be found on the [arbotix_python Wiki page](#). So let's focus on the parameter values that we have set in our current configuration file.

```

1 port: /dev/ttyUSB0
2 baud: 1000000
3 rate: 100
4 sync_write: True
5 sync_read: False
6 read_rate: 10
7 write_rate: 10
```

The `port` and `baud` parameters are only relevant if using a real controller and servos will be ignored in fake mode . Here is a brief description of the parameters listed in Lines 1–7:

- `port` (default: `/dev/ttyUSB0`) – refers to the serial port on which the controller is connected and it is likely to be `/dev/ttyUSB0`, `/dev/ttyUSB1`, etc. Recall that we can override the port setting on the command line when running our launch file.
- `baud` (default: 115200) – the baud rate of the serial port connected to the controller. If you are using a USB2Dynamixel controller, this parameter has to be set to 1000000. If you are using an ArbotiX controller, the default baud rate is 115200.
- `rate` (default: 100) – how fast (in Hz) to run the main driver loop. The default value of 100 works well, especially when it comes to generating joint trajectories as we will do later on. The reason is that the arbotix controllers interpolate joint positions when moving the servos and a high rate here results in a finer degree of interpolation and therefore smoother servo motion.
- `sync_write` (default: `True`) – if set to `True`, enables the controller to send data to all servos simultaneously which is usually desirable and is in fact the default value for this parameter.
- `sync_read` (default: `True`) – if you are using a USB2Dynamixel controller, the parameter must be set to `False`. The default value of `True` will only work with the ArbotiX controller.
- `read_rate` (default: 10) – how frequently (Hz) data is read from the Dynamixels.
- `write_rate` (default: 10) – how frequently (Hz) data is written to the Dynamixels.

```
9 joints: {
10     head_pan_joint: {id: 1, neutral: 512, min_angle: -145, max_angle: 145},
11     head_tilt_joint: {id: 2, neutral: 512, min_angle: -90, max_angle: 90},
12     right_arm_shoulder_roll_joint: {id: 3, neutral: 512, invert: True},
13     right_arm_elbow_flex_joint: {id: 4, neutral: 512, min_angle: -90,
max_angle: 90},
14     right_arm_forearm_flex_joint: {id: 5, neutral: 512, min_angle: -90,
max_angle: 90},
```

```

15     right_arm_wrist_flex_joint: {id: 6, neutral: 512, min_angle: -90,
max_angle: 90},
16     right_arm_shoulder_pan_joint: {id: 7, ticks: 4096, neutral: 2048,
min_angle: -100, max_angle: 100},
17     right_arm_shoulder_lift_joint: {id: 8, ticks: 4096, neutral: 1024, invert:
True},
18     right_gripper_finger_joint: {id: 9, neutral: 512, min_angle: -20,
max_angle: 25}
19 }

```

In Lines 9–19 we set the parameters for our servos by using the `joints` parameter which is a dictionary of joint names together with a separate parameter dictionary for each servo. (More on these parameters below.) Note that the joint names must be the same here as they are in the URDF model for your robot. You can use the above configuration file as a template for your own robot, but be sure to edit the names to match your robot's URDF/Xacro model.

IMPORTANT NOTE: The AX and RX series of Dynamixel servos have an 8-bit angular resolution so that position values range from 0-1023 with a mid-point setting of 512. They also have a range of 300 degrees. The EX and MX series have a 12-bit angular resolution so that position values range from 0-4095 with a mid-point value of 2048. These servos have a range of a full 360 degrees. Pi Robot uses a pair of MX-28T servos for his shoulder pan and lift joints and AX-12 servos everywhere else. You will therefore see that the `ticks` and `neutral` parameters for these two joints above refer to the 12-bit range of the MX servos. More details are given below in the parameter summary.

Each joint or servo has its own set of parameters as follows:

- `id` – the Dynamixel hardware ID for this servo
- `ticks` (default: 1024) – the range of position values for this servo. The default value is valid for AX and RX series Dynamixels. For EX and MX models, a value of 4096 must be used in the configuration file.
- `neutral` (default: 512) – the neutral position value is mapped by the arbotix driver into 0 radians when publishing joint states. The default value of 512 is usually appropriate for AX and RX model servos. However, for EX and MX servos, a mid-point of 2048 must be explicitly specified in the configuration file. Depending on how a particular servo connects to the rest of the robot, you will sometimes set a neutral value other than the mid-point of its range. For example, for Pi Robot's shoulder lift joint (which uses an MX-28T servo), the neutral parameter is set to 1024 which represents $\frac{1}{4}$ of a turn. This was done so

that a value of 0 radians corresponds to the arm hanging straight down instead of straight out.

- `range` (default: 300) – the working angular range of the servo. The default value of 300 applies to AX and RX servos but a value of 360 must be explicitly specified in the configuration file for EX and MX models.
- `max_speed` (default: 684) – the maximum speed of the servo in degrees per second. The default value is suitable for an AX-12 servo. For an MX-28 servo, use 702.
- `min_angle` (default: -150) – the minimum angle (in degrees) that the controller will turn the servo. The default value of -150 is half the full range of an AX or RX servo. For EX or MX servos, that value can be as high as -180. Note that you can set smaller values if the rotation of the servo is restricted by the way it is mounted. For example, Pi Robot's elbow joint can rotate only 90 degrees in either direction before its bracket runs into the upper arm. We therefore specify min and max angles parameters of -90 and 90 in the configuration file shown above.
- `max_angle` (default: 150) – The maximum angle (in degrees) that the controller will turn the servo. The default value of 150 is half the full range of an AX or RX servo. For EX or MX servos, that value can be as high as 180. Note that you can set smaller values if the rotation of the servo is restricted by the way it is mounted. For example, Pi Robot's elbow joint can rotate only 90 degrees in either direction before its bracket runs into the upper arm. We therefore specify min and max angles parameters of -90 and 90 in the configuration file shown above.
- `invert` (default: `False`) – some servos will be mounted so that an increasing value of the rotation angle will result in a negative rotation in the ROS-sense; i.e. when using the right-hand rule as we did in the chapter on URDF models. If you find that a particular servo rotates a joint in a direction opposite to the direction you specified in your URDF model, then set the `invert` parameter to `True` in the arbotix config file. Note that we have done this in the configuration file shown above for both the `right_arm_shoulder_roll` joint and the `right_arm_shoulder_lift` joint.

For each joint listed in the configuration file, the arbotix driver launches a joint controller and several ROS services similar to those we used in *Volume 1* with the `dynamixel_motor` package. For example, to control the position (in radians) of the `right_arm_elbow_flex_joint`, we can publish a `Float64` message on the `/right_arm_elbow_flex_joint/command` topic. To set the speed of the head pan

servo, we can use the `/head_pan_joint/set_speed` service. We will say more about these controllers and services in the next section on testing the servos.

Now back to the configuration file.

```
21 controllers: {
22     base_controller: {type: diff_controller, base_frame_id: base_footprint,
base_width: 0.26, ticks_meter: 4100, Kp: 12, Kd: 12, Ki: 0, Ko: 50, accel_limit:
1.0 },
23     right_arm_controller: {onboard: False, action_name:
follow_joint_trajectory, type: follow_controller, joints:
[right_arm_shoulder_pan_joint, right_arm_shoulder_lift_joint,
right_arm_shoulder_roll_joint, right_arm_elbow_flex_joint,
right_arm_forearm_flex_joint, right_arm_wrist_flex_joint]}
24     head_controller: {onboard: False, action_name:
head_controller/follow_joint_trajectory, type: follow_controller, joints:
[head_pan_joint, head_tilt_joint]}
25 }
```

Here we see `controllers` section of the configuration file. In the case of Pi Robot, we are configuring three controllers: the base controller for a differential drive base, a trajectory action controller for the right arm joints, and a second trajectory action controller for the head joints. Let's look at each of these in turn.

For a real robot, the base controller settings appearing in Line 22 above are only relevant if you are using an actual [ArbotiX controller](#) to control a motor driver and a differential drive base. However, we also use the base controller in fake mode when we want to test a mobile robot in the ArbotiX simulator as we did in *Volume 1* when learning about ROS navigation. The official reference for the base parameters can be found on the ArbotiX [diff_controller Wiki page](#). When using a fake robot, the most important parameter here is the `base_frame_id` which we have set to `base_footprint`. If your robot does not use a `base_footprint` frame, change the value here accordingly; e.g. `base_link`.

Next we turn to the joint trajectory controllers for the arm and head. The use of ROS joint trajectories to control multiple joints simultaneously will be covered in detail in the chapter on Arm Navigation. For now we will simply describe the various parameters used in the configuration file. Let's start with the arm controller.

In Line 23 above, we have named the arm controller `right_arm_controller` and it is defined by a dictionary of parameters as shown above. Let's describe each of these in turn:

- `onboard` (default: `True`) – if you are using a USB2Dynamixel controller, this parameter must be set to `False`. The default value of `True` works only with the ArbotiX controller that includes code to do trajectory interpolation in firmware.

- `action_name` (default: `follow_joint_trajectory`) – this parameter defines the namespace for the joint trajectory action controller. In the configuration file above we prepend the controller name to the action name so that the full namespace becomes `right_arm_controller/follow_joint_trajectory`. This enables us to add a left arm at a later time whose controller would then operate in the `left_arm_controller/follow_joint_trajectory` namespace. If this controller is going to be used with MoveIt (as we will in the next several sections), the `action_name` must match the `action_ns` parameter set in the `controllers.yaml` file in the corresponding MoveIt configuration.
- `type` – the `arbotix` node includes two types of controllers: a `diff_controller` for controlling a base as we used in *Volume 1*, and a `follow_controller` for managing joint trajectories. Of course, our robot's right arm controller must be of type `follow_controller`.
- `joints` – the final parameter is the list of joints to be managed by this controller. In the configuration file above, we list the joints of the right arm in the same order in which they appear in the kinematic tree. Note that we do not include the gripper joint since the gripper is managed by its own controller.

The Arbotix `follow_controller` implements a ROS [joint trajectory action server](#) and responds to action goals using the `FollowJointTrajectoryGoal` message type that we will discuss fully in the chapter on Arm Navigation. The controller defined in the configuration file above listens on the topic `/right_arm_controller/follow_joint_trajectory/goal` for trajectory goals.

In Line 24 above we define a second trajectory action controller, this time for the pan-and-tilt servos of the head. Although we won't use the head controller much in this volume, it can be used to send motion goals to both head joints simultaneously.

5.4 Testing the ArbotiX Joint Controllers in Fake Mode

Before trying the `arbotix` controllers with real servos, let's run a few tests in the ArbotiX simulator. First bring up the fake version of Pi Robot as we have done before:

```
$ roslaunch rbx2 Bringup pi_robot_with_gripper.launch sim:=true
```

You should see the following output on the screen:

```

process[arbotix-1]: started with pid [11850]
process[right_gripper_controller-2]: started with pid [11853]
process[robot_state_publisher-3]: started with pid [11859]
[INFO] [WallTime: 1401976945.363225] ArbotiX being simulated.
[INFO] [WallTime: 1401976945.749586] Started FollowController
(right_arm_controller). Joints: ['right_arm_shoulder_pan_joint',
'right_arm_shoulder_lift_joint', 'right_arm_shoulder_roll_joint',
'right_arm_elbow_flex_joint', 'right_arm_forearm_flex_joint',
'right_arm_wrist_flex_joint'] on C1
[INFO] [WallTime: 1401976945.761165] Started FollowController
(head_controller). Joints: ['head_pan_joint', 'head_tilt_joint'] on C2

```

The key items to note are highlighted in bold above: the right gripper controller is started; the ArbotiX controller is running in simulation mode; and the trajectory controllers for the right arm and head have been launched.

Now bring up RViz with the `urdf.rviz` config file from the `rbx2_description` package:

```
$ rosrun rviz rviz -d `rospack find rbx2_description`/urdf.rviz
```

Next, bring up a new terminal and let's try publishing a few simple joint position commands. We first learned about these commands in *Volume 1* in the chapter on controlling Dynamixel servos with ROS. We also ran some similar tests near the end of the chapter on URDF models.

The first command should pan the head to the left (counter-clockwise) through 1.0 radians 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 raising the arm 2.0 radians (about 114 degrees) using the `right_arm_shoulder_lift_joint`:

```
$ rostopic pub -1 /right_arm_shoulder_lift_joint/command \
std_msgs/Float64 -- -2.0
```

Then lower it back down:

```
$ rostopic pub -1 /right_arm_shoulder_lift_joint/command \
  std_msgs/Float64 -- 0.0
```

To disable a servo so that it no longer responds to position commands, use the `enable` service with the argument `false`:

```
$ rosservice call /head_pan_joint/enable false
```

Next, try sending the servo a new position:

```
$ rostopic pub -1 /head_pan_joint/command std_msgs/Float64 -- -1.0
```

Notice how the servo does not respond. To re-enable the servo, run the command:

```
$ rosservice call /head_pan_joint/enable true
```

Then try the position command again:

```
$ rostopic pub -1 /head_pan_joint/command std_msgs/Float64 -- -1.0
```

This time the servo should move. The enable service comes in handy when we want to disable a servo for a period of time to let it cool down or otherwise take it offline for a bit. In the meantime, other nodes can still publish position commands and they will simply be ignored. We will look at two more services, `relax` and `set_speed` in the next section on testing real servos.

We will defer testing arm and head trajectory controllers until the chapter on Arm Navigation after we have learned more about joint trajectories and arm kinematics.

5.5 Testing the Arbotix Joint Controllers with Real Servos

Now that we have things working in the ArbotiX simulator, it is time to try things out with real servos. Assuming you have a robot with an arm or at least a pan-and-tilt head, start by running the launch file for your robot which in turn should load the controller configuration file for your servos.

If your robot does not have an arm but does have a pan-and-tilt head that uses AX-12 servos, you can use the launch file [pi_robot_head_only.launch](#) in the `rbx2_bringup/launch` directory. This launch file uses the configuration file [pi_robot_head_only.yaml](#) in the `rbx2_dynamixels/config/arbotix` directory and connects to two Dynamixel AX-12 servos with hardware IDs 1 and 2 and with joint names `head_pan_joint` and `head_tilt_joint`.

Assuming you have your servos connected to a USB2Dynamixel controller on USB port /dev/ttyUSB0, run the launch file with the `sim` parameter set to `false`:

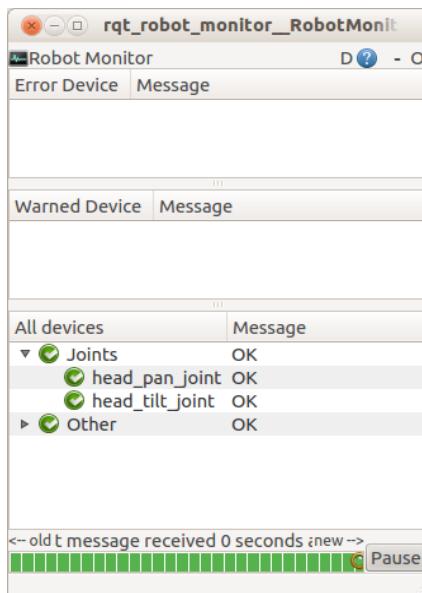
```
$ rosrun rbt2_bringup pi_robot_head_only.launch sim:=false
```

If your USB2Dynamixel controller is connected to a different USB port, you can run the launch file with the `port` argument. For example, if your controller is on USB port /dev/ttyUSB1, use the command:

```
$ rosrun rbt2_bringup pi_robot_head_only.launch sim:=false \
  port:=/dev/ttyUSB1
```

(In the Appendix, we show how to create more descriptive port names like /dev/usb2dynamixel that will work no matter what underlying physical port is used by the OS.)

After a brief delay, you should see the `rqt_robot_monitor` GUI appear that should look like the following image:



If the USB2Dynamixel controller has successfully connected to the two servos, the two joints will appear in the `rqt_robot_monitor` window as shown above. We will explore this monitor in detail in the next chapter on ROS Diagnostics.

You should also find that the servos have been started in the relaxed state so that they can be turned by hand. This is because the launch file above runs the `arbotix_relax_all_servos.py` node.

As with the fake robot, let's begin by trying to move the pan-and-tilt head:

```
$ rostopic pub -1 /head_pan_joint/command std_msgs/Float64 -- 1.0
```

This command should pan the head smoothly to the left (counter-clockwise) through 1.0 radians or about 57 degrees. (If the head rotates clockwise, then your `arbotix`'s configuration file needs to be edited so that the head servo's `invert` parameter is set to `True`. Or, if it was already set to `True` for some reason, set it to `False` instead. Then restart your robot's launch file and try the above command again.)

To change the speed of the head pan servo to 1.5 radians per second, use the `set_speed` service:

```
$ rosservice call /head_pan_joint/set_speed 1.5
```

Then try panning the head back to center at the new speed:

```
$ rostopic pub -1 /head_pan_joint/command std_msgs/Float64 -- 0.0
```

To relax a servo so that you can move it by hand, use the `relax` service:

```
$ rosservice call /head_pan_joint/relax
```

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. Note also that the servo also remembers the last speed setting.

To fully disable a servo so that it relaxes and no longer responds to position commands, use the `enable` service with the argument `false`:

```
$ rosservice call /head_pan_joint/enable false
```

You should now be able to turn the servo by hand. Next, try sending the servo a new position:

```
$ rostopic pub -1 /head_pan_joint/command std_msgs/Float64 -- -1.0
```

This time the servo should not respond. To re-enable the servo, run the command:

```
$ rosservice call /head_pan_joint/enable true
```

Then try the position command again:

```
$ rostopic pub -1 /head_pan_joint/command std_msgs/Float64 -- -1.0
```

Now the servo should move again. The enable service comes in handy when we want to disable a servo for a period of time to let it cool down or otherwise take it offline for a bit. In the meantime, other nodes can still publish position commands and they will simply be ignored.

5.6 Relaxing All Servos

In *Volume 1* we used the script called `relax_all_servos.py` found in the `rbx1_dynamixels` package to turn off the torque on all Dynamixel servos at the same time. That script was written assuming the `dynamixel_motor` package was being used to controller the servos. Now that we are using the `arbotix` package, we need to modify the script to use the appropriate services used by the `arbotix` joint controllers.

The new script is called `arbotix_relax_all_servos.py` and is found in the directory `rbx2_dynamixels/scripts`. To try it out, run the command:

```
$ rosrun rbx2_dynamixels arbotix_relax_all_servos.py
```

When the command is complete, all servos should be movable by hand.

Let's take a quick look at the script since it nicely illustrates how to access the `arbotix` joint services from within a ROS node.

Link to source: [arbotix_relax_all_servos.py](#)

```
1  #!/usr/bin/env python
2
3  import rospy
4  from std_srvs.srv import Empty
5  from arbotix_msgs.srv import SetSpeed
6
7  class Relax():
8      def __init__(self):
```

```

9      rospy.init_node('relax_all_servos')
10
11     # The list of joints is stored in the /arbotix/joints parameter
12     self.joints = rospy.get_param('/arbotix/joints', '')
13
14     # Set a moderate default servo speed
15     default_speed = rospy.get_param('~default_speed', 0.5)
16
17     # A list to hold the relax services
18     relax_services = list()
19
20     # A list to hold the set_speed services
21     set_speed_services = list()
22
23     # Connect to the relax and set_speed service for each joint
24     for joint in self.joints:
25         relax_service = '/' + joint + '/relax'
26         rospy.wait_for_service(relax_service)
27         relax_services.append(rospy.ServiceProxy(relax_service, Empty))
28
29         speed_service = '/' + joint + '/set_speed'
30         rospy.wait_for_service(speed_service)
31         set_speed_services.append(rospy.ServiceProxy(speed_service,
32                                         SetSpeed))
33
34         # Set the servo speed to the default value
35         for set_speed in set_speed_services:
36             set_speed(default_speed)
37
38         # Relax each servo
39         for relax in relax_services:
40             relax()
41
42         # Do it again just to be sure
43         for relax in relax_services:
44             relax()
45
46 if __name__ == '__main__':
47     try:
48         Relax()
49         rospy.loginfo("All servos relaxed.")
50     except rospy.ROSInterruptException:
51         rospy.loginfo("All servos relaxed.")

```

Let's break this down line by line.

```

4  from std_srvs.srv import Empty
5  from arbotix_msgs.srv import SetSpeed

```

First we import the two service message types we will need. The arbotix relax service uses an Empty message from the standard services package (std_srvs). The set_speed service uses the SetSpeed message from the arbotix_msgs package.

```
12         self.joints = rospy.get_param('/arbotix/joints', '')
```

The arbotix_driver node stores all of its parameters under the /arbotix namespace on the ROS parameter server. The /arbotix/joints parameter contains the list of joints (and their individual parameters). We will need this list so we can iterate through all the joints to relax each one in turn.

```
17     # A list to hold the relax services
18     relax_services = list()
19
20     # A list to hold the set_speed services
21     set_speed_services = list()
```

Here we initialize a pair of lists to hold the relax and set_speed services for each servo we will connect to shortly.

```
24     for joint in self.joints:
25         relax_service = '/' + joint + '/relax'
26         rospy.wait_for_service(relax_service)
27         relax_services.append(rospy.ServiceProxy(relax_service, Empty))
28
29         speed_service = '/' + joint + '/set_speed'
30         rospy.wait_for_service(speed_service)
31         set_speed_services.append(rospy.ServiceProxy(speed_service,
32 SetSpeed))
```

Now we iterate through the list of joints, connect to the relax and set_speed service for each joint, then append a proxy to that service to the appropriate list. For example, if the first joint in the list is called head_pan_joint, then the variable relax_service is set to '/head_pan_joint/relax' which we know from the previous section is the correct service name. We then use a call to `wait_for_service()` to make sure the service is alive before appending the appropriate ServiceProxy object to the list of relax services. Notice how we use the `Empty` service type we imported at the top of the script to specify the type for the relax service. The same steps are then followed for the set_speed service.

```
33     # Set the servo speed to the default value
34     for set_speed in set_speed_services:
35         set_speed(default_speed)
```

With the lists of services in hand, we loop through the `set_speed_services` list and set each service to the default speed which should be set moderately low. (We use 0.5 radians per second in the script). The reason we do this is to protect us from any surprises the next time the service is activated. If we were to relax a servo right after it was set to a high speed, the next position request could set it flying faster than we would like. Line 35 shows how to call the `set_speed` service using the proxy we created together with an argument equal to the desired speed.

```
37      # Relax each servo
38      for relax in relax_services:
39          relax()
```

With all servos set to a safe speed, we then iterate through the relax services to turn off the torque for each servo. Note how in Line 39 we call the `relax()` service proxy with an empty argument since `relax` service uses the `Empty` message type.

5.7 Enabling or Disabling All Servos

Two additional scripts are provided in the `rbx2_dynamixels/scripts` directory to disable or enable all servos at once. To disable all servos at once, use the script `arbotix_disable_all_servos.py`:

```
$ rosrun rbx2_dynamixels arbotix_disable_all_servos.py
```

This script can be used as a kind of software "stop switch" since disabling the servos will cause them to both relax and become unresponsive to any further motion requests.

To re-enable all servos at the same time, run the `arbotix_enable_all_servos.py` script:

```
$ rosrun rbx2_dynamixels arbotix_enable_all_servos.py
```

The servos now respond to position requests.

6. ROBOT DIAGNOSTICS

Any self-respecting autonomous robot would be expected to monitor the status of its hardware to ensure it can carry out assigned tasks without running out of power or damaging its components. For a mobile robot equipped with a pan-and-tilt head or a multi-jointed arm, the key variables that require monitoring include:

- battery levels for both the robot and onboard computer (e.g. laptop battery)
- current drawn by the drive motors (to avoid overload)
- temperatures and loads on the joints (to avoid frying the servos)

It is up to the hardware driver for a given device to read the raw values of these variables. But we need a way to represent the values using a standard message type so that other ROS nodes can monitor them without having to worry about the underlying driver and measurement units.

To this end, ROS provides the [diagnostics](#) stack for collecting and summarizing data from sensors and actuators that can be used to flag potential problems. The key components of the diagnostics stack include the [DiagnosticStatus](#) message type, the [diagnostic_updater](#) API, the [diagnostic_aggregator](#), and an [analyzer configuration file](#). The API enables us to convert raw sensor values to an array of DiagnosticStatus fields which is then published as a [DiagnosticArray](#) message on the /diagnostics topic. The diagnostic_aggregator categorizes and organizes these messages into a hierarchical tree as specified by one or more analyzer configuration files and publishes the result on the /diagnostics_agg topic. The [rqt_robot_monitor](#) utility subscribes to the /diagnostics_agg topic and displays the status of any monitored components in a color-coded graphical form that makes it easy to spot a problem, or to drill down to a more detailed view of a given component.

You can also write your own nodes that can subscribe to the /diagnostics or /diagnostics_agg topics to monitor the status of various components and act appropriately when something is in trouble.

In some cases, the ROS drivers for the hardware you are using will already publish diagnostics information on the /diagnostics topic; you can then simply subscribe to the /diagnostics topic to make use of the information in your own nodes or use [rqt_robot_monitor](#) to visually inspect diagnostic information. In other cases, you might have to write your own driver for publishing the diagnostics information you require. Or, if a driver already exists but the data is not yet in the form of a ROS

diagnostics message, you will need to convert it appropriately as we show later in the chapter.

We will begin by assuming that the information we need is already published on the `/diagnostics` topic by one or more nodes. In a later section we'll describe how to add a diagnostics publisher to your own hardware drivers.

6.1 The DiagnosticStatus Message

The ROS `DiagnosticStatus` message type allows us to represent different kinds of diagnostic information in a common format. To see the message definition, run the command:

```
$ rosmsg show diagnostic_msgs/DiagnosticStatus
```

```
byte OK=0
byte WARN=1
byte ERROR=2
byte level
string name
string message
string hardware_id
diagnostic_msgs/KeyValue[] values
  string key
  string value
```

As we can see above, the message first enumerates three general status levels: `OK`, `WARN` and `ERROR`. The `level` field holds the current status of the device itself and will have a value of 0, 1 or 2 to indicate a status of `OK`, `WARN` or `ERROR`, respectively. Next we see three `string` fields for storing the `name` of the component, an arbitrary `message`, and its `hardware_id`. The rest of the message is an array of `key-value` pairs that can store the raw values returned by the hardware driver. For example, the pair `key=temperature, value=31` would mean that the temperature of the device is 31 degrees Celsius.

NOTE: For readers upgrading from ROS Hydro, a [new status](#) of `STALE` has been added to the `DiagnosticStatus` message. This means that the message now has a new MD5 signature and will not be compatible with nodes running under ROS Hydro or earlier. Therefore, be sure not mix diagnostics nodes or robot monitors running on pre-Indigo versus post-Indigo ROS distributions.

A given hardware driver will typically publish an array of such values using the [`DiagnosticArray`](#) message type which looks like this:

```
std_msgs/Header header
diagnostic_msgs/DiagnosticStatus[] status
```

As we can see, this message consists of a ROS header and an array of `DiagnosticStatus` messages, one for each component being monitored. This will become clearer below when we look at a concrete example using Dynamixel servos.

6.2 The Analyzer Configuration File

To monitor a device using the diagnostic aggregator, we need an entry in an analyzer configuration file such as the one shown below that is often used with Dynamixel servos:

```
1 pub_rate: 1.0
2 analyzers:
3   joints:
4     type: GenericAnalyzer
5     path: 'Joints'
6     timeout: 5.0
7     contains: '_joint'
```

While it is possible to define custom analyzers for different devices, we will only need to use the [GenericAnalyzer](#) type. You can find out more about analyzer configuration parameters in the [GenericAnalyzer Tutorial](#) on the ROS Wiki. For now, the parameters shown above will suffice. Let's break it down line by line.

Line 1 specifies the rate at which we want the aggregator to publish diagnostics information. A rate of 1.0 Hz is probably fast enough for most devices such as servos, batteries, drive motors, etc.

Line 2 specifies that all our parameters will fall under the `~analyzers` namespace which is required when using a `GenericAnalyzer` with the diagnostic aggregator. We won't need to access this namespace directly but the line must appear in our config file.

Line 3 indicates that we will use the `joints` namespace under the `~analyzers` namespace to store the aggregated diagnostics for servo joints.

Line 4 indicates that we will use the `GenericAnalyzer` plugin type to analyze these devices.

In Line 5, the `path` parameter defines the string that will be used to categorize these devices when viewed in the `rqt_robot_monitor` GUI. In this case, we want all our servos to be listed under the category 'Joints'.

Line 6 defines a `timeout` for receiving an update from the device. If no data is received from the device within this time frame, the device will be marked as "Stale" in the `rqt_robot_monitor`.

Finally, line 7 is really the key to the entire file. Here we specify a string that should appear somewhere in the diagnostic array message to identify the type of device we want to monitor. As we will see in the next section, the names of our joint controllers all contain the string "`_joint`", namely `head_pan_joint` and `head_tilt_joint`. So using the string "`_joint`" for the `contains` parameter should work in this case. As shown in the online [GenericAnalyzer Tutorial](#), one can also use the parameter `startswith` instead of `contains`. In that case, we could use the word '`Joint`' since the name field in the diagnostic messages for the servos all begin with '`Joint`'.

In a later section, we will see how we can configure more than one analyzer in a single configuration file.

6.3 Monitoring Dynamixel Servo Temperatures

If you are using Dynamixel servos on your robot, you know they can get hot under load. Both overheating and excessive loads can easily damage a servo which can become rather expensive if you are not careful.

Dynamixels have built-in protection against such failures by automatically shutting down when either the temperature, the load, or the voltage exceeds a pre-defined threshold. The thresholds are set directly in firmware so this level of damage control occurs independently of ROS. The factory default values for these thresholds are generally OK. However, it's not a bad idea to monitor servo temperatures and loads at the ROS level as well. This way we can preemptively give a servo or group of servos a rest when they might be in danger of overheating or overloading. Furthermore, if a servo is allowed to be shutdown via the firmware mechanism, it is often necessary to power-cycle the entire bus to regain control of the servo even after it cools down or is no longer overloaded.

6.3.1 Monitoring the servos for a pan-and-tilt head

For the sake of illustration, we'll assume that your robot has a pan-and-tilt head using a pair of Dynamixel servos like we covered in Chapter 12 of *Volume 1*. We'll use the launch file `pi_robot_head_only.launch` in the `rbx2_bringup/launch` directory as we did when testing the `arbotix` driver with real servos in the previous chapter. This launch file uses the `arbotix` driver to connect to two AX-12 Dynamixel servos with hardware IDs 1 and 2 and with joint names `head_pan_joint` and `head_tilt_joint`.

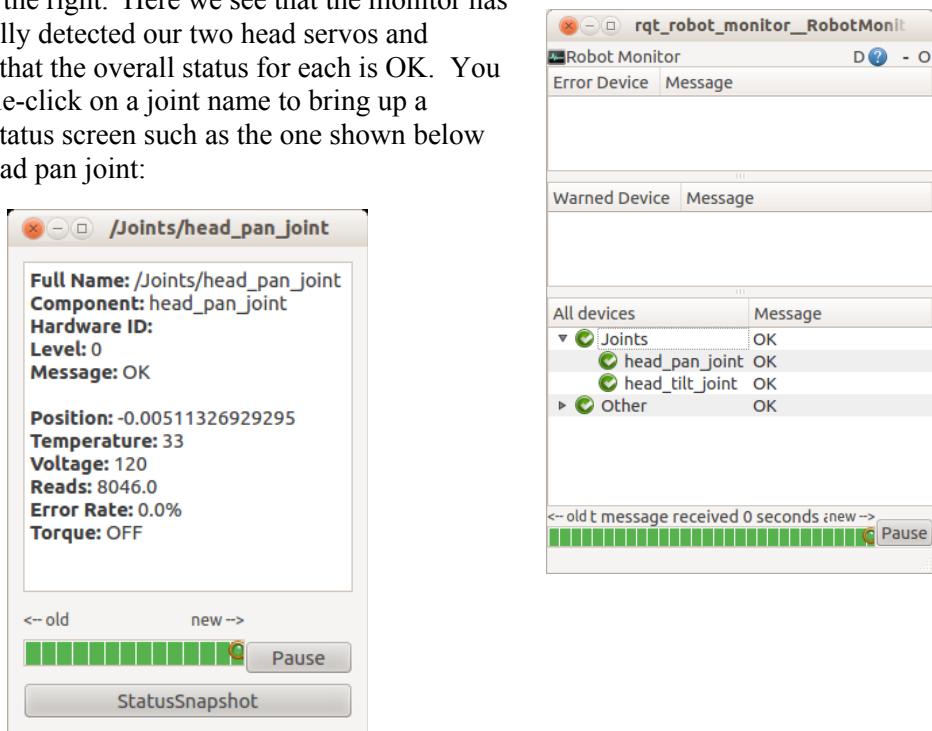
Assuming you have your servos connected to a USB2Dynamixel controller on USB port `/dev/ttyUSB0`, run the launch file with the `sim` parameter set to `false`:

```
$ roslaunch rbx2_bringup pi_robot_head_only.launch sim:=false
```

If your USB2Dynamixel controller is connected to a different USB port, you can run the launch file with the `port` argument. For example, if your controller is on USB port `/dev/ttyUSB1`, use the command:

```
$ roslaunch rbx2_bringup pi_robot_head_only.launch sim:=false \
  port:=/dev/ttyUSB1
```

After a brief delay, you should see the `rqt_robot_monitor` window appear like the image on the right. Here we see that the monitor has successfully detected our two head servos and indicates that the overall status for each is OK. You can double-click on a joint name to bring up a detailed status screen such as the one shown below for the head pan joint:



This status screen indicates that the head pan servo is essentially centered (position = -0.005), the temperature is a safe 33 degrees Celsius, the input voltage is 12V (multiplied by 10 for some reason—it is not 120 volts!), the error rate is a perfect 0.0% and the torque is currently off meaning the servo is relaxed and can be turned by hand.

The `arbotix` driver is programmed to publish diagnostics data for the servos it controls on the `/diagnostics` topic. Our launch file then runs the `diagnostic_aggregator` node and `rqt_robot_monitor` to summarize the status of each servo. If a servo becomes overheated or overloaded, its status will turn to `ERROR` and it will be displayed in red instead of green. If it is getting warm but not yet too hot, the status will be `WARN` with a display color of yellow. The `arbotix` driver is hard coded to assign an `ERROR` status for any temperature over 60 degrees Celsius and a `WARN` status for a temperature of 50 degrees or more. We will learn how to write our own diagnostics publisher later in the chapter and how we can set the thresholds as we like.

Let us now take a closer look at the `pi_robot_head_only.launch` file to see how the aggregator and rqt monitor nodes are run. Near the bottom of the launch file you will find the following lines:

```
<node pkg="diagnostic_aggregator" type="aggregator_node"
name="diagnostic_aggregator" clear_params="true" unless="$(arg sim)">
    <rosparam command="load" file="$(find
rbx2_dynamixels)/config/head_only_diagnostics.yaml" />
</node>

<node pkg="rqt_robot_monitor" type="rqt_robot_monitor"
name="rqt_robot_monitor" unless="$(arg sim)" />
```

Here we run the `diagnostic_aggregator` node which loads the configuration file `head_only_diagnostics.yaml` file found in the `rbx2_dynamixels/config` directory. We'll look at that file shortly. We also delete any left over diagnostic aggregator parameters that might be left over from previous runs by using the `clear_params="true"` argument.

Next we fire up the `rqt_robot_monitor` node which generates the GUI shown earlier for viewing servo status visually.

The `head_only_diagnostics.yaml` file defines the analyzers we want to run and how the information should be organized. Here's what that file looks like:

```
1 pub_rate: 1.0
2 analyzers:
3     joints:
4         type: GenericAnalyzer
5         path: 'Joints'
6         timeout: 5.0
7         contains: '_joint'
```

We have already described this very same configuration file in the section on Analyzer Configuration files. If you think that a publishing rate of 1 Hz is too slow, you can increase the `pub_rate` parameter. Line 7 indicates that the data we want to summarize comes from diagnostic entries whose name contains the string '`_joint`'. To see why this works, we need to examine the actual diagnostic messages being published on the `/diagnostics` topic. Let's turn to that next.

6.3.2 Viewing messages on the `/diagnostics` topic

As we have already explained, the `arbotix` driver takes care of publishing ROS diagnostic messages for us. The driver talks directly to the servo firmware over the serial port and then uses the ROS Diagnostics API to publish the data on the `/diagnostics` topic.

To see the messages being published on the `/diagnostics` topic while you are connected to the servos, open another terminal and run the command:

```
$ rostopic echo /diagnostics | more
```

The first part of the output should look something like this:

```
header:
  seq: 2125
  stamp:
    secs: 1405518802
    nsecs: 322613000
  frame_id: ''
status:
-
  level: 0
  name: head_controller
  message: OK
  hardware_id: ''
  values:
  -
    key: State
    value: Not Active
-
  level: 0
  name: head_pan_joint
  message: OK
  hardware_id: ''
  values:
  -
    key: Position
    value: -1.01242732
```

```
key: Temperature
value: 39
-
key: Voltage
value: 120
```

First we see the header fields, then the beginning of the `status` array. The hyphen (-) character indicates the beginning of an array entry. The first hyphen above indicates the beginning of the first entry of the `status` array. This entry has array index 0 so this first entry would be referred to as `/diagnostics/status[0]`. In this case, the entry refers to the head controller that controls both servos. Since the head controller is a software component, it does not have position or temperature so let's look at the next array entry.

The second block above would have array index 1 so this entry would be accessed as `/diagnostics/status[1]`. In this case, the entry refers to the head pan joint. The indented hyphens below the `values` keyword indicate the entries in the key-value array for this diagnostic element. So the `Temperature` value for this servo would be indexed as `/diagnostics/status[1].values[1]`. Note how we use a period (.) to indicate subfields of an array element. If you wanted to echo just the temperature of the head pan servo, you could use the following command:

```
$ rostopic echo /diagnostics/status[1].values[1]
```

Returning now to the previous output above, notice that the first few status fields for the head pan joint are:

```
status:
-
  level: 0
  name: head_pan_joint
  message: OK
  hardware_id: ''
```

The critical result here is the value for the `level` subfield which is 0 in this case. Recall that a value of 0 maps to a diagnostics status of `OK`. The relevant component should be identified by the `name` and sometimes the `hardware_id` fields. Here we see that this array element refers to the `head_pan_joint` but that the hardware ID is not specified.

If you issue the command '`rostopic echo /diagnostics | more`' again and keep hitting the space bar to scroll through the messages, you will see status messages for each servo. In particular, the message for the head tilt joint begins like this:

```
level: 0
name: head_tilt_joint
message: OK
hardware_id: ''
```

So we see that the value of the name field is simply the joint name we assigned to each servo in our ArbotiX config file. Since each joint name ends with the string '_joint', our analyzer configuration file, `head_only_diagnostics.yaml`, can use this string for the value of the `contains` parameter to let the aggregator know that diagnostic messages containing this string in their name field are the ones we're interested in.

6.3.3 Protecting servos by monitoring the `/diagnostics` topic

The `rbx2_diagnostics` package includes a script called `monitor_dynamixels.py` in the `nodes` subdirectory. This node subscribes to the `/diagnostics` topic, extracts the messages relevant to the servos and disables the servos if they appear to be in trouble. The script then counts off a timer to let the servos cool down, then re-enables them when their temperature is back to normal.

The script is fairly long so let's focus on the key lines.

```
4  from diagnostic_msgs.msg import DiagnosticArray, DiagnosticStatus
5  from arbotix_msgs.srv import Relax, Enable
```

Near the top of the script, we import the diagnostics message types we will need as well as the `Relax` and `Enable` services from the `arbotix_msgs` package.

```
12      # The arbotix controller uses the /arbotix namespace
13      namespace = '/arbotix'
14
15      # Get the list of joints (servos)
16      self.joints = rospy.get_param(namespace + '/joints', '')
17
18      # Minimum time to rest servos that are hot
19      self.minimum_rest_interval = rospy.get_param('~minimum_rest_interval',
60)
20
21      # Initialize the rest timer
22      self.rest_timer = 0
23
24      # Are we already resting a servo?
25      self.resting = False
26
27      # Are the servos enabled?
28      self.servos_enabled = False
29
30      # Have we issued a warning recently?
31      self.warned = False
```

```

32     # Connect to the servo services
33     self.connect_servos()
34
35     rospy.Subscriber('diagnostics', DiagnosticArray, self.get_diagnostics)

```

The list of joints controlled by the `arbotix_driver` node is stored in the ROS parameter `/arbotix/joints`. In fact, this parameter is the same as the `joints` parameter we defined in our `arbotix` configuration file. So we store the joint list (actually a dictionary) in the variable `self.joints`.

We also read in a parameter for the `minimum_rest_interval` (in seconds) to give a servo time to cool off before re-enabling. This parameter can be set in the launch file but we give it a default value of 60 seconds. If we left out the minimum cool-off period, then an overheating servo would be disabled just long enough to cool off one or two degrees, then it would probably just overheat right away again and so on.

Next we initialize a timer to track how long we have disabled a servo as well as a couple of boolean flags to indicate when the servos are disabled and when we are resting. Then we call the `connect_servos()` function (described below) that takes care of connecting to the various topics and services related to controlling the Dynamixels.

The final line above subscribes to the `/diagnostics` topic and sets the callback function to `self.get_diagnostics()` that we will look at next.

```

35     def get_diagnostics(self, msg):
36         if self.rest_timer != 0:
37             if rospy.Time.now() - self.rest_timer <
rospy.Duration(self.minimum_rest_interval):
38                 return
39             else:
40                 self.resting = False
41                 rest_timer = 0

```

In the first part of the callback function, we check the status of the `rest_timer` and if we still have some time left on the clock we return immediately. Otherwise we reset the timer to 0 and the `resting` flag to `False`, then continue on to the following lines.

```

45     for k in range(len(msg.status)):
46         # Check for the Dynamixel identifying string in the name field
47         if not '_joint' in msg.status[k].name:
48             # Skip other diagnostic messages
49             continue

```

The callback function receives the `DiagnosticArray` message as the argument `msg`. Each element of the array is an individual diagnostic message so we want to iterate over all such messages. The first thing we check for is that the string '`_joint`' is in the message name and if it is not, we skip to the next message using the `continue` statement.

```

48      # Check the DiagnosticStatus level for this servo
49      if msg.status[k].level == DiagnosticStatus.ERROR:
50          # If the servo is overheating, then disable all servos
51          if not self.resting:
52              rospy.loginfo("DANGER: Overheating servo: " +
str(msg.status[k].name))
53              rospy.loginfo("Disabling servos for a minimum of " +
str(self.minimum_rest_interval) + " seconds...")
54
55              self.disable_servos()
56              self.servos_enabled = False
57              self.resting = True
58              break
59      elif msg.status[k].level == DiagnosticStatus.WARN:
60          # If the servo is getting toasty, display a warning
61          rospy.loginfo("WARNING: Servo " + str(msg.status[k].name) + " " +
"getting hot...")
62          self.warned = True
63          warn = True

```

Now that we know we are dealing with a joint message, we check the diagnostic status level. If the status indicates `ERROR`, then this servo is overheated and we need to disable it. We could disable just the one servo, but to simplify things for now, we disable all servos when any one of them overheats.

If the servo is not hot enough for an `ERROR` status but it is warm enough for a `WARN` status, we display a warning message but do not disable the servos. We also set a couple of flags so we don't keep repeating the same warning message.

```

65      # No servo is overheated so re-enable all servos
66      if not self.resting and not self.servos_enabled:
67          rospy.loginfo("Dynamixel temperatures OK so enabling")
68          self.enable_servos()
69          self.servos_enabled = True
70          self.resting = False

```

Finally, if no servos are overheating and we are not currently in a rest period, re-enable the servos.

Let's now take a look at the three functions we called earlier in the script: `connect_servos()`, `disable_servos()` and `enable_servos()`.

```

78     def connect_servos(self):
79         # Create a dictionary to hold the torque and enable services
80         self.relax = dict()
81         self.enable = dict()
82
83         # Connect to the set_speed services and define a position publisher
84         for each servo
85             rospy.loginfo("Waiting for joint controllers services...")
86
87             for joint in sorted(self.joints):
88                 # A service to relax a servo
89                 relax = '/' + joint + '/relax'
90                 rospy.wait_for_service(relax)
91                 self.relax[joint] = rospy.ServiceProxy(relax, Relax)
92
93                 # A service to enable/disable a servo
94                 enable_service = '/' + joint + '/enable'
95                 rospy.wait_for_service(enable_service)
96                 self.enable[joint] = rospy.ServiceProxy(enable_service, Enable)
97
98             rospy.loginfo("Connected to servos.")

```

The `connect_servos()` function works in a similar way to the `arbotix_relax_all_servos.py` script we examined in the previous chapter. In this case, we create a proxy to the `relax` and `enable` services for each joint and store the proxies in a dictionary indexed by the joint name. We can then use these services in the `disable_servos()` and `enable_servos()` functions.

```

99     def disable_servos(self):
100         for joint in sorted(self.joints):
101             self.enable[joint](False)
102
103     def enable_servos(self):
104         for joint in sorted(self.joints):
105             self.enable[joint](True)

```

Once we have the `relax` and `enable` service proxies defined, it is straightforward to disable or enable all servos by iterating through the list of joints and calling the appropriate service. To relax and disable a servo, we call the `enable` service with the request value set to `False` and the servo will relax and ignore any future position requests. To re-enable a servo, we call the `enable` service with a request value of `True`.

To try out the script, first make sure you are running the launch file for your Dynamixels, like the `pi_robot_head_only.launch` file we used in the previous section. Then run the `monitor_dynamixels.launch` file:

```
$ rosrun rbt2_diagnostics monitor_dynamixels.launch
```

You should see the following series of INFO messages:

```
process[monitor_dynamixels-1]: started with pid [5797]
[INFO] [WallTime: 1403361617.457575] Waiting for joint controllers
services...
[INFO] [WallTime: 1403361617.471666] Connected to servos.
[INFO] [WallTime: 1403361617.662016] Dynamixel temperatures OK so
enabling
```

You can now run any other nodes such as head tracking that use the servos. The `monitor_dynamixels` node will monitor the servo temperatures and if any servo gets too hot, it will disable all servos until they are all back to a safe temperature. In the meantime, other nodes can continue publishing servo commands but they will simply be ignored until the Dynamixels are re-enabled.

6.4 Monitoring a Laptop Battery

If your robot carries a laptop on board, then you might like to monitor the laptop's battery status and publish the result on the `/diagnostics` array. We can then use that information to know when the battery is in trouble or the computer needs a recharge.

Fortunately for us, the folks at Willow Garage long ago created a node to monitor the TurtleBot's laptop battery by way of the system file `/proc/acpi/battery/BAT1`. We have included a modified version of that script called `laptop_battery.py` and it can be found in the `rbt2_bringup/nodes` subdirectory.

If you are working on a laptop as you read this, you can test out the script as follows:

```
$ rosrun rbt2_bringup laptop_battery.launch
```

The launch file specifies the rate at which to publish the diagnostics data (default 1Hz) and the `acpi_path` to the battery file (default `/proc/acpi/battery/BAT1`). Assuming the launch file runs without error, take a look at the data being published on the `/diagnostics` topic:

```
$ rostopic echo /diagnostics | more
```

The output on the screen should look something like this:

header:

```
seq: 21
stamp:
  secs: 1395878798
  nsecs: 494307994
frame_id: ''
status:
-
  level: 0
  name: Laptop Battery
  message: OK
  hardware_id: ''
values:
-
  key: Voltage (V)
  value: 15.12
-
  key: Current (A)
  value: -19.232
-
  key: Charge (Ah)
  value: 27.412
-
  key: Capacity (Ah)
  value: 33.46
-
  key: Design Capacity (Ah)
  value: 46.472
---
```

As you can see, the laptop battery node publishes the raw voltage, charge, and capacity values as key-value pairs. What is not apparent until you look carefully at the `laptop_battery.py` script, is that the overall diagnostics level will always be 0 (i.e. OK) as long as the battery is detected and its status has been updated in a timely manner. In other words, the status published here is not related to the battery's charge—it only reflects whether or not the battery is present and can be monitored. A new script, detailed in the next section, will publish a diagnostics message based on the charge itself.

6.5 Creating your Own Diagnostics Messages

The laptop battery node introduced in the previous section also publishes information about the battery charge on the topic `/laptop_charge`. Assuming you still have the laptop battery node running, you can view the messages being published on this topic with the command:

```
$ rostopic echo /laptop_charge | more
```

```
header:
```

```

seq: 31
stamp:
  secs: 1395880223
  nsecs: 497864007
frame_id: ''
voltage: 15.1199998856
rate: -11.0220003128
charge: 21.4249992371
capacity: 33.4599990845
design_capacity: 46.4720001221
percentage: 64
charge_state: 0
present: True
---

```

As you can see from the output, the message includes the voltage, rate of charge or discharge (depending on the sign), charge level, capacity, percent charge remaining, charge_state (0 = discharging, 1 = charging, 2 = charged) and whether or not it is present.

What we'd like to do is turn the percent charged value into a diagnostics level of OK, WARN or ERROR and publish it on the /diagnostics topic as a standard ROS diagnostics message. Our new script called, `monitor_laptop_charge.py`, subscribes to the /laptop_charge topic and converts the percent charge to one of the three standard diagnostic status levels: OK, WARN or ERROR.

We will use the `monitor_laptop_charge.launch` file to bring up the node and run the diagnostic aggregator with an appropriate configuration file. The launch file looks like this:

```

1 <launch>
2   <node pkg="rbx2_diagnostics" type="monitor_laptop_charge.py"
3     name="monitor_laptop_charge" output="screen">
4       <param name="warn_percent" value="50" />
5       <param name="error_percent" value="20" />
6     </node>
7   <node pkg="diagnostic_aggregator" type="aggregator_node"
8     name="diagnostic_aggregator" clear_params="true">
9     <rosparam command="load" file="$(find
rbx2_diagnostics)/config/power.yaml" />
10    </node>
11  <node pkg="rqt_robot_monitor" type="rqt_robot_monitor"
12    name="rqt_robot_monitor" />
13 </launch>

```

Lines 2–5 launch the `monitor_laptop_charge.py` node and set the warning battery level to 50% and the error level to 20%. Lines 7–9 bring up the `diagnostic_aggregator` with the configuration file `power.yaml` found in the `rbx2_diagnostics/config` directory. Finally, Line 11 brings up the `rqt_robot_monitor` GUI in case it is not already running.

The `power.yaml` configuration file looks like this:

```
1. pub_rate: 1.0
2. analyzers:
3.   power:
4.     type: GenericAnalyzer
5.     path: 'Power System'
6.     timeout: 5.0
7.     contains: ['Robot Battery', 'Robot Charge', 'Laptop Battery', 'Laptop Charge']
```

The publishing rate of 1.0 Hz specified by the `pub_rate` parameter seems appropriate for a battery monitor since battery levels do not change very quickly.

Line 3 indicates that we will use the `power` namespace under the `~analyzers` namespace to store the aggregated diagnostics for battery power.

Line 4 indicates that we will use the `GenericAnalyzer` plugin type to analyze these devices.

In Line 5, the `path` parameter defines the string that will be used to categorize these devices when viewed in the `rqt_robot_monitor` GUI. In this case, we want all our servos to be listed under the category 'Power System'.

Line 6 defines a timeout for receiving an update from the device. If no data is received from the device within this time frame, the device will be marked as "Stale" in the `rqt_robot_monitor`.

Finally, in Line 7 we specify a list of strings that should appear somewhere in the diagnostic array message to identify the type of device we want to monitor.

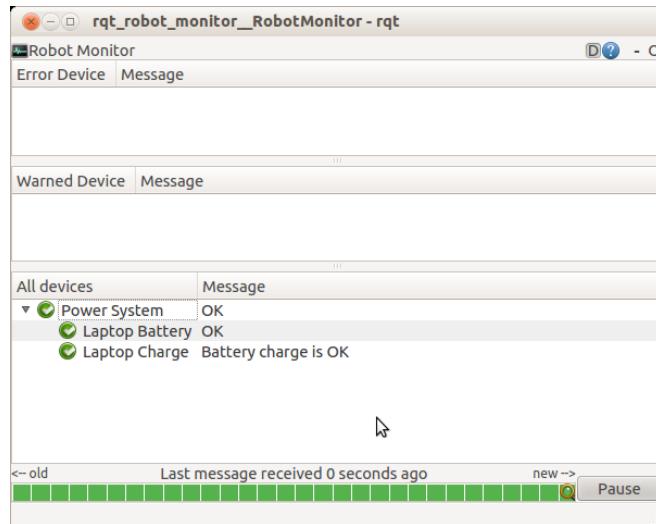
To try it out, first make sure you are on a laptop and then run the `laptop_battery.launch` file if it is not already running:

```
$ roslaunch rbx2_bringup laptop_battery.launch
```

Next, fire up `monitor_laptop_charge.launch`:

```
$ roslaunch rbx2_diagnostics monitor_laptop_charge.launch
```

After a brief delay you should see the `rqt_robot_monitor` window appear that should look like this:



Under the **Power System** category, we have two status messages: one for the battery overall (i.e. it can be detected) and one for the current charge level. The first status is provided by the `laptop_battery.py` node whereas the second is published by our new node `monitor_laptop_charge.py`. Let's take a look at that script now.

Link to source: [monitor_laptop_charge.py](#)

```
1 #!/usr/bin/env python
2
3 import rospy
4 from diagnostic_msgs.msg import DiagnosticArray, DiagnosticStatus, KeyValue
5 from smart_battery_msgs.msg import SmartBatteryStatus
6
7 class MonitorLaptopCharge():
8     def __init__(self):
9         rospy.init_node("monitor_laptop_charge")
10
11     # Get the parameters for determining WARN and ERROR status levels
12     self.warn_percent = rospy.get_param("~warn_percent", 50)
13     self.error_percent = rospy.get_param("~error_percent", 20)
14
15     # A diagnostics publisher
16     self.diag_pub = rospy.Publisher('diagnostics', DiagnosticArray,
queue_size=5)
17
```

```

18     # Subscribe to the laptop_charge topic
19     rospy.Subscriber('laptop_charge', SmartBatteryStatus,
self.pub_diagnostics)
20
21     rospy.loginfo("Monitoring laptop charge...")
22
23     def pub_diagnostics(self, msg):
24         # Pull out the percent charge from the message
25         percent_charge = msg.percentage
26
27         # Initialize the diagnostics array
28         diag_arr = DiagnosticArray()
29
30         # Time stamp the message with the incoming stamp
31         diag_arr.header.stamp = msg.header.stamp
32
33         # Initialize the status message
34         diag_msg = DiagnosticStatus()
35
36         # Make the name field descriptive of what we are measuring
37         diag_msg.name = "Laptop Charge"
38
39         # Add a key-value pair so we can drill down to the percent charge
40         diag_msg.values.append(KeyValue('percent_charge',
str(percent_charge)))
41
42         # Set the diagnostics level based on the current charge and the
threshold
43         # parameters
44         if percent_charge < self.error_percent:
45             diag_msg.level = DiagnosticStatus.ERROR
46             diag_msg.message = 'Battery needs recharging'
47         elif percent_charge < self.warn_percent:
48             diag_msg.level = DiagnosticStatus.WARN
49             diag_msg.message = 'Battery is below 50%'
50         else:
51             diag_msg.level = DiagnosticStatus.OK
52             diag_msg.message = 'Battery charge is OK'
53
54         # Append the status message to the diagnostic array
55         diag_arr.status.append(diag_msg)
56
57         # Publish the array
58         self.diag_pub.publish(diag_arr)
59
60 if __name__ == '__main__':
61     MonitorLaptopCharge()
62     rospy.spin()

```

Now let's break it down line by line:

```
4 from diagnostic_msgs.msg import DiagnosticArray, DiagnosticStatus, KeyValue
```

First we need a few message types from the `diagnostics_msgs` package.

```
5  from smart_battery_msgs.msg import SmartBatteryStatus
```

We also need the `SmartBatteryStatus` message type which is defined in the `smart_battery_msgs` package.

```
16      self.diag_pub = rospy.Publisher('diagnostics', DiagnosticArray,
queue_size=5)
```

Here we create a publisher to publish the laptop charge as a `DiagnosticArray` message on the `/diagnostics` topic.

```
19      rospy.Subscriber('laptop_charge', SmartBatteryStatus,
self.pub_diagnostics)
```

And here we create a subscriber to monitor the `/laptop_charge` topic. The callback function `self.pub_diagnostics` (explained below) will convert the laptop charge to a ROS diagnostics message.

```
23  def pub_diagnostics(self, msg):
24      # Pull out the percent charge from the message
25      percent_charge = msg.percentage
```

We begin the callback function assigned to the `/laptop_charge` topic subscriber. Recall that the `Status` message includes a field for the percentage charge remaining so we pull that value from the `msg` variable that is passed to the callback.

```
27      # Initialize the diagnostics array
28      diag_arr = DiagnosticArray()
29
30      # Time stamp the message with the incoming stamp
31      diag_arr.header.stamp = msg.header.stamp
```

We want to publish the percent charge as a `DiagnosticArray` message so first we create an empty array as the variable `diag_arr`. We then give the array the same timestamp as the incoming message stamp.

```
33      # Initialize the status message
34      diag_msg = DiagnosticStatus()
35
36      # Make the name field descriptive of what we are measuring
37      diag_msg.name = "Laptop Charge"
```

Since a DiagnosticArray is made up of individual DiagnosticStatus messages, we initial the variable diag_msg accordingly and assign a suitable label to the name field.

```
39         # Add a key-value pair so we can drill down to the percent charge
40         diag_msg.values.append(KeyValue('percent_charge',
41                                         str(percent_charge)))
```

Recall the that the values array in a DiagnosticStatus message are stored as KeyValue pairs where both members of the pair are strings. So here we append a KeyValue pair where the key is the string 'percent_charge' and the value is the actual percent charge converted to a string.

```
44     if percent_charge < self.error_percent:
45         diag_msg.level = DiagnosticStatus.ERROR
46         diag_msg.message = 'Battery needs recharging'
47     elif percent_charge < self.warn_percent:
48         diag_msg.level = DiagnosticStatus.WARN
49         diag_msg.message = 'Battery is below 50%'
50     else:
51         diag_msg.level = DiagnosticStatus.OK
52         diag_msg.message = 'Battery charge is OK'
```

We then set the value of the level and message fields of the diag_msg variable according to the thresholds we have set in the error_percent and warn_percent parameters.

```
54     # Append the status message to the diagnostic array
55     diag_arr.status.append(diag_msg)
56
57     # Publish the array
58     self.diag_pub.publish(diag_arr)
```

Finally, we append the diagnostic status message we have created to the diagnostics array and publish the result.

6.6 Monitoring Other Hardware States

The monitor_laptop_battery.py script described in the previous section illustrates how to construct and publish a diagnostics message based on some other value such as a sensor reading or, in that case, the charge level of the laptop battery. You can mimic this script to monitor the values of other hardware states you might want to add to the diagnostics array.

For example, if your robot is powered by a main battery and you can measure that battery's charge level with some type of sensor (e.g. a Phidgets voltage sensor), then you can convert that sensor reading to a ROS diagnostics message just as we did with the laptop charge. Similarly, some motor controller boards have pins for reporting the instantaneous current being drawn by the motors. If you want to keep tabs on these readings, convert them to ROS diagnostic messages and add them to your array.

The end result will be that all your hardware diagnostics can be displayed in the same `rqt_monitor` and with a quick glance at the monitor, you can see if any component has turned red indicating a status of `WARN` or `ERROR`. Conversely, if all status indicators are green, your robot should be generally OK.

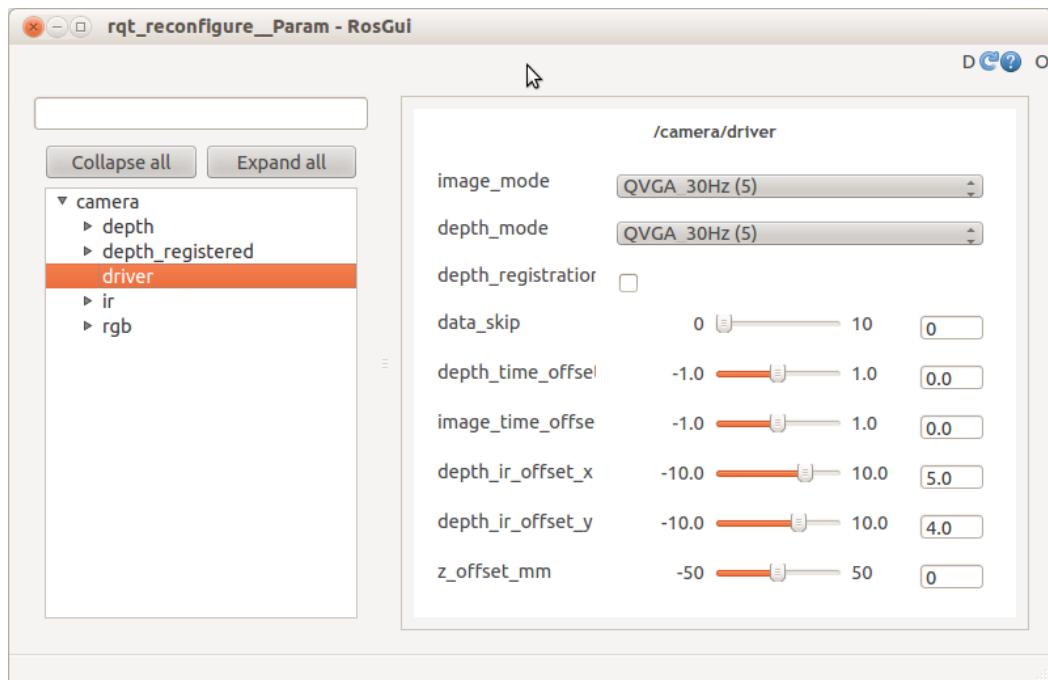
For a more in depth look at writing your own ROS diagnostics code, take a look at the [diagnostic_updater](#) Wiki page at ros.org. While the example given there is written in C++, a similar API exists for Python.

7. DYNAMIC RECONFIGURE

By now, we are quite familiar with setting ROS parameters in launch files and reading their values in our nodes using the `rospy.get_param()` function. But being able to change ROS parameters on the fly is often useful for tuning or debugging a running application. In *Volume 1*, we occasionally used the ROS `rqt_reconfigure` GUI to change the parameters for a running node. For example, if we are using a Kinect or Xtion Pro camera, we can change the camera's resolution without shutting down the driver by first bringing up the `rqt_reconfigure` GUI as follows

```
$ rosrun rqt_reconfigure rqt_reconfigure
```

After a short delay, the window similar to the following should appear:



In the image above, we have expanded the **camera** node and then selected the **driver** category. In the right-hand panel we can then change a number of the driver's parameters including the resolution using the pull-down menus labeled **image_mode** and **depth_mode**.

In addition to `rqt_reconfigure`, we can also use the dynamic reconfigure [command line](#) utilities to change parameters. However, in both cases there is a catch: only nodes that have been programmed using the dynamic reconfigure API can be updated this way. (In fact, other nodes will not even appear in the `rqt_reconfigure` GUI.) Fortunately, most nodes in the key ROS stacks and packages such as Navigation do support dynamic reconfigure, but many third-party nodes do not and therefore can only be tweaked by editing launch files and restarting, or by using the [rosparam](#) command line tool followed by a node restart.

7.1 Adding Dynamic Parameters to your own Nodes

Adding dynamic reconfigure support to your own nodes is not difficult and is covered for both C++ and Python in the step-by-step [Dynamic Reconfigure Tutorials](#) on the ROS Wiki. Here we will review those concepts using the fake battery simulator as an example.

7.1.1 Creating the `.cfg` file

Each parameter you want to control dynamically must be specified in the `.cfg` file for your node. These configuration files live in the `cfg` subdirectory of the package containing your node. The `cfg` file for the fake battery simulator is called `BatterySimulator.cfg` and lives in the directory `rbx2_utils/cfg`. Let's take a look at that file now:

```
1 #!/usr/bin/env python
2
3 PACKAGE = "rbx2_utils"
4
5 from dynamic_reconfigure.parameter_generator_catkin import *
6
7 gen = ParameterGenerator()
8
9 gen.add("battery_runtime", int_t, 0, "Battery runtime in seconds", 30, 1,
7200)
10
11 gen.add("new_battery_level", int_t, 0, "New battery level", 100, 0, 100)
12
13 exit(gen.generate(PACKAGE, "battery_simulator", "BatterySimulator"))
```

As you can see, the file is fairly simple. First we set the `PACKAGE` variable to the package we are in. Then we import the `catkin` dynamic reconfigure generator library. This allows us to create a `ParameterGenerator()` object and add a couple of integer parameters. Line 9 adds the `battery_runtime` parameter with a default value of 60 seconds and a range of 1-7200 seconds. Line 11 adds the `new_battery_level`

parameter with a default of 100 and a range of 0-100. The general form of the `add()` function is:

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

where the fields are:

- **name** - the name of the parameter in quotations
- **type** - the type of value stored, and can be any of `int_t`, `double_t`, `str_t`, or `bool_t`
- **level** - a bitmask which will later be passed to the dynamic reconfigure callback. When the callback is called all of the level values for parameters that have been changed are ORed together and the resulting value is passed to the callback. This can be used to organize parameters into groups such as those that control sensors that must be shutdown a restarted when a parameter is changed. See [this tutorial](#) for an example.
- **description** - string which describes the parameter
- **default** - the default value
- **min** - the min value (optional and does not apply to string and bool types)
- **max** - the max value (optional and does not apply to string and bool types)

NOTE: the **name** string cannot be the value "i", "state", or "name" as discussed in this [issue](#).

The final line of the `.cfg` file must be written carefully:

```
13 exit(gen.generate(PACKAGE, "battery_simulator", "BatterySimulator"))
```

The first string inside quotations should be the same as the name of the node we are configuring but without the file extension. The second string inside quotations must be the name of the `.cfg` file itself without the `.cfg` extension.

7.1.2 Making the `.cfg` file executable

Once the `.cfg` file is created, it must be made executable using the following commands:

```
$ roscd rbx2_utils/cfg  
$ chmod a+x BatterySimulator.cfg
```

7.1.3 Configuring the `CMakeLists.txt` file

Any package that contains dynamically configurable nodes must have a few configuration lines added to the package's `CMakeLists.txt` file. These lines cause `catkin_make` to run the `.cfg` file that generates the required runtime code. For the `BatterySimulator.cfg` file to be run when building the `rbx2_utils` package, the following lines would have to appear in the `CMakeLists.txt` file:

```
1 cmake_minimum_required(VERSION 2.8.3)
2
3 project(rbx2_utils)
4
5 find_package(catkin REQUIRED COMPONENTS dynamic_reconfigure)
6
7 generate_dynamic_reconfigure_options(
8     cfg/BatterySimulator.cfg
9     cfg/Pub3DTarget.cfg
10 )
11
12 catkin_package(DEPENDS CATKIN DEPENDS dynamic_reconfigure message_runtime)
```

The `find_package()` macro ensures that we include the `dynamic_reconfigure` package and all its supporting dependencies. The macro `generate_dynamic_reconfigure_options()` takes the paths to all the `.cfg` files (relative to the package directory) that we want to build. Here we list the `BatterySimulator.cfg` file we have been discussing as well as second `.cfg` file (`Pub3DTarget.cfg`) that we will use for another node in later chapters.

Finally, we include the `dynamic_reconfigure` package in the `catkin_package()` macro.

7.1.4 Building the package

After adding a `.cfg` file or making changes to an existing file, run `catkin_make` as usual:

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

7.2 Adding Dynamic Reconfigure Capability to the Battery Simulator Node

Recall that the battery simulator script `battery_simulator.py` can be found in the directory `rbx2_utils/nodes`. Let's take a more detailed look at that script now.

Link to source: [battery_simulator.py](#)

```

1  #!/usr/bin/env python
2
3  import rospy
4  from diagnostic_msgs.msg import *
5  from std_msgs.msg import Float32
6  from rbx2_msgs.srv import *
7  import dynamic_reconfigure.server
8  from rbx2_utils.cfg import BatterySimulatorConfig
9  import thread
10
11 class BatterySimulator():
12     def __init__(self):
13         rospy.init_node("battery_simulator")
14
15         # The rate at which to publish the battery level
16         self.rate = rospy.get_param("~rate", 1)
17
18         # Convert to a ROS rate
19         r = rospy.Rate(self.rate)
20
21         # The battery runtime in seconds
22         self.battery_runtime = rospy.get_param("~battery_runtime", 30) #
seconds
23
24         # The initial battery level - 100 is considered full charge
25         self.initial_battery_level = rospy.get_param("~initial_battery_level",
100)
26
27         # Error battery level for diagnostics
28         self.error_battery_level = rospy.get_param("~error_battery_level", 20)
29
30         # Warn battery level for diagnostics
31         self.warn_battery_level = rospy.get_param("~warn_battery_level", 50)
32
33         # Initialize the current level variable to the startup level
34         self.current_battery_level = self.initial_battery_level
35
36         # Initialize the new level variable to the startup level
37         self.new_battery_level = self.initial_battery_level
38
39         # The step sized used to decrease the battery level on each publishing
loop
40         self.battery_step = float(self.initial_battery_level) / self.rate /
self.battery_runtime
41
42         # Reserve a thread lock
43         self.mutex = thread.allocate_lock()
44
45         # Create the battery level publisher
46         battery_level_pub = rospy.Publisher("battery_level", Float32,
queue_size=5)
47
48         # A service to manually set the battery level
49         rospy.Service('~set_battery_level', SetBatteryLevel,
self.SetBatteryLevelHandler)
```

```

50          # Create a diagnostics publisher
51          diag_pub = rospy.Publisher("diagnostics", DiagnosticArray,
52                                      queue_size=5)
53
54          # Create a dynamic_reconfigure server and set a callback function
55          dyn_server = dynamic_reconfigure.server.Server(BatterySimulatorConfig,
56                                              self.dynamic_reconfigure_callback)
57
58          rospy.loginfo("Publishing simulated battery level with a runtime of "
59 + str(self.battery_runtime) + " seconds...")
59
60          # Start the publishing loop
61          while not rospy.is_shutdown():
62              # Initialize the diagnostics status
63              status = DiagnosticStatus()
64              status.name = "Battery Level"
65
66              # Set the diagnostics status level based on the current battery
67              # level
68              if self.current_battery_level < self.error_battery_level:
69                  status.message = "Low Battery"
70                  status.level = DiagnosticStatus.ERROR
71              elif self.current_battery_level < self.warn_battery_level:
72                  status.message = "Medium Battery"
73                  status.level = DiagnosticStatus.WARN
74              else:
75                  status.message = "Battery OK"
76                  status.level = DiagnosticStatus.OK
77
78              # Add the raw battery level to the diagnostics message
79              status.values.append(KeyValue("Battery Level",
80                                           str(self.current_battery_level)))
81
82              # Build the diagnostics array message
83              msg = DiagnosticArray()
84              msg.header.stamp = rospy.Time.now()
85              msg.status.append(status)
86
87              diag_pub.publish(msg)
88
89              battery_level_pub.publish(self.current_battery_level)
90
91              self.current_battery_level = max(0, self.current_battery_level -
92                                              self.battery_step)
93
94              r.sleep()
95
96          def dynamic_reconfigure_callback(self, config, level):
97              if self.battery_runtime != config['battery_runtime']:
98                  self.battery_runtime = config['battery_runtime']
99                  self.battery_step = 100.0 / self.rate / self.battery_runtime
100
101              if self.new_battery_level != config['new_battery_level']:
102                  self.new_battery_level = config['new_battery_level']

```

```

99         self.mutex.acquire()
100        self.current_battery_level = self.new_battery_level
101        self.mutex.release()
102
103    return config
104
105 def SetBatteryLevelHandler(self, req):
106     self.mutex.acquire()
107     self.current_battery_level = req.value
108     self.mutex.release()
109     return SetBatteryLevelResponse()
110
111if __name__ == '__main__':
112     BatterySimulator()

```

Since much of the script uses already familiar concepts, let's focus on the lines related to dynamic reconfigure.

```

7 import dynamic_reconfigure.server
8 from rbx2_utils.cfg import BatterySimulatorConfig

```

First we import the dynamic reconfigure server library as well as the config file for the battery simulator itself. Notice that this `BatterySimulatorConfig` object was created by `catkin_make` from our `.cfg` file.

```

9         # Create a dynamic_reconfigure server and set a callback function
10        dyn_server = dynamic_reconfigure.server.Server(BatterySimulatorConfig,
11 self.dynamic_reconfigure_callback)

```

Here we create the dynamic reconfigure server object that will service requests of type `BatterySimulatorConfig` and we set the callback function to `self.dynamic_reconfigure_callback()` that will be described next.

```

92     def dynamic_reconfigure_callback(self, config, level):
93         if self.battery_runtime != config['battery_runtime']:
94             self.battery_runtime = config['battery_runtime']
95             self.battery_step = 100.0 / self.rate / self.battery_runtime
96
97         if self.new_battery_level != config['new_battery_level']:
98             self.new_battery_level = config['new_battery_level']
99             self.mutex.acquire()
100            self.current_battery_level = self.new_battery_level
101            self.mutex.release()
102
103    return config

```

The dynamic reconfigure callback automatically receives two arguments: the `config` argument that contains the names and values of the parameters being passed by the requesting client (e.g. `rqt_reconfigure`) and the `level` argument which we won't use here. The `config` argument is a dictionary mapping parameter names to values. In the case of our battery simulator, the `BatterySimulator.cfg` file defines two configurable parameters, `battery_runtime` and `new_battery_level`, so these names will be the keys of the `config` dictionary.

On Line 93 above we compare the current `battery_runtime` to the value passed in through the `config` argument and if they differ, we update the value in the script.

Similarly, in Line 97, we do the same with the `new_battery_level`, although this time we have to acquire a lock since the script value is also modified by the main loop that decreases its value on every "tick" of the clock.

And that's pretty much all there is to it. Our battery simulator node can now be configured on the fly using `rqt_reconfigure` or any [other node](#) that sends a client request to update its parameters.

7.3 Adding Dynamic Reconfigure Client Support to a ROS Node

In the previous section, we showed how to add dynamically configurable parameters to the battery simulator node. To do this, we used the dynamic reconfigure *server* library. Running the node as a dynamic reconfigure server enables it to accept requests to change parameters from client nodes such as `rqt_reconfigure`. However, occasionally, you might want one of your own nodes to act as a client and request parameter changes in other nodes. Alternatively, you might want a node to simply know when another node's parameters have been changed and act accordingly. While not as common as setting up a node as a dynamic configure server, let's take a quick look at how it can be done.

Our demo script is called `dyna_client.py` and can be found in the `rbx2_utils/nodes` directory. This node connects to the battery simulator and sets the battery level alternately between 100 and 0 every 10 seconds. While not the most useful example, it at least illustrates the process. Here is the full script.

Link to source: [dyna_client.py](#)

```
1 #!/usr/bin/env python
2
3 import rospy
4 import dynamic_reconfigure.client
5
6 class DynaClient():
7     def __init__(self):
```

```

8     rospy.init_node("dynamic_client")
9
10    rospy.loginfo("Connecting to battery simulator node...")
11
12    client = dynamic_reconfigure.client.Client("battery_simulator",
13        timeout=30, config_callback=self.callback)
14
15    r = rospy.Rate(0.1)
16
17    charge = True
18
19    while not rospy.is_shutdown():
20        if charge:
21            level = 100
22        else:
23            level = 0
24
25        charge = not charge
26
27        client.update_configuration({"new_battery_level": level})
28
29        r.sleep()
30
31    def callback(self, config):
32        rospy.loginfo("Battery Simulator config set to: " +
33        str(config['new_battery_level']))
34
35 if __name__ == "__main__":
36     DynaClient()

```

Let's now look at the key lines.

```
4 import dynamic_reconfigure.client
```

First we import the dynamic configure client library rather than the server library.

```
12     client = dynamic_reconfigure.client.Client("battery_simulator",
13         timeout=30, config_callback=self.callback)
```

Here we create a client connection to the battery simulator node, set the timeout to 30 seconds, and assign a callback function defined below. Note that the callback function will be called whenever the parameters of the battery simulator node are changed, even by another node such as `rqt_reconfigure`. This allows the `dyna_client` node to monitor parameter changes in the battery simulator node and adjust its own behavior if desired.

```
13     r = rospy.Rate(0.1)
```

This line sets the rate for our main loop to 0.1 Hz or once cycle per 10 seconds.

```
16     charge = True
17
18     while not rospy.is_shutdown():
19         if charge:
20             level = 100
21         else:
22             level = 0
23
24         charge = not charge
25
26         client.update_configuration({ "new_battery_level": level})
27
28         r.sleep()
```

We use the `charge` variable to alternate between a battery level of 100 and 0. We then enter the main loop and run the `client.update_configuration()` function to set the battery simulator's `new_battery_level` parameter to either 100 or 0 depending on the value of `charge`. Note how the parameter name and value are specified as a Python dictionary. If we wanted to change both the `new_battery_level` and `battery_runtime` parameters at the same time, the update line would look like this:

```
client.update_configuration({ "new_battery_level": level, "battery_runtime": runtime})
```

where `runtime` would hold the new runtime value.

Finally, let's look at the client callback function:

```
30     def callback(self, config):
30         rospy.loginfo("Battery Simulator config set to: " +
str(config[ 'new_battery_level']))
```

The dynamic configure client callback function automatically receives the current parameter configuration as an argument that we have named `config` above. This argument is a dictionary of values and we can pull out the current value of any parameter by name. In Line 30 above, we simply display the value of the `new_battery_level` parameter in the terminal window.

To see the script in action, first fire up the battery simulator node;

```
$ roslaunch rbx2_utils battery_simulator.launch
```

Next, run the `dyna_client.py` node:

```
$ rosrun rbx2_utils dyna_client.py
```

If you monitor the `dyna_client.py` terminal window, you will see the output from the `rospy.loginfo()` command above every 10 seconds and the value of the `new_battery_level` parameter will alternate between 100 and 0. To verify that the battery simulator is actually responding to the new battery level, open another terminal and monitor the battery level itself as published on the `/battery_level` topic:

```
$ rostopic echo /battery_level
```

Here you should see the battery level count down as usual starting at 100 but every 10 seconds it will jump either to 0 or back to 100.

Finally, bring up `rqt_reconfigure` and change the `new_battery_level` parameter using the GUI. Back in the `dyna_client.py` window, you should see the output from the callback function both every 10 seconds from its own action as well as any time you change the value using `rqt_reconfigure`.

7.4 Dynamic Reconfigure from the Command Line

It is also possible to change the value of a dynamic parameter from the command line as detailed on the [ROS Wiki](#). The main command is `dynparam` that takes a number of subcommands as follows:

- `dynparam list`: list currently running nodes that are dynamically configurable
- `dynparam get`: get node configuration
- `dynparam set`: configure node
- `dynparam set_from_parameters`: copy configuration from parameter server
- `dynparam dump`: dump configuration to file
- `dynparam load`: load configuration from file

To run any of these commands, use the syntax:

```
$ rosrun dynamic_reconfigure dynparam COMMAND
```

For example, to set the resolution of a Kinect or Xtion Pro depth camera to 640x480 (parameter/value `image_mode=2`), and assuming the camera node is running as `/camera/driver`, you would use the command:

```
$ rosrun dynamic_reconfigure dynparam set /camera/driver image_mode 2
```

And to set the depth mode similarly:

```
$ rosrun dynamic_reconfigure dynparam set /camera/driver depth_mode 2
```

The `dump` and `load` commands are useful if you have spent some time tweaking a set of parameters using `rqt_reconfigure` or another method and want to save those changes to a file that can be loaded at a later time. To save the current configuration of a node called `my_node` to a file called `my_node.yaml`, use the command:

```
$ rosrun dynamic_reconfigure dynparam dump /my_node my_node.yaml
```

NOTE: There is currently a bug in ROS Indigo running on Ubuntu 14.04 that prevents reloading parameters. You can follow the progress of [this issue](#) on Github.

Then to load this configuration at a later time, run the command:

```
$ rosrun dynamic_reconfigure dynparam load /my_node my_node.yaml
```

You can also load previously saved parameters by way of a launch file like this:

```
<launch>
  <node pkg="dynamic_reconfigure" name="dynaparam" type="dynaparam"
    command="load" file="$(find my_package)/my_node.yaml" />
</launch>
```

The `<node>` line above will load the dynamic parameters from the file `my_node.yaml` located in the package `my_package`. Of course, you can have other nodes launched in the same launch file.

8. MULTIPLEXING TOPICS WITH MUX & YOCS

In the chapter on ROS Diagnostics, we programmed a node (`monitor_dynamixels.py`) that can disable one or more Dynamixel servos if they get too hot. Other nodes can then attempt to control the servos but their requests will be ignored by the `arbotix` driver. However, not all hardware controllers provide this level of control. Furthermore, there may be other reasons why we want to control access to a shared resource that requires a more flexible approach.

Suppose for example that we want the robot base controller to always give highest priority to manual input such as a joystick. Other nodes such as `move_base` might be attempting to control the base at the same time, but the user should always be able to override the input using manual control. Rather than build this kind of control override into the base driver itself, we can use the ROS [mux](#) utility that is part of the [topic_tools](#) meta package.

The `mux` node enables us to multiplex several input topics into one output topic. Only one input at a time is passed through to the output. Services are provided for selecting the active input as well as for adding and deleting input topics as we shall describe later. Note that even without multiplexing, one can control a ROS base controller using multiple inputs simultaneously. The problem is that all such inputs will typically publish their `Twist` messages on the same `/cmd_vel` topic. The result is jerky motion of the robot as it alternately responds to one `Twist` message or another from the various sources. The `mux` utility gets around this problem by allowing only one input through at any given time.

In the case of our base controller example, let's call the navigation input topic `move_base_cmd_vel` and the joystick input topic `joystick_cmd_vel`. Let the output topic be `cmd_vel`. The following command would then set up our multiplexer:

```
$ rosrun topic_tools mux cmd_vel move_base_cmd_vel joystick_cmd_vel \
  mux:=mux_cmd_vel
```

Here we run the `mux` node from the `topic_tools` package with the first argument specifying the output topic and the next two arguments specifying the two input topics. The last argument above gives the `mux` node a unique name and determines the namespace for the service and topic names as we will see later on.

Note that at startup, the first input topic on the command line is selected. So if we were to execute the command above, the node named `mux_cmd_vel` would be initially listening on the topic `move_base_cmd_vel` for messages that it would then pass on to the output `cmd_vel` topic.

8.1 Configuring Launch Files to use `mux` Topics

So far so good, but how do all of these topics and messages actually end up controlling the robot base? Up until now, we have assumed that the robot base controller is listening for Twist messages on the `/cmd_vel` topic. With the above `mux` node running, we could still control the base that way: by publishing Twist messages directly to the `/cmd_vel` topic. However, to implement the multiplexing feature of the `mux` node, we should configure our navigation controller (e.g. `move_base`) to publish its Twist messages on the topic `move_base_cmd_vel` and the joystick teleop node to publish on the topic `joystick_cmd_vel`. This way, neither node is publishing directly on the traditional `/cmd_vel` topic; instead, the `mux` node gets to decide which input will be passed along to `/cmd_vel`.

An example of how to do this is included in the two launch files `mux_fake_move_base.launch` and `mux_joystick_teleop.launch` found in the `rbx2_nav/launch` directory. Let's look at `mux_fake_move_base.launch` first:

```
<launch>
  <node pkg="move_base" type="move_base" respawn="false" name="move_base"
  clear_params="true" output="screen">
    <remap from="cmd_vel" to="move_base_cmd_vel" />
    <rosparam file="$(find rbx2_nav)/config/fake/costmap_common_params.yaml"
    command="load" ns="global_costmap" />
    <rosparam file="$(find rbx2_nav)/config/fake/costmap_common_params.yaml"
    command="load" ns="local_costmap" />
    <rosparam file="$(find rbx2_nav)/config/fake/local_costmap_params.yaml"
    command="load" />
    <rosparam file="$(find rbx2_nav)/config/fake/global_costmap_params.yaml"
    command="load" />
    <rosparam file="$(find rbx2_nav)/config/fake/base_local_planner_params.yaml"
    command="load" />
  </node>
</launch>
```

The key line is highlighted in bold above. This line simply remaps the `cmd_vel` topic that `move_base` normally publishes on to the `move_base_cmd_vel` topic. That's all there is to it. The `mux_joystick_teleop.launch` file uses a similar remapping:

```
<launch>
  <!-- Teleop Joystick -->
  <node pkg="turtlebot_teleop" type="turtlebot_teleop_joy"
  name="turtlebot_teleop_joystick" output="screen">
    <remap from="turtlebot_teleop_joystick/cmd_vel" to="joystick_cmd_vel" />
```

```

<param name="scale_angular" value="1.5"/>
<param name="scale_linear" value="0.3"/>
<param name="axis_deadman" value="7"/>
</node>

<node pkg="joy" type="joy_node" name="joystick" output="screen" />
</launch>

```

Once again, the highlighted line provides the remapping, this time from `turtlebot_teleop_joy_stick_cmd_vel` that is used by default by the `turtlebot_teleop_joy` node we are using to the topic `joystick_cmd_vel` that we will use with our `mux` node. (We could have just used the topic name `turtlebot_teleop_joy_stick_cmd_vel` directly with our `mux` node but we chose `joystick_cmd_vel` to be more generic and so we need to remap it here.)

8.2 Testing `mux` with the Fake TurtleBot

To try it all out, first launch the fake TurtleBot as we did in *Volume I*:

```
$ rosrun roslaunch rbx1 Bringup fake_turtlebot.launch
```

Next, run the `mux_fake_move_base_blank_map.launch` file. This file includes the `mux_fake_move_base.launch` file described above but also brings up a blank map:

```
$ rosrun roslaunch rbx2_nav mux_fake_move_base_blank_map.launch
```

Assuming you have a joystick attached to your computer, bring up the `mux` joystick node:

```
$ rosrun roslaunch rbx2_nav mux_joystick_teleop.launch
```

Finally, bring up `RViz` with a suitable configuration file:

```
$ rosrun rviz rviz -d `rospack find rbx2_nav`/config/nav.rviz
```

The first thing you will notice is that the fake TurtleBot does not respond to any control input—neither the joystick nor by setting 2D Nav goals in `RViz`. That is because the ArbotiX base controller we are using for the fake TurtleBot is listening on the `/cmd_vel` topic for `Twist` messages but all of our control inputs are publishing on their own topics. So now let's run the `mux` node described earlier:

```
$ rosrun topic_tools mux cmd_vel move_base_cmd_vel joystick_cmd_vel \
mux:=mux_cmd_vel
```

And you should see the message:

```
[ INFO] [1395796224.736609630]: advertising
```

on the terminal. Since the first input topic listed on the `mux` command line above is `move_base_cmd_vel` and since this is the topic used by our `mux_fake_move_base.launch` file, you should now be able to control the fake TurtleBot by setting 2D navigation goals in `RViz` with the mouse. However, at this point, the joystick should not have any effect on the robot's motion. Let's turn to that next.

8.3 Switching Inputs using `mux` Services

Referring back to the `mux` Wiki page, we see that a `mux` node defines three services:

- `mux/select` – Select an input topic to output, or `__none` to turn off output
- `mux/add` – Add a new input topic
- `mux/delete` – Delete an input topic

The one we are likely to use the most is the `select` service for selecting which input topic currently has control. Let's select the joystick as the control input. Open another terminal and run the command:

```
$ rosservice call mux_cmd_vel/select joystick_cmd_vel
```

Notice how the `select` service name reflects the name of the `mux` node we started earlier, in this case `mux_cmd_vel`. The output on the terminal should be:

```
prev_topic: /move_base_cmd_vel
```

which simply reflects the previous control topic.

You should now be able to control the fake TurtleBot with the joystick. At the same time, you will no longer be able to move the robot by setting navigation goals in `RViz`.

To go back to navigation control using `RViz` and `move_base`, run the service call:

```
$ rosservice call mux_cmd_vel/select move_base_cmd_vel
```

Now the joystick should no longer work to control the robot but setting navigation goals in RViz will.

8.4 A ROS Node to Prioritize `mux` Inputs

Suppose we'd like to automate the switching of control inputs such that the joystick takes priority over `move_base`. One way to do this can be found in the node `select_cmd_vel.py` in the `rbx2_nav/nodes` directory. Our overall strategy is to subscribe to each of the input topics and then use the `mux` select service to select the control input based on the priorities we have set.

Before looking at the code, let's try it out. Make sure you have all the same launch files running as in the previous section, as well as the `mux` node for multiplexing the joystick and `move_base` inputs. Then run our new node:

```
$ rosrun rbt2_nav select_cmd_vel.py
```

```
[INFO] [WallTime: 1395841719.884079] Waiting for mux select service...
[INFO] [WallTime: 1395841719.887139] Connected to mux select service.
[INFO] [WallTime: 1395841719.887343] Ready for input.
```

Now give the robot a navigation goal using RViz and pick a goal location far away from the robot so it takes some time to get there. As the robot is moving toward the goal, override its motion with the joystick. You should find the robot reacts smoothly to your joystick input. As soon as you release the dead man button on the joystick, the robot should head back toward the navigation goal.

Let's now look at the `select_cmd_vel.py` script.

Link to source: [select_cmd_vel.py](#)

```
1 #!/usr/bin/env python
2
3 import rospy
4 from geometry_msgs.msg import Twist
5 from topic_tools.srv import MuxSelect
6 import thread
7
8 class SelectCmdVel:
9     def __init__(self):
10         rospy.init_node("select_cmd_vel")
11
12         # The rate at which to update the input selection
13         rate = rospy.get_param('~rate', 5)
14
15         # Convert to a ROS rate
```

```

16     r = rospy.Rate(rate)
17
18     # Get a lock for updating the selected cmd_vel input
19     self.lock = thread.allocate_lock()
20
21     # Set the default input control
22     self.move_base = True
23     self.joystick = False
24
25     # Track the last input control
26     self.last_joystick = self.joystick
27     self.last_move_base = self.move_base
28
29     # Subscribe to the control topics and set a callback for each
30     rospy.Subscriber('joystick_cmd_vel', Twist, self.joystick_cb)
31     rospy.Subscriber('move_base_cmd_vel', Twist, self.move_base_cb)
32
33     # Wait for the mux select service
34     rospy.loginfo("Waiting for mux select service...")
35     rospy.wait_for_service('mux_cmd_vel/select', 60)
36
37     # Create a proxy for the mux select service
38     mux_select = rospy.ServiceProxy('mux_cmd_vel/select', MuxSelect)
39
40     rospy.loginfo("Connected to mux select service.")
41
42     rospy.loginfo("Ready for input.")
43
44     # Main loop to switch inputs if user move joystick
45     while not rospy.is_shutdown():
46         if self.joystick and self.joystick != self.last_joystick:
47             mux_select('joystick_cmd_vel')
48         elif self.move_base and self.move_base != self.last_move_base:
49             mux_select('move_base_cmd_vel')
50
51         self.last_joystick = self.joystick
52         self.last_move_base = self.move_base
53
54         r.sleep()
55
56     # If the joystick is moved, get the message here
57     def joystick_cb(self, msg):
58         self.lock.acquire()
59         self.joystick = True
60         self.move_base = False
61         self.lock.release()
62
63     # If move_base is active, get the message here
64     def move_base_cb(self, msg):
65         self.lock.acquire()
66         self.joystick = False
67         self.move_base = True
68         self.lock.release()
69
70 if name == 'main':

```

```
71     SelectCmdVel()
72     rospy.spin()
```

Let's break this down line by line:

```
4  from geometry_msgs.msg import Twist
5  from topic_tools.srv import MuxSelect
6  import thread
```

Near the top of the script we first import the `Twist` message type and the `MuxSelect` service type. We also import the `thread` library as we will need to lock our callbacks.

```
19         self.lock = thread.allocate_lock()
```

Here we create a lock to use later on in our subscriber callback functions.

```
21         # Set the default input control
22         self.move_base = True
23         self.joystick = False
24
25         # Track the last input control
26         self.last_joystick = self.joystick
27         self.last_move_base = self.move_base
```

We use a handful of variables to set the initial control mode as well as to track the last control mode used.

```
29         # Subscribe to the control topics and set a callback for each
30         rospy.Subscriber('joystick_cmd_vel', Twist, self.joystick_cb)
31         rospy.Subscriber('move_base_cmd_vel', Twist, self.move_base_cb)
```

Here we subscribe to the two `cmd_vel` topics used with our mux node, one for the joystick and one for move_base. We also set the callback function to point to functions defined later in the script.

```
33         # Wait for the mux select service
34         rospy.loginfo("Waiting for mux select service...")
35         rospy.wait_for_service('mux_cmd_vel/select', 60)
36
37         # Create a proxy for the mux select service
38         mux_select = rospy.ServiceProxy('mux_cmd_vel/select', MuxSelect)
```

Before connecting to the `mux select` service, we wait to see if it is alive. Then we assign a `ServiceProxy` to the variable `mux_select` to be used later.

```

56      # If the joystick is moved, get the message here
57      def joystick_cb(self, msg):
58          self.lock.acquire()
59          self.joystick = True
60          self.move_base = False
61          self.lock.release()
62
63      # If move_base is active, get the message here
64      def move_base_cb(self, msg):
65          self.lock.acquire()
66          self.joystick = False
67          self.move_base = True
68          self.lock.release()

```

Jumping down the script a bit, these are the two callback functions to handle messages received on the `joystick_cmd_vel` topic and the `move_base_cmd_vel` topic. Since ROS subscriber callbacks run in separate threads, we need to use the lock created earlier before setting the `self.joystick` and `self.move_base` flags.

```

44      # Main loop to switch inputs if user move joystick
45      while not rospy.is_shutdown():
46          if self.joystick and self.joystick != self.last_joystick:
47              mux_select('joystick_cmd_vel')
48          elif self.move_base and self.move_base != self.last_move_base:
49              mux_select('move_base_cmd_vel')
50
51          self.last_joystick = self.joystick
52          self.last_move_base = self.move_base
53
54          r.sleep()

```

Finally, we jump back to the main loop where the actual input selection is done. Since we want the joystick to take priority, we check it first. If the `self.joystick` flag is `True` (which means joystick input was detected in the `joystick_cb` function described above), we use the `mux_select` service proxy to set the input control topic to `joystick_cmd_vel`. Otherwise, we give control to the `move_base_cmd_vel` topic.

8.5 The YOCS Controller from Yujin Robot

The folks at [Yujin Robot](#), makers of the [Kobuki](#) robot (a.k.a TurtleBot 2), have created a very useful `mux`-based controller that does everything our `select_cmd_vel.py` node does but makes it easy to add any number of control inputs and set their priorities by using a simple YAML configuration file. The package is called [`yocs_cmd_vel_mux`](#) and you can learn more about how it works on the [Kobuki Control Systems](#) Wiki page.

The `yocs_cmd_vel_mux` package uses a ROS [nodelet](#) (called `CmdVelMuxNodelet`) to handle the `mux` switching between inputs together with a configuration file to specify input topics, their priorities, and the final output topic. The configuration file that would reproduce the functionality of our `select_cmd_vel.py` node looks like this:

```
subscribers:
  - name:      "Joystick control"
    topic:     "/joystick_cmd_vel"
    timeout:   1.0
    priority:  1

  - name:      "Navigation stack"
    topic:     "/move_base_cmd_vel"
    timeout:   1.0
    priority:  0

publisher:  "output/cmd_vel"
```

Here we see that our two input topics are listed under the `subscribers` section. Each subscriber is given a descriptive `name`, the `topic` on which it is listening for commands, a `timeout` (in seconds) and a `priority` where larger numbers have higher priority and must be unique—i.e., two or more subscribers cannot have the same priority. The `output` topic is specified by the `publisher` parameter and in this case is set to `output/cmd_vel`. Note that since the `output` topic does not have a leading `/`, it will be prepended with the name of the nodelet namespace which is `/cmd_vel_mux` so that the full output topic name is `/cmd_vel_mux/output/cmd_vel`.

This configuration can be found in the file [`yocs_cmd_vel.yaml`](#) in the `rbx2_nav/config` directory and will be loaded by the launch file we turn to next.

To run the `CmdVelMuxNodelet`, we use the launch file [`yocs_cmd_vel.launch`](#) in the `rbx2_nav/launch` directory as shown below:

```
<launch>
  <arg name="robot_name" default="turtlebot"/>

  <node pkg="nodelet" type="nodelet" name="$(arg robot_name)_nodelet_manager"
args="manager"/>

  <node pkg="nodelet" type="nodelet" name="cmd_vel_mux"
args="load yocs_cmd_vel_mux/CmdVelMuxNodelet $(arg
robot_name)_nodelet_manager">
    <param name="yaml_cfg_file" value="$(find
rbx2_nav)/config/yocs_cmd_vel.yaml"/>
    <remap from="cmd_vel_mux/output/cmd_vel" to="/cmd_vel"/>
  </node>
</launch>
```

We begin the launch file by using an argument to assign a default name for the robot we are controlling. This can be essentially anything you like. We then load the overall nodelet manager from the ROS nodelet package before we can load the CmdVelMuxNodelet itself. Next we load the CmdVelMuxNodelet nodelet and read in the `yocs_cmd_vel.yaml` configuration file. Finally, we remap the output topic `/cmd_vel_mux/output/cmd_vel` to the standard `/cmd_vel` topic that our fake TurtleBot subscribes to for motion commands.

To test it all out, terminate any nodes and launch files from the previous two sections and start from scratch as follows.

Bring up the fake TurtleBot:

```
$ rosrun roslaunch rbx1 Bringup fake_turtlebot.launch
```

Next, run the `mux_fake_move_base_blank_map.launch` file:

```
$ rosrun roslaunch rbx2_nav mux_fake_move_base_blank_map.launch
```

Assuming you have a joystick attached to your computer, bring up the `mux joystick` node:

```
$ rosrun roslaunch rbx2_nav mux_joystick_teleop.launch
```

Finally, bring up `RViz` with a suitable configuration file:

```
$ rosrun rviz rviz -d `rospack find rbx2_nav`/config/nav.rviz
```

We can now run the `yocs_cmd_vel.launch` file. This takes the place of both the `mux` node we ran earlier and our `select_cmd_vel.py` node:

```
$ rosrun roslaunch rbx2_nav yocs_cmd_vel.launch
```

The output should look something like this:

```
process[turtlebot_nodelet_manager-1]: started with pid [8473]
process[cmd_vel_mux-2]: started with pid [8474]
```

Here we see that the nodelet manager is started followed by the `cmd_vel_mux` nodelet.

You should now be able to control the fake robot with the joystick as well as by setting 2D Nav Goals in `RViz` with your mouse. If you first set a 2D Nav Goal, you should then be able to override the motion of the robot at any time using the joystick since we gave the joystick higher priority than `move_base` in the `yocs_cmd_vel.yaml` config file. Once you release the joystick control, the robot should continue on to the last 2D Nav Goal.

8.5.1 Adding input sources

Configuring the `yocs_cmd_vel` controller to manage additional input sources is very easy since all we need to do is add an appropriate section to the configuration file and run the node that publishes the new input messages. For example, the following configuration includes an input for keyboard teleop control:

```
subscribers:
  - name: "Joystick control"
    topic: "/joystick_cmd_vel"
    timeout: 1.0
    priority: 2

  - name: "Navigation stack"
    topic: "/move_base_cmd_vel"
    timeout: 1.0
    priority: 1

  - name: "Keyboard control"
    topic: "/keyboard_cmd_vel"
    timeout: 1.0
    priority: 0

publisher: "output/cmd_vel"
```

Here we assume that the keyboard teleop node is publishing Twist commands on the `/keyboard_cmd_vel` topic. We have also given it the lowest priority. This configuration can be found in the file [`yocs_cmd_vel_with_keyboard.yaml`](#) in the `rbx2_nav/config` directory. You can try it out by terminating `yocs_cmd_vel.launch` file from the previous section, then running the following two commands:

```
$ roslaunch rbx2_nav mux_keyboard_teleop.launch
```

```
$ roslaunch rbx2_nav yocs_cmd_vel_with_keyboard.launch
```

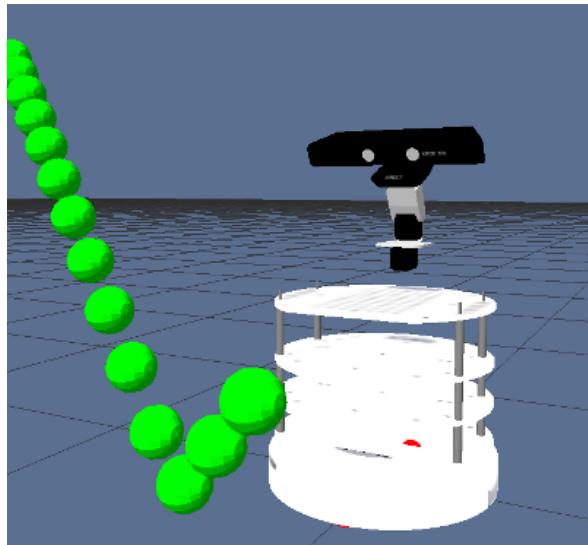
Bring the keyboard teleop terminal to the foreground and try controlling the fake TurtleBot with key commands. Notice that if you set a 2D Nav Goal in `RViz` you can override the robot's motion with the joystick but not the keyboard. This is because we gave the joystick a higher priority than `move_base` in the `yocs` config file but the keyboard was given a lower priority.

NOTE: Most keyboard teleop nodes (including the TurtleBot teleop node we are using in the `mux_keyboard_teleop.launch` file) continue to publish an empty `Twist` message even when all keys have been released. On the one hand, this is a safety feature since it means the robot will stop if the user is not pressing a key. On the other hand, it does not play well with the `yocs_cmd_vel` nodelet since unless the keyboard input is put last in the priority list, it will always override all other inputs below it.

9. HEAD TRACKING IN 3D

In *Volume 1* we learned how to program a head tracking node that moves the camera to follow a face or any other target published on the `/roi` topic using a `RegionOfInterest` message type. In those cases, the target was represented as a ROS `RegionOfInterest` message that defines a rectangular region typically embedded in the plane of the camera view surrounding the object to be tracked.

However, the real world is three dimensional and since depth cameras like the Kinect are relatively inexpensive, there is little reason not to take advantage of the 3rd dimension. What's more, ROS itself was built from the ground up to work in all three dimensions. In particular, the `tf` library has no trouble transforming 3-dimensional sensor data from one reference frame to another which takes all the pain out of computing how an object is space is positioned relative to the robot. (Unless you are particularly good at multiplying quaternions in your head.)



In this chapter we will add depth information to the problem of visual object tracking so that instead of a 2D `RegionOfInterest` message to represent the object, we will use a 3D `PoseStamped` message. For head tracking, we will not need the orientation components of this message type so we could get away with a simpler `PointStamped` message; however, since target orientation will be important when it comes to grasping an object in the chapter on Arm Navigation and MoveIt, we will use the more general `PoseStamped` message here too.

A `PoseStamped` message can represent the location of a face or colored object, but it can also track other points in space such as the location of gripper on the robot's arm that might not even be in view of the camera. (Recall that we always know the location of the gripper or other parts of the robot relative to the camera by way of the robot's URDF model and the `robot_state_publisher`.)

Our challenge is to project this 3D pose onto the camera plane so that we know how to move the camera to face the target. Before looking at the code, we will run a couple of examples using the new head tracker. But first, let's create a node that publishes a fictional moving 3D point that we can use to test our tracking.

9.1 Tracking a Fictional 3D Target

Before worrying about tracking a real object with a real camera, let's create a node that publishes a `PoseStamped` message simulating a moving object so that we can test tracking with an imaginary target first.

The Python script `pub_3d_target.py` found in the `rbx2_utils/nodes` directory does the trick. This node simply publishes a time varying `PoseStamped` message where the x, y and z coordinates of the point vary sinusoidally. Of course, you could modify the script or create your own to generate different motions of the target point. The script also publishes a visualization marker to indicate the location of the target so that we can see it in `RViz`.

Let's try it out using a fake version of Pi Robot:

```
$ roslaunch rbox2_bringup pi_robot_with_gripper.launch sim:=true
```

Next, fire up `RViz` using the `fake_target.rviz` config file that is configured to display the 3D marker:

```
$ rosrun rviz rviz -d `rospack find rbox2_utils`/fake_target.rviz
```

Now launch the `pub_3d_target.py` node:

```
$ roslaunch rbox2_utils pub_3d_target.launch
```

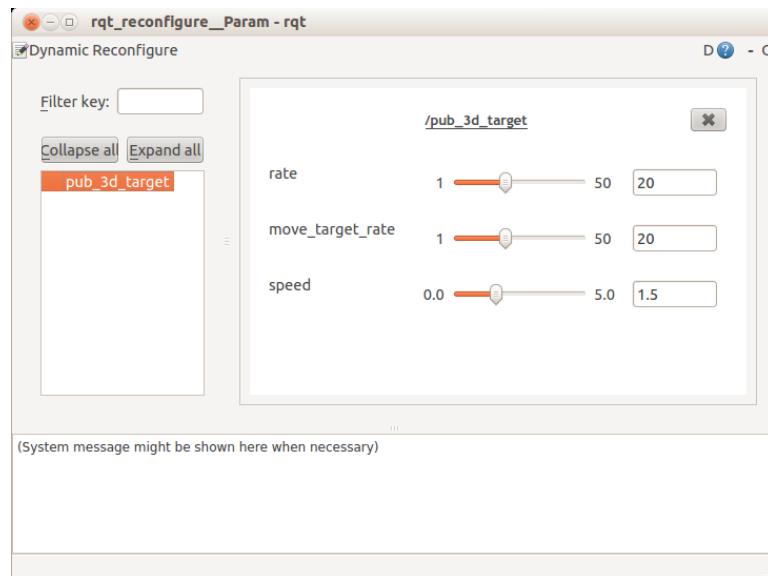
You should see a yellow sphere moving in `RViz` in front of the robot.

Finally, launch the `head_tracker.py` node in fake mode (`sim:=true`):

```
$ roslaunch rbox2_dynamixels head_tracker.launch sim:=true
```

Back in `RViz`, you should see the head track the motion of the yellow sphere. You can change the way the sphere moves using `rqt_reconfigure`:

```
$ rosrun rqt_reconfigure rqt_reconfigure
```



For example, setting the `move_target_rate` to 1 and the speed to 2.0 will cause the sphere to jump over larger gaps instead of moving smoothly in between.

9.2 Tracking a Point on the Robot

A robot running ROS can always determine the 3D position and orientation of each of its links and joints. This is made possible by the URDF model and the `robot_state_publisher` node that together keep the `tf` tree in sync with the robot's current joint states. As a result, we can use our head tracking node to follow any part of the robot as it moves. For example, if the robot is handing an object to someone, we can program the head to track the gripper location as the arm reaches outward toward the person. It turns out that this type of head motion while giving an object to someone else is an important social cue in humans that signals that the receiver should reach out to take the object. See for example the recent study described [here](#).

The position of a point on the robot can be specified relative to any convenient frame of reference. To track a particular link in the robot's URDF model, we can simply use the coordinates (0, 0, 0) in that link's reference frame. So to track the gripper location, the head should follow the point (0, 0, 0) in the gripper frame

The script called `pub_tf_frame.py` in the `rbx2_utils/nodes` directory simply publishes a `PoseStamped` message on the `target_pose` topic with coordinates (0, 0,

0) and orientation (0, 0, 0, 1) relative to the reference frame we want to track. The default target frame in the script is the `right_gripper_link`.

To try it, use all the same launch files as in the previous section but `Ctrl-C` out of the `pub3d_target.launch` file and launch the following `pub_tf_frame.launch` file instead:

```
$ roslaunch rbx2_utils pub_tf_frame.launch
```

The frame we want to track is specified in the launch file by the parameter `target_frame`. By default, this frame is set to '`right_gripper_link`'.

If the head tracker node is not already running, bring it up now:

```
$ roslaunch rbx2_dynamixels head_tracker.launch sim:=true
```

Back in `RViz`, you should see the head track rotate downward and to the right to look at the right gripper.

Finally, bring up the `arbotix_gui` control so that we can manually move the arm:

```
$ arbotix_gui
```

Use the slider controls on the `arbotix_gui` to move the arm joints. As you move the sliders, the arm should move in `RViz` and the head will move to track the position of the right gripper.

Let's now look at the code.

Link to source: [pub_tf_frame.py](#)

```
1 #!/usr/bin/env python
2
3 import rospy
4 from geometry_msgs.msg import PoseStamped
5
6 class PubFrame():
7     def __init__(self):
8         rospy.init_node('pub_tf_frame')
9
10    # The rate at which we publish target messages
11    rate = rospy.get_param('~rate', 20)
12
13    # Convert the rate into a ROS rate
14    r = rospy.Rate(rate)
```

```

15      # The frame we want to track
16      target_frame = rospy.get_param('~target_frame', 'right_gripper_link')
17
18      # The target pose publisher
19      target_pub = rospy.Publisher('target_pose', PoseStamped)
20
21      # Define the target as a PoseStamped message
22      target = PoseStamped()
23
24      target.header.frame_id = target_frame
25
26      target.pose.position.x = 0
27      target.pose.position.y = 0
28      target.pose.position.z = 0
29
30      target.pose.orientation.x = 0
31      target.pose.orientation.y = 0
32      target.pose.orientation.z = 0
33      target.pose.orientation.w = 1
34
35      rospy.loginfo("Publishing target on frame " + str(target_frame))
36
37      while not rospy.is_shutdown():
38          # Get the current timestamp
39          target.header.stamp = rospy.Time.now()
40
41          # Publish the target
42          target_pub.publish(target)
43
44          r.sleep()
45
46 if __name__ == '__main__':
47     try:
48         target = PubFrame()
49         rospy.spin()
50     except rospy.ROSInterruptException:
51         rospy.loginfo("Target publisher is shut down.")

```

As you can see, the script is fairly straightforward. After reading in the publishing rate parameter, we get the target frame as a parameter that defaults to `right_gripper_link` and can be changed in the launch file. Then we define a publisher for the `target_pose` topic. We then set the `PoseStamped` target to be the origin of the target frame by giving it position coordinates (0, 0, 0) and orientation values (0, 0, 0, 1). Finally, we enter a loop to publish this pose with a new timestamp on each cycle.

You might be wondering, if we are publishing the same pose every time, how can the head tracker know that the link is actually moving. The answer is that the head tracker script uses the `tf transformPoint()` function to lookup the latest transformation

between the camera frame (attached to the head) and the `target_pose` frame (attached to the right hand in this case). So although the target coordinates are the same relative to the target frame, the target frame itself is moving relative to the rest of the robot.

It is instructive to try different coordinates in the `pub_tf_frame.py` script. For example, try `target.pose.position.x = 0.1`, then terminate and restart the `pub_tf_frame.launch` file and the robot should now look 10cm outward from the gripper link. You can imagine that this might become useful when the robot is grasping a tool or other object so that the camera view can be centered on the object.

9.3 The 3D Head Tracking Node

Our new head tracking node can be found in the file [`head_tracker.py`](#) in the directory `rbx2_dynamixel/nodes`. This code is similar to the script we use in *Volume 1* (`rbx1_dynamixels/nodes/head_tracker.py`) so we will concentrate only on what is new. The main difference is that we now subscribe to a topic with message type `PoseStamped` instead of a `RegionOfInterest` and we use a new callback function to compute the pan and tilt updates for the head servos. We also include support for both the [`dynamixel_motor`](#) package and the [`arbotix`](#) package. In *Volume 1*, we used the `dynamixel_motor` package with real servos whereas in this volume we are using the arbotix package.

9.3.1 Real or fake head tracking

In the chapter on URDF models, we used the [`arbotix_python`](#) package in fake mode to test the movement of simulated Dynamixel servo joints. When using the fake simulator for head tracking, it turns out that our speed tracking technique causes the head to oscillate around the target. This is because the fake camera, servos and other robot components have no inertia to dampen the motion. Consequently, we need to use position tracking rather than speed tracking when using the fake robot.

Position tracking uses a fixed servo speed and aims the camera right at the target point. On the other hand, recall that speed tracking aims the camera ahead of the target but adjusts the speed to be proportional to the displacement of the target from the center of the field of view. The use of position tracking with real hardware causes jerky motion of the servos which is why we use speed tracking instead. However, for the fake robot, position tracking works fine.

Near the top of head tracking script, you will see the following lines:

```
# Are we running in fake mode?  
FAKE = rospy.get_param('~sim', False)  
  
# For fake mode we use the arbotix controller package and position tracking
```

```

if FAKE:
    CONTROLLER_TYPE = 'arbotix'
    TRACKER_TYPE = 'position'
else:
    # Specify either 'dynamixel_motor' or 'arbotix' for the controller package
    CONTROLLER_TYPE = rospy.get_param('~controller_type', 'arbotix')

    # Specify either 'speed' or 'position' for the type of tracking
    TRACKER_TYPE = rospy.get_param('~tracker_type', 'speed')

```

By setting the parameter `sim` to `True` in the `head_tracker.launch` file, we can run head tracking in the ArbotiX simulator using position tracking. When using real servos, set this parameter to `False` and speed tracking will be used instead.

Further down in the `head_tracker.py` script, the controller type and tracker type determines the callback we run on the `PoseStamped` target type:

```

if CONTROLLER_TYPE == 'arbotix' and TRACKER_TYPE == 'position':
    rospy.Subscriber('target_topic', PoseStamped, self.update_joint_positions)

else:
    rospy.Subscriber('target_topic', PoseStamped, self.update_joint_speeds)

```

As you can see, for fake head tracking (`controller_type='arbotix'` and `tracker_type='position'`) we use the `update_joint_position()` callback whereas for real tracking we use the `update_joint_speeds()` callback. Note how we use the generic topic name '`target_topic`' which can then be remapped in the launch file to the specific topic that we want to track.

9.3.2 Projecting the target onto the camera plane

Whether we are doing real or fake head tracking, we need to project the 3D target position onto the camera plane so we know how far to pan and tilt the camera. Let's use the `update_joint_speeds()` callback to examine the process.

The `update_joint_speeds()` callback looks like this:

```

1 def update_joint_speeds(self, msg):
2     # Acquire the lock
3     self.lock.acquire()
4
5     try:
6         # If message is empty, return immediately
7         if msg == PointStamped():
8             return
9

```

```

10     # If we get this far, the target is visible
11     self.target_visible = True
12
13     # Get position component of the message
14     target = PointStamped()
15     target.header.frame_id = msg.header.frame_id
16     target.point = msg.pose.position
17
18     # Project the target point onto the camera link
19     camera_target = self.tf.transformPoint(self.camera_link, target)
20
21     # The virtual camera image is in the y-z plane
22     pan = -camera_target.point.y
23     tilt = -camera_target.point.z
24
25     # Compute the distance to the target in the x direction
26     distance = float(abs(camera_target.point.x))
27
28     # Convert the pan and tilt values from meters to radians
29     # Check for exceptions (NaNs) and use minimum range as fallback
30     try:
31         pan /= distance
32         tilt /= distance
33     except:
34         pan /= 0.5
35         tilt /= 0.5
36
37     # Get the current pan and tilt position
38     try:
39         current_pan =
self.joint_state.position[self.joint_state.name.index(self.head_pan_joint)]
40         current_tilt =
self.joint_state.position[self.joint_state.name.index(self.head_tilt_joint)]
41     except:
42         return
43
44     # Pan camera only if target displacement exceeds the threshold
45     if abs(pan) > self.pan_threshold:
46         # Set pan speed proportion to horizontal displacement
47         self.pan_speed = trunc(min(self.max_joint_speed,
max(self.min_joint_speed, self.gain_pan * abs(pan))), 2)
48
49         if pan > 0:
50             self.pan_position = max(self.min_pan, current_pan -
self.lead_target_angle)
51         else:
52             self.pan_position = min(self.max_pan, current_pan +
self.lead_target_angle)
53     else:
54         self.pan_position = current_pan
55         self.pan_speed = self.min_joint_speed
56
57     # Tilt camera only if target displacement exceeds the threshold
58     if abs(tilt) > self.tilt_threshold:
59         # Set tilt speed proportion to vertical displacement

```

```

60             self.tilt_speed = trunc(min(self.max_joint_speed,
max(self.min_joint_speed, self.gain_tilt * abs(tilt))), 2)
61
62         if tilt < 0:
63             self.tilt_position = max(self.min_tilt, current_tilt -
self.lead_target_angle)
64         else:
65             self.tilt_position = min(self.max_tilt, current_tilt +
self.lead_target_angle)
66
67     else:
68         self.tilt_position = current_tilt
69         self.tilt_speed = self.min_joint_speed
70
71 finally:
72     # Release the lock
73     self.lock.release()

```

Let's look at the key lines of the callback function.

```
3     self.lock.acquire()
```

First we acquire a lock at the beginning of the callback. This is to protect the variables `self.pan_speed`, `self.tilt_speed` and `self.target_visible` which are also modified in our main loop.

```

14     target = PointStamped()
15     target.header.frame_id = msg.header.frame_id
16     target.point = msg.pose.position

```

Before projecting the `PoseStamped` message into the camera plane, we extract the position component since we do not require the orientation of the target for this process.

```
19     camera_target = self.tf.transformPoint(self.camera_link, target)
```

Here we use the `transformPoint()` function from the `tf` library to project the `PointStamped` target onto the camera frame. (A good reference for other transformation functions can be found on the ROS Wiki on the page [Using Python tf](#).)

```

21     # The virtual camera image is in the y-z plane
22     pan = -camera_target.point.y
23     tilt = -camera_target.point.z
24
25     # Compute the distance to the target in the x direction
26     distance = float(abs(camera_target.point.x))

```

Recall that the ROS camera frame uses y for the horizontal (left/right) axis and z for the vertical (up/down) axis. So we initially set the `pan` variable opposite to the displacement of the target in the y direction and similarly for the `tilt` variable in the z direction. The camera's x axis points perpendicular to the plane of the camera and therefore represents the `distance` of the target to the camera.

```
30     try:
31         pan /= distance
32         tilt /= distance
33     except:
34         pan /= 0.5
35         tilt /= 0.5
```

We need to convert the pan and tilt values to angular displacements (in radians) which we can do by dividing each by the distance to the target. In case the distance is zero, we set it to 0.5 meters which is roughly the minimum distance a Kinect or Xtion Pro can reliably measure.

The rest of the script is essentially the same as the head tracking script we used in *Volume 1* and so will not be described further here.

In the next chapter we will make use of this new head tracker node to locate special patterns called AR tags in 3-dimensional space.

9.4 Head Tracking with Real Servos

If you have a pan-and-tilt head using Dynamixel servos, you can try out the 3D head tracking for real. We will use the `pi_robot_head_only.launch` file that loads the `pi_robot_head_only.yaml` configuration file and runs the `arbotix` node to connect to two AX-12 Dynamixel servos that are assumed to have hardware IDs of 1 for the pan servo and 2 for the tilt servo. Use your own launch file if you already have one.

The `pi_robot_head_only.launch` file loads the model of the Pi Robot without an arm but with a Kinect mounted on a set of pan-and-tilt servos. We will run through two scenarios: the first uses the fake 3D target introduced in the previous sections. This way we can test tracking even without a depth camera. The second assumes you have a depth camera on top of the pan-and-tilt servos and then we will track the nearest real object in front of the camera.

9.4.1 Real servos and fake target

We can test the functioning of the servos by publishing the fictitious 3D target we used earlier to give the real head something to track, even though the target will only be in the robot's imagination, so to speak.

First, terminate any launch files you might already have running from previous sections.

Next, make sure your servos are connected and powered up, then run the `pi_robot_head_only.launch` file from the `rbx2_bringup` package. Note that we set the `sim` parameter to `false` since we are using a real robot. Alternatively, use your own startup file if you have one:

```
$ rosrun rbt2_bringup pi_robot_head_only.launch sim:=false \
port:=/dev/ttyUSB0
```

The launch file assumes your USB2Dynamixel controller is on `/dev/ttyUSB0` but you can use the port parameter as shown above to change the port if your controller has a different USB ID, e.g. `/dev/ttyUSB1`.

If all goes well, you should eventually see the `rqt_robot_monitor` window appear displaying the status of the pan and tilt servos under the Joints category.

Next bring up the Dynamixel monitoring node we created in the chapter on Diagnostics. Remember that this node helps prevent the servos from overheating by disabling all servos if any one of them exceeds 60 degrees C.

```
$ rosrun rbt2_diagnostics monitor_dynamixels.launch
```

Fire up RViz with the `fake_target.rviz` config file:

```
$ rosrun rviz rviz -d `rospack find rbt2_utils`/fake_target.rviz
```

Now start the fake 3D target moving:

```
$ rosrun rbt2_utils pub_3d_target.launch
```

The moving yellow "balloon" should appear in RViz.

Finally, start the head tracker node, but this time with the `sim` parameter set to `false`:

```
$ rosrun rbt2_dynamixels head_tracker.launch sim:=false
```

If all goes right, your pan-and-tilt head should begin tracking the imaginary balloon. In the meantime, the robot's virtual head in RViz should reflect the real servo motion.

As we did in the earlier demo, bring up `rqt_reconfigure` and try changing the motion parameters of the target:

```
$ rosrun rqt_reconfigure rqt_reconfigure
```

To stop the tracking and center the servos, simply type `Ctrl-C` in the terminal window you used to launch the head tracking node.

9.4.2 Real servos, real target

Assuming everything went well when tracking the fictitious target in the previous section, we can now try tracking a real target using a depth camera mounted to the pan-and-tilt servos. Rather than tracking a specific type of object like a face, we will use a node that computes the center-of-gravity (COG) of the nearest point cloud in front of the camera. We used the same technique in *Volume 1* for the person follower application that allowed a mobile robot equipped with a depth camera to follow the nearest "blob" in front of the camera.

The script that does the work is called `nearest_cloud.py` in the `rbx2_vision/nodes` directory and we will describe it shortly. However, let's first give it a try.

We'll assume you still have the launch files running from the previous section to connect to your pan-and-tilt servos. If not, run those files now.

Next, `Ctrl-C` out of the `pub3d_target.launch` file if it is still running. The yellow sphere should stop moving in RViz.

Next, bring up the OpenNI node for your depth camera. For example:

For the Microsoft Kinect:

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

For the Asus Xtion, Xtion Pro, or Primesense 1.08/1.09 cameras:

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

Now bring up the nearest cloud node using the corresponding launch file:

```
$ roslaunch rbx2_vision nearest_cloud.launch
```

This launch file runs a `VoxelGrid` filter and two `PassThrough` filters to restrict the point cloud processing to a box in front of the camera that is 0.6 meters wide (about a foot on either side), extends outward to 1.0 meters (about 3 feet), and runs 0.3 meters (about 1 foot) below the camera and 0.15 meters (6 inches) above the camera. This creates a kind of "focus of attention" for the robot so that we can track objects that are relatively near the camera. The resulting point cloud is output on the topic called `/cloud_filtered`.

Next, exit out of `RViz` if it is still running then fire it up again with the `track_pointcloud.rviz` config file:

```
$ rosrun rviz rviz -d `rospack find \`rbx2_vision`/config/track_pointcloud.rviz`
```

This configuration file includes a **PointCloud2** display that is subscribed to the `/cloud_filtered` topic. Now move your hand or some object in front of the camera and you should see the corresponding point cloud in `RViz`. Notice how your hand or other object will appear and disappear as you move it in and out of the boundaries defined by the `PassThrough` filters.

Finally, if the head tracker node is not already running, start it now with the `sim` argument set to `false`:

```
$ roslaunch rbx2_dynamixels head_tracker.launch sim:=false
```

After a brief delay, the pan-and-tilt head should begin tracking the nearest object in front of the camera. Keep in mind that both the Kinect and the Xtion Pro cannot see the depth objects closer than about 0.5 meters (a little less than 2 feet). Also remember that the `nearest_cloud.launch` file filters out points that are further away than 1 meter. (You can run `rqt_reconfigure` and adjust the parameters for the `VoxelGrid` and `PassThrough` filters on the fly, or change them in the launch file.)

If you stand in front of the camera within the filter box and move left or right, the camera should track your motion. If you move your body back beyond 1 meter from the camera and then extend your hand into the filter box, the head and camera should track your hand. If you are in doubt about what the camera is detecting, make sure you can see `RViz` on your computer screen so you can monitor the point cloud image.

9.4.3 The `nearest_cloud.py` node and launch file

The launch file `nearest_cloud.launch`, found in the `rbx2_vision/launch` directory, runs a `VoxelGrid` PCL nodelet and two `PassThrough` nodelets to filter the point cloud from a depth camera so that only the points within a relatively small box in front of the camera are retained. This launch file is similar to the `follower2.launch` file we used in *Volume I* (and located in the `rbx1_apps/launch` directory) that we used to enable a mobile robot to follow a person moving in front of the robot's depth camera.

After filtering the point cloud, the `nearest_cloud.launch` file runs the `nearest_cloud.py` node that then computes the center-of-gravity (COG) of the filtered points and publishes these coordinates as a `PoseStamped` message with unit orientation on the topic `/target_pose`. Let's take a look at that script now.

Link to source: [nearest_cloud.py](#)

```
1 #!/usr/bin/env python
2
3 import rospy
4 from roslib import message
5 from sensor_msgs import point_cloud2
6 from sensor_msgs.msg import PointCloud2
7 from geometry_msgs.msg import Point, PoseStamped
8 import numpy as np
9
10 class NearestCloud():
11     def __init__(self):
12         rospy.init_node("nearest_cloud")
13
14         # Define the target publisher
15         self.target_pub = rospy.Publisher('target_pose', PoseStamped,
queue_size=5)
16
17         rospy.Subscriber('point_cloud', PointCloud2, self.get_nearest_cloud)
18
19         # Wait for the pointcloud topic to become available
20         rospy.wait_for_message('point_cloud', PointCloud2)
21
22     def get_nearest_cloud(self, msg):
23         points = list()
24
25         # Get all the points in the visible cloud (may be prefiltered by other
nodes)
26         for point in point_cloud2.read_points(msg, skip_nans=True):
27             points.append(point[:3])
28
29         # Convert to a numpy array
30         points_arr = np.float32([p for p in points]).reshape(-1, 1, 3)
31
32         # Compute the COG
```

```

33     cog = np.mean(points_arr, 0)
34
35     # Abort if we get an NaN in any component
36     if np.isnan(np.sum(cog)):
37         return
38
39     # Store the COG in a ROS Point object
40     cog_point = Point()
41     cog_point.x = cog[0][0]
42     cog_point.y = cog[0][1]
43     cog_point.z = cog[0][2]
44
45     # Give the COG a unit orientation and store as a PoseStamped message
46     target = PoseStamped()
47     target.header.stamp = rospy.Time.now()
48     target.header.frame_id = msg.header.frame_id
49     target.pose.position = cog_point
50     target.pose.orientation.w = 1.0
51
52     # Publish the PoseStamped message on the /target_pose topic
53     self.target_pub.publish(target)
54
55 if __name__ == '__main__':
56     try:
57         NearestCloud()
58         rospy.spin()
59     except rospy.ROSInterruptException:
60         rospy.loginfo("Nearest cloud node terminated.")
61

```

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.

```

15     self.target_pub = rospy.Publisher('target_pose', PoseStamped)

```

We will publish the COG of the nearest point cloud on the topic `/target_pose` as a `PoseStamped` message. This way we can use our head tracker script out of the box to track the cloud.

```

17     rospy.Subscriber('point_cloud', PointCloud2, self.get_nearest_cloud)

```

Next we subscribe to the generic topic name 'point_cloud' and set the callback to the function `get_nearest_cloud()` that we will look at next. The launch file `nearest_cloud.launch` remaps the `point_cloud` topic to the `cloud_filtered` topic that is output by the chain of `VoxelGrid` and `PassThrough` filters we use to form the filter box.

```

22     def get_nearest_cloud(self, msg):
23         points = list()
24
25         # Get all the points in the visible cloud (may be prefiltered by other
26         # nodes)
27         for point in point_cloud2.read_points(msg, skip_nans=True):
28             points.append(point[:3])

```

Here we begin the callback function that fires whenever we receive a point cloud message on the `cloud_filtered` topic. First we initialize a list called `points` that we will use to store the points in the point cloud. Then loop through all the points in the incoming message using the `read_points()` function from the `point_cloud2` library that we imported at the top of the script. The first three values of each point gets appended to the `points` list. These first three values are the x, y and z coordinates of the point relative to the camera depth frame.

```

29     # Convert to a numpy array
30     points_arr = np.float32([p for p in points]).reshape(-1, 1, 3)
31
32     # Compute the COG
33     cog = np.mean(points_arr, 0)
34
35     # Abort if we get an NaN in any component
36     if np.isnan(np.sum(cog)):
37         return

```

Next we convert the `points` list into a numpy array so that we can quickly compute the COG using the numpy `mean()` function. It is also a good idea to test for NaNs (not a value) that can sometimes occur in point clouds returned from depth cameras.

```

39     # Store the COG in a ROS Point object
40     cog_point = Point()
41     cog_point.x = cog[0][0]
42     cog_point.y = cog[0][1]
43     cog_point.z = cog[0][2]

```

To prepare the COG for publishing on the `/target_pose` topic, we first store the x, y and z values as the components of a ROS `Point` object.

```

45     # Give the COG a unit orientation and store as a PoseStamped message

```

```
46     target = PoseStamped()
47     target.header.stamp = rospy.Time.now()
48     target.header.frame_id = msg.header.frame_id
49     target.pose.position = cog_point
50     target.pose.orientation.w = 1.0
```

We then turn the COG into a `PoseStamped` message by adding a timestamp and the `frame_id` of the incoming point cloud message as well as a unit orientation.

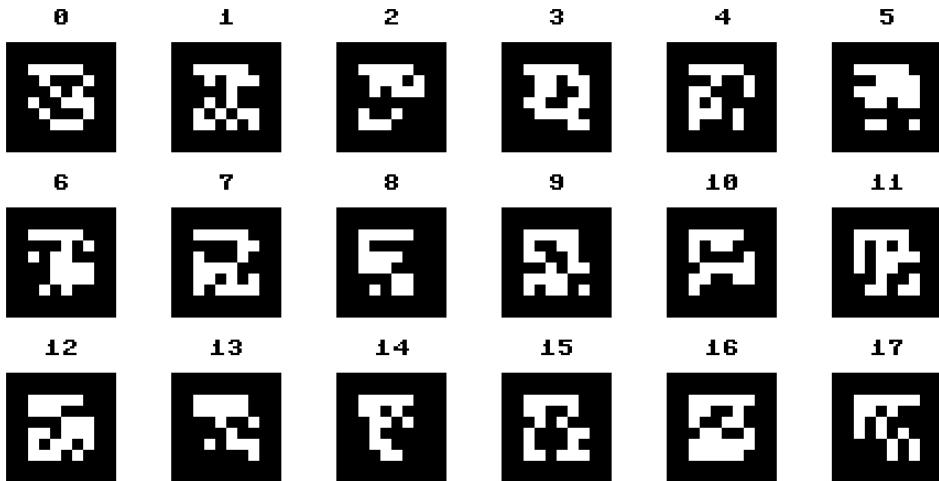
```
53     self.target_pub.publish(target)
```

Finally, we publish the message on the `/target_pose` topic.

10. DETECTING AND TRACKING AR TAGS

IMPORTANT NOTE: Earlier versions of the `ar_track_alvar` package worked only on 64-bit installations of Ubuntu. If you have a 32-bit system, you can try the Indigo Debian package to see if it now works. If not, you can try a source install as described below.

Even when using state-of-the art computer vision algorithms, programming a robot to reliably detect and recognize objects in a natural setting is an ongoing challenge. The next best thing to recognizing objects themselves is to tag them in some way that makes identification easier.



It turns out that certain kinds of checkerboard-like patterns like those shown above are particularly easy to recognize under different lighting conditions, viewing angles, and distances. These patterns are called *fiducial markers* or *AR tags*. AR stands for "augmented reality" because such tags can be added to objects and detected in video recordings so that animations or other artificial labels (such as the name of the object) can be overlaid on the video in place of the tag.

AR tags can also be used for robot localization by placing a number of tags on walls or ceilings in such a way that the robot can always compute its position and orientation relative to one or more tags.

In this chapter, we will learn how to use the [ar_track_alvar](#) ROS package for detecting and tracking AR tags. We will also develop a "follow me" node that enables a mobile robot to follow a moving AR tag.

10.1 Installing and Testing the `ar_track_alvar` Package

Install the Debian Ubuntu packages for `ar_track_alvar` as follows:

```
$ sudo apt-get install ros-indigo-ar-track-alvar*
```

If you need to do a source install on a 32-bit system, follow these steps:

```
$ cd ~/catkin_ws/src  
$ git clone -b indigo-devel \  
https://github.com/sniekum/ar_track_alvar.git  
$ cd ~/catkin_ws  
$ catkin_make  
$ rospack profile
```

10.1.1 Creating your own AR Tags

The `rbx2` repository includes a package called `rbx2_ar_tags`. We will create a few AR tags and store them in the `data` subdirectory of this package. Of course, you can also store these tags in a package of your own choosing.

First move into the tags directory:

```
$ roscd rbx2_ar_tags/data
```

Now run the `createMarker` utility included in the `ar_track_alvar` package. The `createMarker` command takes an ID argument from 0-65535 specifying the particular pattern we want to generate. Let's start with ID 0:

```
$ rosrun ar_track_alvar createMarker 0
```

This command will create a file called `MarkerData_0.png` in the current directory. You can view the image using a utility such as `eog` ("eye of Gnome"):

```
$ eog MarkerData_0.png
```

should result in the following image:



If you run the `createMarker` utility without arguments, you will see a menu of options:

```
$ rosrun ar_track_alvar createMarker
```

```
Usage:  
/opt/ros/Indigo/lib/ar_track_alvar/createMarker [options] argument  
  
65535           marker with number 65535  
-f 65535        force hamming(8,4) encoding  
-l "hello world" marker with string  
-2 catalog.xml  marker with file reference  
-3 www.vtt.fi   marker with URL  
-u 96            use units corresponding to 1.0 unit per 96 pixels  
-uin            use inches as units (assuming 96 dpi)  
-ucm            use cm's as units (assuming 96 dpi) <default>  
-s 5.0          use marker size 5.0x5.0 units (default 9.0x9.0)  
-r 5            marker content resolution -- 0 uses default  
-m 2.0          marker margin resolution -- 0 uses default  
-a              use ArToolkit style matrix markers  
-p              prompt marker placements interactively from the user
```

Of the options listed, you will probably make most use of the `-s` parameter to adjust the size (default is 9 x 9 units). You can also use the `-u` parameter to change the measurement units (the default is cm). So to make a marker with ID 7 that is 5cm x 5cm big, use the command:

```
$ rosrun ar_track_alvar createMarker -s 5 7
```

NOTE: the space between `-s` and `5` above is required.

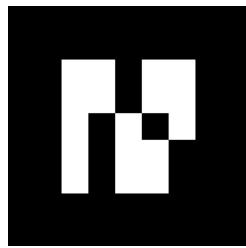
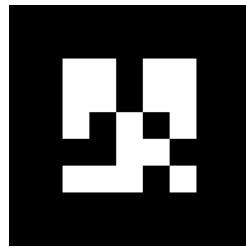
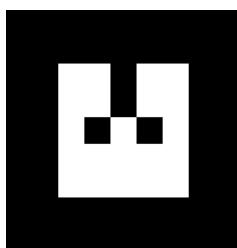
Larger tags are useful for navigation and localization since they are more easily recognized from a greater distance. Smaller tags can be used to label objects that will be viewed at a closer range.

10.1.2 Generating and printing the AR tags

Let's create three small AR tags to use for testing:

```
$ roscd rbx2_ar_tags/data
$ rosrun ar_track_alvar createMarker -s 5 0
$ rosrun ar_track_alvar createMarker -s 5 1
$ rosrun ar_track_alvar createMarker -s 5 2
```

The three generated markers should look like this:



Next we need to print out the markers. You can do so either one at a time or bring all three into an image editor like Gimp and print them on one sheet. You can find all three on a single graphic in the file `Markers_0_2.png` in the `rbx2_ar_tags/config` directory. Print this file so that we can test all three markers at once.

10.1.3 Launching the camera driver and ar_track_alvar node

The `ar_track_alvar` package can utilize depth data from an RGB-D camera to assist in the detection and localization of a marker. We will therefore assume we have a Kinect or Xtion Pro Live camera. Bring up the appropriate camera driver now:

For the Microsoft Kinect:

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

For the Asus Xtion, Xtion Pro, or Primesense 1.08/1.09 cameras:

```
$ rosrun openni2_launch openni2.launch
```

Next, launch the AR node using the file `ar_indiv_kinect.launch` in the `rbx2_ar_tags` package:

```
$ rosrun rbx2_ar_tags ar_indiv_kinect.launch
```

You should see the following INFO messages on the screen:

```
process[ar_track_alvar-1]: started with pid [25697]
[ INFO] [1380290350.331876962]: Subscribing to info topic
[ INFO] [1380290351.344021916]: Subscribing to image topic
```

Before going any further, let's look at the `ar_indiv_kinect.launch` file.

```
<launch>
  <arg name="marker_size" default="6.6" />
  <arg name="max_new_marker_error" default="0.08" />
  <arg name="max_track_error" default="0.05" />

  <arg name="cam_image_topic" default="/camera/depth_registered/points" />
  <arg name="cam_info_topic" default="/camera/rgb/camera_info" />
  <arg name="output_frame" default="/camera_link" />

  <node name="ar_track_alvar" pkg="ar_track_alvar" type="individualMarkers"
respawn="false" output="screen" args="$(arg marker_size) $(arg
max_new_marker_error) $(arg max_track_error) $(arg cam_image_topic) $(arg
cam_info_topic) $(arg output_frame)" />
</launch>
```

Note that we have set the marker size to 6.6 even though we created the markers with a size parameter of 5cm. Depending on your printer, this size discrepancy is to be expected but the number we want to use in the launch file is the size you actually *measure* on the printout. In my case, the three markers were each 6.6cm on a side.

The `max_new_marker_error` and `max_track_error` are described on the [ar_track_alvar](#) Wiki page and the values given above generally work fairly well. Here is the description from the Wiki:

- `max_new_marker_error` (double) – A threshold determining when new markers can be detected under uncertainty
- `max_track_error` (double) – A threshold determining how much tracking error can be observed before an tag is considered to have disappeared

These parameters can also be changed dynamically using `rqt_reconfigure`.

Returning to the launch file above, the `cam_image_topic` and `cam_info_topic` are what we would expect when using the OpenNI driver. The `output_frame` is a useful parameter to note. By default, the frame is set to the `/camera_link` frame; however, if your camera is part of a robot, then you can use any other frame on the robot here. For example, if you use the `/base_link` frame, then tag poses will be published relative to the base frame. This feature will be used in our "follow me" script later in the chapter.

10.1.4 Testing marker detection

Now that we have the `ar_track_alvar` node running, hold up the test printout in front of the camera. You should see a series of messages similar to the following in the terminal window used to launch the `ar_track_alvar` node:

```
-----  
***** ID: 0  
***** ID: 1  
***** ID: 2  
-----  
***** ID: 0  
***** ID: 1  
***** ID: 2
```

These messages indicate that the `ar_track_alvar` node has detected three markers with ID's 0, 1 and 2. Try covering one of the markers with your hand and you should see only two IDs displayed.

10.1.5 Understanding the `/ar_pose_marker` topic

The `ar_track_alvar` node publishes the ID and pose of each detected marker on the `/ar_pose_marker` topic. Each message on this topic consists of a header and an array of markers. Each marker has message type `AlvarMarker` whose fields can be seen by issuing the command:

```
$ rosmsg show ar_track_alvar_msgs/AlvarMarker
```

```
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
uint32 id
uint32 confidence
geometry_msgs/PoseStamped pose
  std_msgs/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 message is quite straightforward including a header, the marker's ID, a confidence value (not currently implemented), and the estimated pose of the marker relative to the output frame listed in the top level header `frame_id`.

To see the marker poses currently being detected, hold the test printout up to the camera and echo the `/ar_pose_marker` topic:

```
$ rostopic echo /ar_pose_marker
```

Remember, that the message is an array of markers, so to view the data for a single marker, use an array index like this:

```
$ rostopic echo /ar_pose_marker/markers[0]
```

(**NOTE:** This command will fail with an error unless you are already holding the test markers in front of the camera. If this happens to you, make sure you hold the printout in front of the camera before running the command. Note also that the array index does not correspond to the marker ID. For example, index 0 might hold the data for marker ID 2, so do not rely on the indexes to identify the markers.)

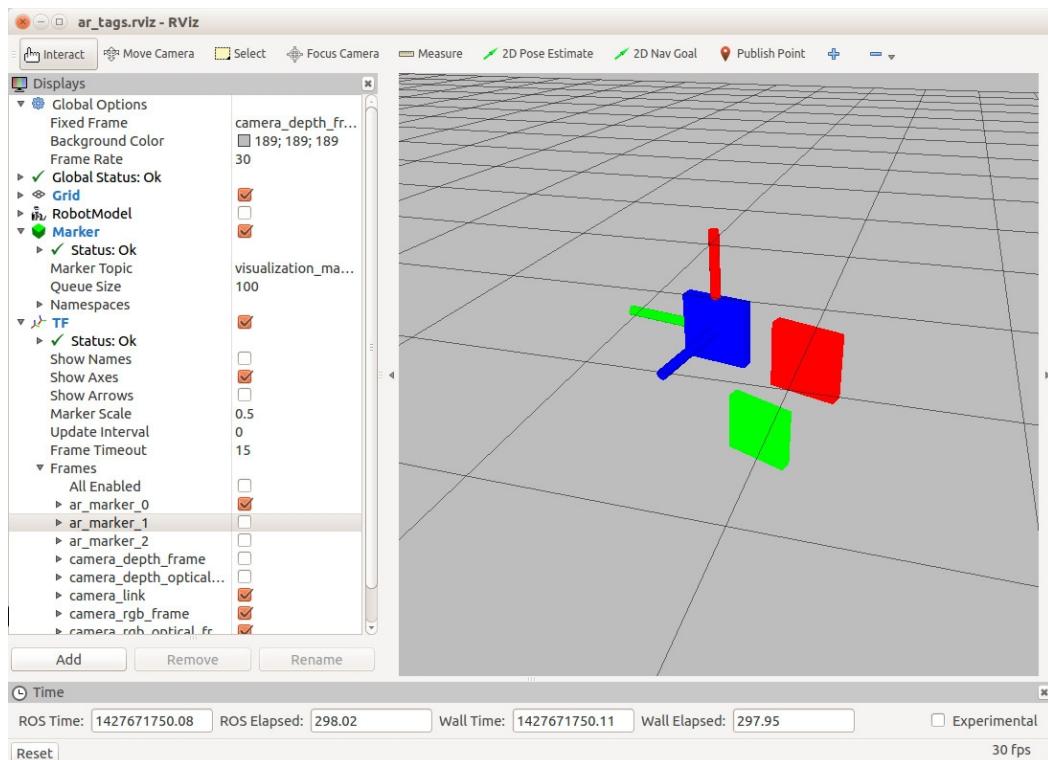
As you move the markers around in front of the camera, you should see the position and orientation values changes accordingly. Note that you can rotate the markers as much as you like and even tilt the printout so that the camera is viewing the tags on an oblique angle, yet tracking should still be successful for most poses of the targets.

10.1.6 Viewing the markers in RViz

The `ar_track_alvar` node publishes a visualization marker for each detected tag on the topic `/visualization_markers` so we can view the markers in `RViz` while tracking. Bring up `RViz` with the included launch file:

```
$ rosrun rviz rviz -d `rospack find rbx2_ar_tags`/ar_tags.rviz
```

If you now hold the test printout in front of the camera, you should see all three markers appear in `RViz` in the same relative position and orientation as they appear on the page. Try rotating and moving the page and the markers should move accordingly. If you cover a tag with your hand, its should disappear from `RViz`. The image below shows the view when all three markers are detected. We have also displayed the reference axes for one of the markers as computed by the `ar_track_alvar` node.



10.2 Accessing AR Tag Poses in your Programs

As we have seen, the `ar_track_alvar` node publishes the poses of detected tags on the topic `/ar_pose_marker`. So we only need to subscribe to this topic in our own nodes to access the poses of each marker.

You can find an example of how to do this in the script `ar_tags_cog.py` in the directory `rbx2_ar_tags/nodes`. This node simply computes the combined COG of all detected tags and publishes the result as a `PoseStamped` message on the topic `/target_pose`. We will then use the 3D head tracking node we created in the previous chapter to track the tags.

10.2.1 The `ar_tags_cog.py` script

Before looking at the code, let's try it out. If you are not still running the camera node and the AR detection node, bring them up now:

For the Microsoft Kinect:

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

For the Asus Xtion, Xtion Pro, or Primesense 1.08/1.09 cameras:

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

Then the `ar_track_alvar` node:

```
$ roslaunch rbx2_ar_tags ar_indiv_kinect.launch
```

Now open another terminal and run the `ar_tags_cog.launch` file:

```
$ roslaunch rbx2_ar_tags ar_tags_cog.launch
```

You should see a startup message similar to the following:

```
[INFO] [WallTime: 1385342429.874338] Publishing tag COG on topic /target_pose...
```

Next, open another terminal and monitor the messages on the `/target_pose` topic:

```
$ rostopic echo /target_pose
```

Finally, hold the test marker printout in front of the camera and you should start seeing pose messages appear in the `/target_pose` topic window.

The `ar_tags_cog.launch` file runs the `ar_tags_cog.py` node and sets an optional list of valid IDs:

```
<launch>
  <node pkg="rbx2_ar_tags" name="ar_tags_cog" type="ar_tags_cog.py"
output="screen">
  <rosparam>
    tag_ids: [0, 1, 2]
  </rosparam>
</node>
</launch>
```

In this case, we have indicated that we only want to track IDs 0, 1 and 2. Any other tag IDs detected will be ignored by the `ar_tags_cog.py` node as we shall now see.

Link to source: [ar_tags_cog.py](#)

```
1  #!/usr/bin/env python
2
3  import rospy
4  from geometry_msgs.msg import Point, PoseStamped
5  from ar_track_alvar.msg import AlvarMarkers
6
7  class TagsCOG():
8      def __init__(self):
9          rospy.init_node("ar_tags_cog")
10
11         # Read in an optional list of valid tag ids
12         self.tag_ids = rospy.get_param('~tag_ids', None)
13
14         # Publish the COG on the /target_pose topic as a PoseStamped message
15         self.tag_pub = rospy.Publisher("target_pose", PoseStamped,
queue_size=5)
16
17         rospy.Subscriber("ar_pose_marker", AlvarMarkers, self.get_tags)
18
19         rospy.loginfo("Publishing combined tag COG on topic /target_pose...")
20
21     def get_tags(self, msg):
22         # Initialize the COG as a PoseStamped message
23         tag_cog = PoseStamped()
24
25         # Get the number of markers
26         n = len(msg.markers)
27
28         # If no markers detected, just return
29         if n == 0:
30             return
31
```

```

32     # Iterate through the tags and sum the x, y and z coordinates
33     for tag in msg.markers:
34         tag_cog.pose.position.x += tag.pose.pose.position.x
35         tag_cog.pose.position.y += tag.pose.pose.position.y
36         tag_cog.pose.position.z += tag.pose.pose.position.z
37
38     # If we have tags, compute the COG
39     tag_cog.pose.position.x /= n
40     tag_cog.pose.position.y /= n
41     tag_cog.pose.position.z /= n
42
43     # Give the COG a unit orientation
44     tag_cog.pose.orientation.w = 1
45
46     # Add a time stamp and frame_id
47     tag_cog.header.stamp = rospy.Time.now()
48     tag_cog.header.frame_id = msg.markers[0].header.frame_id
49
50     # Publish the COG
51     self.tag_pub.publish(tag_cog)
52
53 if __name__ == '__main__':
54     try:
55         TagsCOG()
56         rospy.spin()
57     except rospy.ROSInterruptException:
58         rospy.loginfo("AR Tags COG node terminated.")
59

```

By this point in your ROS programming, this script will probably appear fairly basic so let's just touch on the key lines.

```
5 from ar_track_alvar.msg import AlvarMarkers
```

As we saw earlier, the `alvar_track` package defines the `AlvarMarkers` message type to describe an array of markers so we import the message type at the top of the script.

```
12     self.tag_ids = rospy.get_param('~tag_ids', None)
```

To improve reliability, we can read in an optional list of tag IDs. Any IDs not in the list will be ignored when computing the COG. The valid tag IDs can be set in the launch file as we saw earlier.

```
15     self.tag_pub = rospy.Publisher("target_pose", PoseStamped, queue_size=5)
```

Next we define a publisher to output the COG of the tracked tags as a `PoseStamped` message on the topic `/target_pose`.

```
17    rospy.Subscriber("ar_pose_marker", AlvarMarkers, self.get_tags)
```

Here we subscribe to the `/ar_pose_marker` topic and set the callback to the function `get_tags()` described next.

```
21  def get_tags(self, msg):
22      # Initialize the COG as a PoseStamped message
23      tag_cog = PoseStamped()
24
25      # Get the number of markers
26      n = len(msg.markers)
27
28      # If no markers detected, just return
29      if n == 0:
30          return
```

The `get_tags()` begins by checking the number of tags detected. If there are none, we simply return.

```
32      # Iterate through the tags and sum the x, y and z coordinates
33      for tag in msg.markers:
34          tag_cog.pose.position.x += tag.pose.pose.position.x
35          tag_cog.pose.position.y += tag.pose.pose.position.y
36          tag_cog.pose.position.z += tag.pose.pose.position.z
37
38      # If we have tags, compute the COG
39      tag_cog.pose.position.x /= n
40      tag_cog.pose.position.y /= n
41      tag_cog.pose.position.z /= n
42
43      # Give the COG a unit orientation
44      tag_cog.pose.orientation.w = 1
45
46      # Add a time stamp and frame_id
47      tag_cog.header.stamp = rospy.Time.now()
48      tag_cog.header.frame_id = msg.markers[0].header.frame_id
49
50      # Publish the COG
51      self.tag_pub.publish(tag_cog)
```

Here we iterate through the markers in the message array and sum up the `x`, `y` and `z` coordinates of each marker. If we have defined a list of valid tag IDs, then any IDs not in the list are skipped. If the tag list is empty (`None`), we include all detected tags. We then compute the mean values for each component which become the position values for the COG. We also assign the COG a unit orientation just to make it a proper pose message. After adding a time stamp and `frame_id`, we publish the COG on the `/target_pose` topic.

10.2.2 Tracking the tags with a pan-and-tilt head

If your robot has a pan-and-tilt head using Dynamixel servos, you can have the camera track the tag target published by the `ar_tags_cog` node.

Skip any of the following launch files if you already have them running from the previous section. Begin by launching the OpenNI node for your depth camera.

For the Microsoft Kinect:

```
$ rosrun freenect_launch freenect.launch
```

For the Asus Xtion, Xtion Pro, or Primesense 1.08/1.09 cameras:

```
$ rosrun openni2_launch openni2.launch
```

Next, bring up the `ar_track_alvar` node:

```
$ rosrun rbx2_ar_tags ar_indiv_kinect.launch
```

Now run the `ar_tags_cog` launch file so that the COG of any detected tags will be published on the `/target_pose` topic:

```
$ rosrun rbx2_ar_tags ar_tags_cog.launch
```

Next, launch the startup file for your robot and servos. We'll use the head-only version of Pi Robot as an example:

```
$ rosrun rbx2_bringup pi_robot_head_only.launch sim:=false
```

To keep the servos from overheating, bring up the `monitor_dynamixels.launch` file:

```
$ rosrun rbx2_diagnostics monitor_dynamixels.launch
```

Now bring up the 3D head tracker node we developed in the last chapter:

```
$ rosrun rbx2_dynamixels head_tracker.launch sim:=false
```

If you move the AR tag(s) in front of the camera, the head should track the COG of any detected markers.

10.3 Tracking Multiple Tags using Marker Bundles

The `ar_track_alvar` Wiki page refers to the use of [tag bundles](#) to improve the robustness of tracking by using more than one marker whose relative positions are known ahead of time. While we have already demonstrated an application that computes the COG of a group of markers, the markers in a tag bundle are assumed to fixed relative to a "master tag" and their relative positions are measured before hand and stored in a configuration file. This would enable the pose of a tracked object to be determined even if one of the markers is not visible.

At the time of this writing, support for tag bundles is not included in the Debian version of the `ar_track_alvar` package and the source version does not appear to run without an error. However, you may wish to try it yourself or at least keep an eye open for updates on the [project repository](#).

10.4 Following an AR Tag with a Mobile Robot

In *Volume 1*, we created a pair of follower applications to enable a mobile robot to follow either a specific type of object (face or color) or a moving person. Following an AR tag is even easier and more robust since a tag tends to be easier to detect under different lighting conditions and backgrounds.

For this application, we will use a fairly large marker such as the example found in the `rbx2_ar_tags/config` directory called `Marker_8_large.png`. This particular image is 17.5 cm on a side and fits nicely on an 8.5 x 11 sheet of paper. Print out this marker and stick it to a piece of cardboard or clipboard so that you can carry it around in front of the robot.

To bring up the `ar_track_alvar` node with parameters appropriate for the larger marker size, we will use the launch file `ar_large_markers_kinect.launch` found in the `rbx2_ar_tags/launch` directory. The file is shown below:

```
1 <launch>
2   <arg name="marker_size" default="17.5" />
3   <arg name="max_new_marker_error" default="0.08" />
4   <arg name="max_track_error" default="0.5" />
5
6   <arg name="cam_image_topic" default="/camera/depth_registered/points" />
7   <arg name="cam_info_topic" default="/camera/rgb/camera_info" />
8   <arg name="output_frame" default="/base_footprint" />
9
10  <arg name="debug" default="false" />
11  <arg if="$(arg debug)" name="launch_prefix" value="xterm -e gdb --args" />
12  <arg unless="$(arg debug)" name="launch_prefix" value="" />
```

```

13
14      <node name="ar_track_alvar" pkg="ar_track_alvar" type="individualMarkers"
15      respawn="false" output="screen" args="$(arg marker_size) $(arg
16      max_new_marker_error) $(arg max_track_error) $(arg cam_image_topic) $(arg
17      cam_info_topic) $(arg output_frame)" launch-prefix="$(arg launch_prefix)" />
18
19  </launch>
```

Notice how we set the `marker_size` parameter to 17.5 cm in line 2. We also set a fairly high `max_track_error` value of 0.5 in line 4. This value was found by trial and error to reduce the number of frames where the `ar_track_alvar` node loses the target. Of course, you can try different values with your setup and `rqt_reconfigure`.

Finally, notice how we set the `output_frame` to `/base_footprint` in line 8. (If your robot model does not use a `/base_footprint` frame, you could use `/base_link` instead.) This tells the `ar_track_alvar` node to publish marker poses relative to the base frame which means we don't have to do any frame transforms ourselves when it comes to adjusting the robot's motion to follow the marker.

The script that actually causes the robot to follow the marker is called `ar_follower.py` and is found in the `rbx2_ar_tags/nodes` subdirectory. And the corresponding launch file where we set various parameters is called `ar_follower.launch` in the `rbx2_ar_tags/launch` directory.

The `ar_follower.py` script is almost identical to the follower nodes studied in *Volume 1*. The main difference is that we now subscribe to the topic `/ar_pose_marker` instead of `/roi` as we did for face or color or a point cloud as was done for the person follower. Once we know the pose of the AR marker relative to the robot's base, we can adjust its rotation and linear speed accordingly.

The key part of the script is the callback for the subscriber to the `/ar_pose_marker` topic so let's look at that in detail.

```

76  def set_cmd_vel(self, msg):
77      # Pick off the first marker (in case there is more than one)
78      try:
79          marker = msg.markers[0]
80          if not self.target_visible:
81              rospy.loginfo("FOLLOWER is Tracking Target!")
82              self.target_visible = True
83      except:
84          # If target is lost, stop the robot by slowing it incrementally
85          self.move_cmd.linear.x /= 1.5
86          self.move_cmd.angular.z /= 1.5
87
88          if self.target_visible:
89              rospy.loginfo("FOLLOWER LOST Target!")
90              self.target_visible = False
```

```

91         return
92
93     # Get the displacement of the marker relative to the base
94     target_offset_y = marker.pose.pose.position.y
95
96     # Get the distance of the marker from the base
97     target_offset_x = marker.pose.pose.position.x
98
99
100    # Rotate the robot only if the displacement of the target exceeds the
threshold
101    if abs(target_offset_y) > self.y_threshold:
102        # Set the rotation speed proportional to the displacement of the
target
103        speed = target_offset_y * self.y_scale
104        self.move_cmd.angular.z = copysign(max(self.min_angular_speed,
105                                              min(self.max_angular_speed,
abs(speed))), speed)
106    else:
107        self.move_cmd.angular.z = 0.0
108
109    # Now get the linear speed
110    if abs(target_offset_x - self.goal_x) > self.x_threshold:
111        speed = (target_offset_x - self.goal_x) * self.x_scale
112        if speed < 0:
113            speed *= 1.5
114        self.move_cmd.linear.x = copysign(min(self.max_linear_speed,
max(self.min_linear_speed, abs(speed))), speed)
115    else:
116        self.move_cmd.linear.x = 0.0

```

Our callback function `set_cmd_vel()` computes the `Twist` command we want to send to the robot to keep it facing the marker and to stay within a given distance of it. Let's start with the first few lines:

```

78     try:
79         marker = msg.markers[0]
80         if not self.target_visible:
81             rospy.loginfo("FOLLOWER is Tracking Target!")
82             self.target_visible = True
83     except:
84         # If target is lost, stop the robot by slowing it incrementally
85         self.move_cmd.linear.x /= 1.5
86         self.move_cmd.angular.z /= 1.5
87
88         if self.target_visible:
89             rospy.loginfo("FOLLOWER LOST Target!")
90             self.target_visible = False
91
92     return

```

Recall that the messages published on the `/ar_pose_marker` topic consist of an array of markers of type `AlvarMarkers`. In Line 79 above we attempt to pull out the first such marker in the array. We use `try-except` around this block since if no marker is visible, attempting to access the first marker (`msg.markers[0]`) will throw an exception. This in turn enables to know that the target is lost so we can stop the robot and set the `self.target_visible` flag to `False`. It is a good idea to slow down the robot incrementally as we have done in Lines 85 and 86 since the `ar_track_alvar` package can sometimes fail to detect the marker for just a frame or two and we don't want the robot to stop and start with a jerky motion when this occurs.

```
94      # Get the displacement of the marker relative to the base
95      target_offset_y = marker.pose.pose.position.y
96
97      # Get the distance of the marker from the base
98      target_offset_x = marker.pose.pose.position.x
```

Assuming we have a marker, we get the lateral offset from the positional `y` component of the marker pose and the fore/aft offset from the `x` component. How do we know these are the right components? Recall that we set the `output_frame` for the `ar_track_alvar` node to be `/base_footprint`. This means that the marker poses as published in the `AlvarMarkers` messages and received by our callback function are already defined in our base frame. Furthermore, our base frame is oriented with the `x`-axis pointing forward and the `y`-axis pointing left.

With the offsets in hand, it is a simple matter to set the rotational and linear components of the `Twist` command appropriately as is done in Lines 101-116. As with our earlier follower programs, we first check that the offset exceeds a minimal threshold, then we set the robot speeds proportional to the offsets while respecting the maximum and minimum speeds set in the launch file.

10.4.1 Running the AR follower script on a TurtleBot

If you have a TurtleBot, you can use the following commands to test the `ar_follower.py` script. You can use similar commands with other robots by simply changing the launch file used to bring up your robot driver.

Start by launching the TurtleBot's startup files. Note that we are using the launch file from the *Volume 1* package and this command must be run on the TurtleBot's laptop:

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

Next, bring up the OpenNI node for the robot's depth camera. This command must also be run on the robot's computer.

For the Microsoft Kinect:

```
$ roslaunch freenect_launch freenect.launch publish_tf:=false
```

For the Asus Xtion, Xtion Pro, or Primesense 1.08/1.09 cameras:

```
$ roslaunch openni2_launch openni2.launch publish_tf:=false
```

NOTE: In the commands above, we set the `publish_tf` argument to `false` since the TurtleBot's launch files and URDF model already take care of publishing the camera frames.

Next, bring up the `ar_track_alvar` node for large markers. You can run this on either the robot's computer or your desktop assuming the two are networked appropriately:

```
$ roslaunch rbx2_ar_tags ar_large_markers_kinect.launch
```

Finally, run the `ar_follower.launch` file to bring up the `ar_follower.py` node and supporting parameters:

```
$ roslaunch rbx2_ar_tags ar_follower.launch
```

This command can also be run on either the robot or a networked desktop.

Now hold the large marker printout in front of the robot's camera a few feet away and start moving backward, forward, left and right. Once the `ar_track_alvar` node has locked on to the marker, the robot should start to follow your movements.

10.5 Exercise: Localization using AR Tags

As we saw in *Volume 1*, one can use the ROS `gmapping` and `amcl` packages to do SLAM using either a laser scanner or a depth camera. However, SLAM does not always work so well in long hallways or when somebody rearranges the furniture. Infrared lasers and depth cameras can also be relatively blind to darker colors, especially black, so if a wall happens to have a black baseboard right at the scan height of the laser or camera, it may not appear in the map at all.

An alternative to scan-based SLAM uses fiducials or AR tags strategically placed on walls or ceilings so that the robot can use AR detection and triangulation to figure out where it is. Although this method involves modifying the environment, fiducials can make for rather interesting wall art and will certainly get some attention at your next party. More importantly, markers can be placed above the furniture and even on the

ceiling so that changes to the layout of a room should have little effect on the robot. Of course, care must be taken that the robot can see at least one marker at all times, although a simple search routine can often get the robot out of a blind spot. Finally, markers can be assigned easy to remember semantic tags such as "middle of hallway" or "entrance to kitchen". This makes it relatively easy to create a mapping between tags and verbal instructions so that we could use a speech recognition node and ask the robot to go to a particular location.

The code you need to detect and localize markers relative to the robot was already covered in the previous section describing the `ar_follower.py` node. But it is left as an exercise for the reader to write a localization node using AR tags. A good place to start might be this student project on [Indoor localization using augmented reality markers](#). Another project written in C is being developed by Wayne Gramlich at <https://github.com/waynegrampich/fiducials> with a conversion to C++ and ROS integration by Austin Hendrix at https://github.com/trainman419/fiducials_ros.

11. ARM NAVIGATION USING MOVEIT!

It is relatively easy to add an arm to a robot, but it is much harder to make it do anything useful. When using our own arms and hands, it seems rather simple to reach for an object in space, even when constrained by nearby obstacles. However, it turns out that controlling such movements is a complex mathematical problem when we try to program a robot to perform similar actions.

Fortunately for us, ROS provides all the tools we need to program a multi-jointed arm to perform complex reaching and grasping tasks under real-world conditions. A gripper or hand attached to the end of an arm is referred to as an *end-effector*, and the general problem of moving the end-effector to a given position and orientation in space while avoiding obstacles is called *arm navigation*. It is also common to refer to a robot arm as a *manipulator*, and when the arm is attached to a mobile robot, we'll often use the term *mobile manipulation* instead.

The most common goal in arm navigation is to move the end-effector to a target position and orientation in space. If the arm includes a gripper or hand of some sort, we might also want to *grasp* an object at the target location and move it somewhere else in a task called *pick and place*.

A powerful ROS package called [MoveIt!](#) addresses almost all aspects of mobile manipulation including Motion Planning, Kinematics, Collision Checking, Grasping, Pick and Place, and Perception.

Originally developed at Willow Garage and now maintained by SRI International, MoveIt! is still actively being developed and not all of its features are complete or fully documented. The goal of this chapter is therefore to introduce a number of the fundamental operations using easy-to-understand code samples. However, to explore the full MoveIt API in all its depth would take an entire volume of its own so we will really only be scratching the surface.

Before we dive into the examples, we need to cover a few background topics. If you are already familiar with the basic terminology used in arm navigation including degrees of freedom, ROS joint types, and forward and inverse kinematics, feel free to skip ahead.

In this chapter we will use MoveIt! to carry out the four most common arm navigation tasks:

- moving the arm to a specified joint configuration using forward kinematics

- moving the end-effector to a specified Cartesian pose (position and orientation) using inverse kinematics
- moving the arm in the presence of obstacles and path constraints
- performing a pick-and-place operation where the arm uses its gripper to grasp an object and move it to another location

But before we get to these tasks, we have to lay some ground work.

11.1 Do I Need a Real Robot with a Real Arm?

Not everyone has a robot with a multi-jointed arm. So MoveIt! provides a demo mode for running various arm navigation scenarios using nothing more than a URDF model of your robot and a set of fake joint controllers that are provided by MoveIt! itself. Once we have MoveIt! working in demo mode, we will switch to the ArbotiX simulator that we used in *Volume 1* and explored more fully in Chapter 5. This will allow us to test how the arm would move if using real servos but without any physics involved. If your robot does have an arm, we provide instructions on how to use MoveIt! to control real Dynamixel servos. Finally, in the next chapter on using the Gazebo simulator, we will run our MoveIt! code using a model of the sophisticated [UBR-1](#) robot.

11.2 Degrees of Freedom

A 3-dimensional object requires six numbers to specify its pose in everyday Cartesian space: three for its position (x, y, z) and three for its orientation (*roll, pitch, yaw*). A general purpose arm therefore requires at least six appropriately oriented joints to have a fighting chance of reaching and grasping nearby objects. It is certainly possible to use arms with fewer joints, especially when operating under more restricted conditions. For example, a fixed position assembly line robot that only needs to pick up objects from directly overhead can get away with just three or four degrees of freedom.

The second complication is the presence of *constraints*. Constraints on the movement of the arm are typically caused by the presence of obstacles, including other parts of the robot itself. For example, we don't want the arm to collide with the robot's head or torso. Similarly, if the robot needs to reach an object that is next to an obstacle, not only must the gripper make it cleanly past the obstacle, but the rest of the arm must not strike it either. Imagine reaching for the butter at the dinner table without spilling the wine glass near your elbow.

We can also envision *path constraints* on the arm. For example, if the task is to move an open container of liquid from one location to another, even in an obstacle-free environment, the hand or gripper must keep the container in a relatively upright orientation while the rest of the arm does the moving. Planning an arm trajectory in the

presence of constraints is a very complex problem and rarely has an analytical solution. As a result, most arm controllers must resort to random sampling of the problem space.

In addition to degrees of freedom and constraints, a number of other factors determine the overall usability of an arm. We must also consider the range of motion of each joint, the orientation of the joint rotation axes, and of course, the lengths of the arm segments between joints. Together, these properties determine the *reachability space* of the arm. A well designed arm will be able to position and orient its gripper in a relatively large volume encompassing the desired workspace of the robot—the region where the robot will do most of its grasping. Robots like the PR2 and UBR-1 have a telescoping torso joint that greatly increases the reachability space of these robots.

The movement and control of robotic arms is called *arm navigation*. Fortunately for us, ROS provides all the tools we need to create a model of our robot's arm and make it perform a number of real-world tasks. But before diving into the code, we need to add a few more concepts to our vocabulary.

11.3 Joint Types

Robot joints come in a number of flavors which are defined on the [ROS wiki](#):

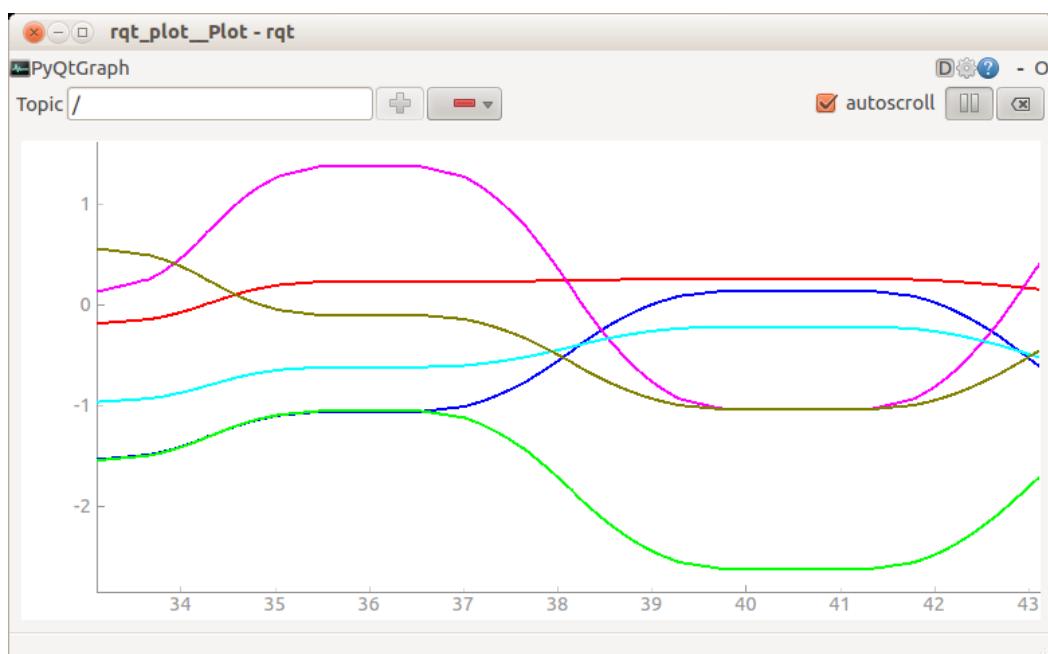
- **revolute** - a joint like a servo that rotates along an axis and has a limited range specified by the upper and lower limits.
- **continuous** – a joint like a drive wheel that rotates around an axis and has no upper or lower limits
- **prismatic** - a sliding joint like a linear actuator that has a limited range specified by the upper and lower limits
- **fixed** - not really a joint because it cannot move. All degrees of freedom are locked.
- **floating** - this joint type allows motion along all 6 degrees of freedom.
- **planar** - this joint type allows motion in a plane perpendicular to the axis

Most of the joints we will deal with are revolute like a regular servo. We will therefore typically use the terms "joint angle" and "joint position" interchangeably. If we are dealing specifically with a prismatic joint like a telescoping torso, we will use the term "position" instead of "angle".

11.4 Joint Trajectories and the Joint Trajectory Action Controller

So far we have been controlling servo motions by sending individual position and speed commands to each joint on every update cycle of our control script. ROS defines a more powerful method for controlling the movements of multiple joints simultaneously called the [Joint Trajectory Action Controller](#) (JTAC). The JTAC is designed to accept an entire *joint trajectory* as input. A joint trajectory is a list or sequence of joint positions, velocities, accelerations and efforts for each joint over a period of time.

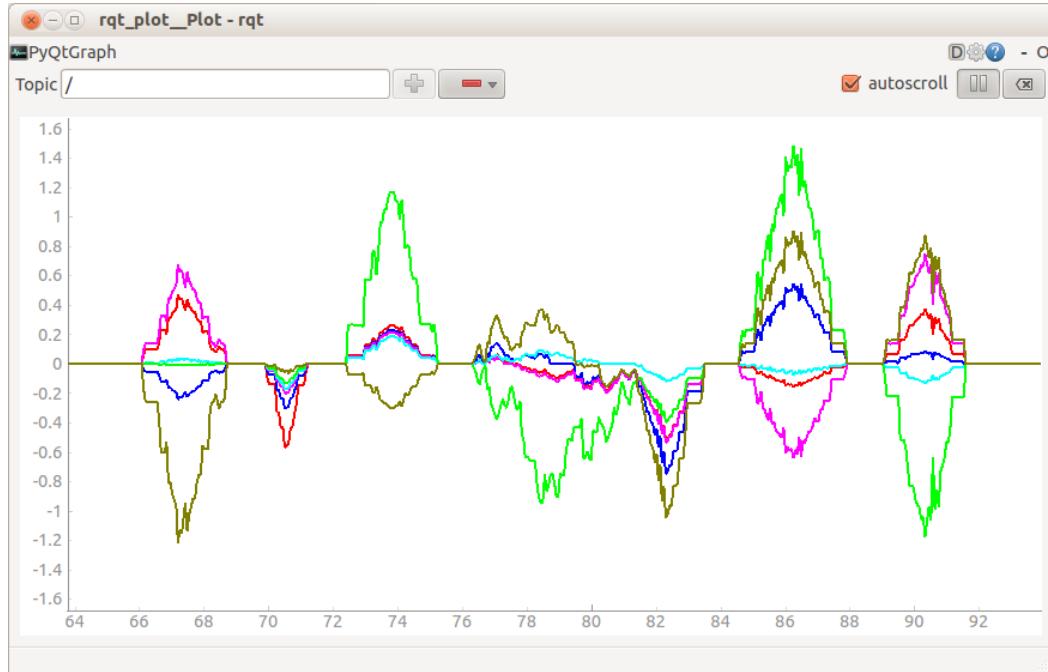
You can view the position and velocity components of a joint trajectory by using `rqt_plot` to graph the relevant components of the `/joint_states` message while the arm is moving. The image below shows the joint positions of Pi Robot's six arm joints as he reaches for an object. The x-axis is time in seconds and the y-axis reflects joint position in radians. Each color represents a different joint.



This plot was generated by running `rqt_plot` on the relevant joint positions while the arm was moving. The actual command used was:

```
$ rqt_plot /joint_states/position[2], /joint_states/position[3],  
/joint_states/position[4], /joint_states/position[5],  
/joint_states/position[6], /joint_states/position[8]
```

The next plot shows the joint *velocities* as the arm moves between seven different poses with a pause in between each motion except the fourth and fifth. Notice how the joint velocities begin and end with 0 for each trajectory except at the boundary between the fourth and fifth trajectories where there is blending from the end of one trajectory into the start of the next.



This plot was generated using the command:

```
$ rqt_plot /joint_states/velocity[2], /joint_states/velocity[3],
/joint_states/velocity[4], /joint_states/velocity[5],
/joint_states/velocity[6], /joint_states/velocity[8]
```

We can see the various fields in a ROS `JointTrajectory` message using the command:

```
$ rosmsg show JointTrajectory
```

which should return the following output:

```
[trajectory_msgs/JointTrajectory] :
std_msgs/Header header
  uint32 seq
```

```

time stamp
string frame_id
string[] joint_names
trajectory_msgs/JointTrajectoryPoint[] points
float64[] positions
float64[] velocities
float64[] accelerations
float64[] effort
duration time_from_start

```

Here we see that a joint trajectory consists of a standard ROS header together with an array of joint names and an array of trajectory points. A trajectory point consists of an array of positions, velocities, accelerations and efforts that describe the motion of the joints at that point. Note that typical trajectories will have all zeros for the joint velocities for the first and last point meaning that the arm starts and finishes at rest. Each trajectory point also includes a `time_from_start` that specifies the number of seconds after the time stamp in the header when we expect the trajectory to pass through this point.

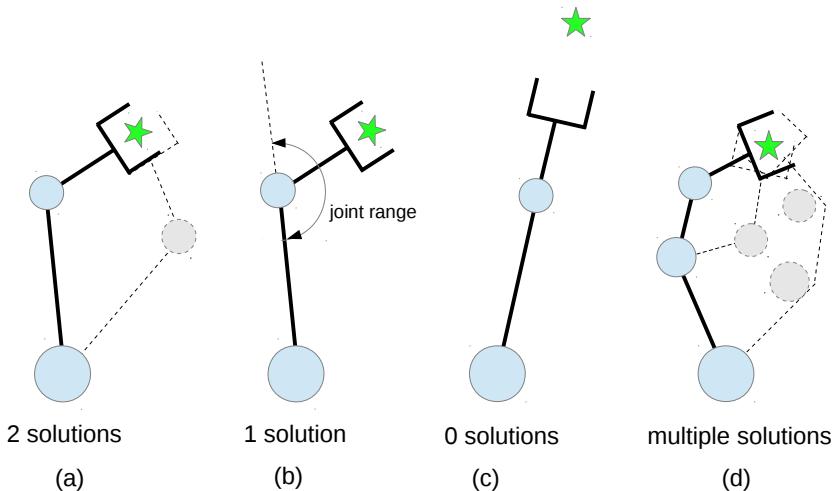
The simplest joint trajectory would contain a single point specifying how we want the joint configuration to look at a given time from start. But most trajectories will include many points that describe the arm configuration at various points over time. It is the job of the joint action trajectory controller to interpolate these points (using [splines](#) for example) to create a smooth trajectory that can be implemented on the underlying hardware. Both the `arbotix` and `dynamixel_motor` packages provide a joint trajectory action controller that works with Dynamixel servos. So to move a multi-jointed Dynamixel arm a certain way over time, we can send the desired joint trajectory to one of these controllers which will then compute the required interpolation between points and control the servos to move along the requested trajectory. We will see how this works later in the chapter.

You might be wondering how on earth can we figure out the positions, velocities, accelerations and efforts of every joint along an entire trajectory? And the answer is that we almost never have to do this ourselves. Instead, we will normally specify the much simpler goal of moving the end-effector to a specific position and orientation in space. We will then rely on MoveIt! to compute a joint trajectory that will move the end-effector into the desired pose, all while avoiding obstacles and satisfying any other constraints specific to the task. This trajectory will then be sent to the joint trajectory action controller that will command the servos to execute the planned trajectory on the actual arm.

11.5 Forward and Inverse Arm Kinematics

Using basic trigonometry, it is relatively straightforward to compute the pose of the hand or gripper in space when we know the lengths of the links and the joint angles between them. This is called the *forward kinematics* of the arm and is often abbreviated *FK*. However, if we are given the desired pose of the end-effector in space and asked to compute a set of joint values that will place it there, things start to get messy: in this case we have to compute the *inverse kinematics (IK)* of the arm. An algorithm that computes the inverse kinematics is called an *inverse kinematics solver* or simply an *IK solver*.

Depending on the number of joints in the arm (its degrees of freedom) a given inverse kinematics calculation may have one solution, no solution, more than one solution, or possibly even an infinite number of solutions. Consider for example the four situations illustrated below where we represent a planar arm attempting to reach a goal location indicated by the star.



In the first three images, the arm has two revolute joints and therefore two degrees of freedom. The location of the goal also requires two values, namely its x and y coordinates in the plane. In situations like (a), the arm generally has two inverse kinematic solutions (sometimes called "elbow up" and "elbow down"). However, if a joint has a limited range of motion as shown in (b), then only one of these solutions may actually be realizable by the arm. If the goal is out of reach as shown in (c), then there are no solutions. In (d) we add a third joint so now the arm has an extra degree of freedom. As you might expect, this opens up additional solutions.

Things get even trickier when we add constraints such as the presence of obstacles. If a potential solution would cause any part of the arm to collide with the obstacle, then that

solution has to be eliminated. A kinematic solver that can incorporate constraints is said to be *constraint aware*. Constraints can also be specified in terms of position and orientation of the end-effector. For example, if the robot has to move an open container of liquid from one location on a table to another, then the arm must move so that the orientation of the gripper keeps the container from tipping. MoveIt! makes this type of problem relatively easy to solve.

11.6 Numerical versus Analytic Inverse Kinematics

By default, MoveIt! solves the inverse kinematics problem using **KDL**, the [Kinematics and Dynamics Library](#) that is part of the [Orcos Project](#) (Open Robot Control Software). KDL uses numerical methods to do the computation since the inverse kinematics of a general kinematic chain is not solvable in closed form. While numerical methods are a good place to start, they suffer from a couple of drawbacks.

First, numerical methods tend to be relatively slow due to the large number of iterations typically required to find a solution. Second, these methods may fail to find a solution altogether if the initial configuration ("seed value") used for a given problem leads the process into the wrong part of the configuration space. If you use MoveIt's default KDL plugin for awhile, you will run into occasions where the solver gives up after a number of attempts even though you know that the goal pose for the end effector should be achievable.

Fortunately, a faster and more reliable alternative is available. The [OpenRAVE](#) project enables the creation of custom analytic IK solvers for many different types of robot arm or other kinematic chains. Later in this chapter we will provide detailed instructions on how to generate a custom solver for your robot arm that can then be used with MoveIt! in place of the KDL plugin. The custom solution will typically run 10-30 times faster than KDL and will be less likely to miss a valid solution.

11.7 The MoveIt! Architecture

For an excellent description of the overall MoveIt! framework, see the [Conceptual Overview](#) on the MoveIt! wiki. MoveIt! provides both a Python and C++ API for nearly all aspects of motion control including [kinematics](#), [joint control](#), the [planning scene and collision checking](#), [motion planning](#), and [3D perception](#).

The API we will use in this book is the Python [move group interface](#). A *move group* describes a part of the robot that represents a [kinematic chain](#) which is a series of links connected by joints such as a pan-and-tilt head or an arm. The [planning scene](#) represents an abstraction of the state of the world that can monitor the placement of obstacles and the state of the robot. It can therefore be used for collision checking and constraint evaluation. All planning actions must reference a given move group and take place within a given planning scene.

Perception involves the use of sensor data such as laser scans and point clouds to insert detected objects into the planning scene, but we can just as easily add virtual objects or choose to ignore certain kinds of sensor data. This allows the robot to plan its actions in a virtual environment that may or may not correspond to the immediate state of the world around it. For one thing, this means that we can test all the planning tools using a purely simulated robot and a collection of virtual objects. It also means that we can program the robot to "imagine" planning scenarios or to compare a stored planning scene with a scene that accurately reflects the current state of the world in front of it.

Fortunately for us, MoveIt! includes the [Setup Assistant](#) that makes it easy to configure our robot for motion planning. In particular, the Setup Assistant will take care of the following tasks:

- URDF → SRDF: The URDF model of the robot is used to create an enhanced *Semantic Robot Description File* that includes tags defining move groups, end effectors, and self-collision information. The SRDF file is created automatically from the URDF file by running the MoveIt! Setup Assistant.
- Additional configuration files are also automatically created by the Setup Assistant and these files specify the robot's kinematic solvers, joint controllers and joint limits, motion planners and sensors. The configuration files are created in the `config` subdirectory of the MoveIt! package directory for a given robot and the files we will be most concerned are `kinematics.yaml`, `controllers.yaml`, `joint_limits.yaml` and `sensors_rgbd.yaml`, all of which will be described in detail later in the chapter.
- The Setup Assistant also creates a number of launch files for running key planning and motion control modules including:
 - the [planning scene pipeline](#) including collision checking and an occupancy grid using an [Octomap](#)
 - [motion planning](#) using [OMPL](#) (Open Motion Planning Library)
 - the `move_group` node that provides ROS topics, services and actions used for kinematics, planning and execution, pick and place, state validation and planning scene queries and updates
 - a joint trajectory controller manager for launching the appropriate trajectory controllers for the given robot

Two key items not created by the MoveIt! Setup Assistant are the configuration files and launch files for your robot's particular joint controller. We will therefore cover this task in detail later in the chapter.

11.8 Installing MoveIt!

To install MoveIt!, first make sure you have the latest updates:

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

Then install the packages:

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

We will also need the IK Fast module:

```
$ sudo apt-get install ros-indigo-moveit-ikfast
```

If you also want to experiment with the MoveIt! configuration for the Willow Garage PR2, run the command:

```
$ sudo apt-get install ros-indigo-moveit-full-pr2
```

That's all there is to it!

11.9 Creating a Static URDF Model for your Robot

MoveIt! requires a pure URDF model for the robot—i.e. there can be no `<xacro:include>` tags referring to other files. We can use the ROS `xacro` utility to render our mixed URDF/Xacro model as a static URDF file. For the one-arm model of Pi Robot, the relevant commands would be:

```
$ roscd rbx2_description/urdf/pi_robot
$ rosrun xacro xacro.py pi_robot_with_arm.xacro > pi_robot.urdf
```

If you are using your own robot model located in a different package and/or directory, alter the commands accordingly. Also, the name of the resulting URDF file can be anything you like but you'll need to remember it when running the MoveIt! Setup Assistant.

To double-check the validity of your URDF model, run the ROS `check_urdf` utility:

```
$ check_urdf pi_robot.urdf | less
```

The beginning of the output should look something like this:

```
robot name is: pi_robot
----- Successfully Parsed XML -----
root Link: base_footprint has 1 child(ren)
    child(1): base_link
        child(1): base_l_wheel_link
        child(2): base_r_wheel_link
        child(3): torso_link
            child(1): head_base_link
                child(1): head_pan_servo_link
                    child(1): head_pan_bracket_link
                    child(1): head_tilt_servo_link
                    child(1): head_tilt_bracket_link
```

Note how robot name is output correctly as well as the root link which is `base_footprint` in our case. The rest of the output shows how all the robot's links are connected as a tree off the root link.

NOTE: If you need to change your robot model at a later time, do not edit the static URDF file. Instead, edit the original URDF/Xacro file(s) and then run the `xacro` utility again to regenerate the static file.

11.10 Running the MoveIt! Setup Assistant

If you have your own robot arm, or even just a URDF model of a robot you would like to try in simulation, you will need to first run the MoveIt! Setup Assistant to generate the configuration files specific to your setup. If you just want to use the Pi Robot model in simulation, you can skip this section for now since Pi's MoveIt! configuration files are already included in the `rbx2/pi_robot_moveit_config` directory. Later on in the book, we will also test the MoveIt! configuration for the [UBR-1](#) robot.

The steps below were used to generate the MoveIt! configuration files for Pi Robot. You can use these steps as a guide to create your own configuration files if you have a URDF model of another robot you'd like to work with. You can find similar instructions for the Willow Garage PR2 at:

http://docs.ros.org/indigo/api/moveit_setup_assistant/html/doc/tutorial.html

We are now ready to run the MoveIt! Setup Assistant:

```
$ rosrun moveit_setup_assistant setup_assistant.launch
```

Eventually the following setup screen should appear:

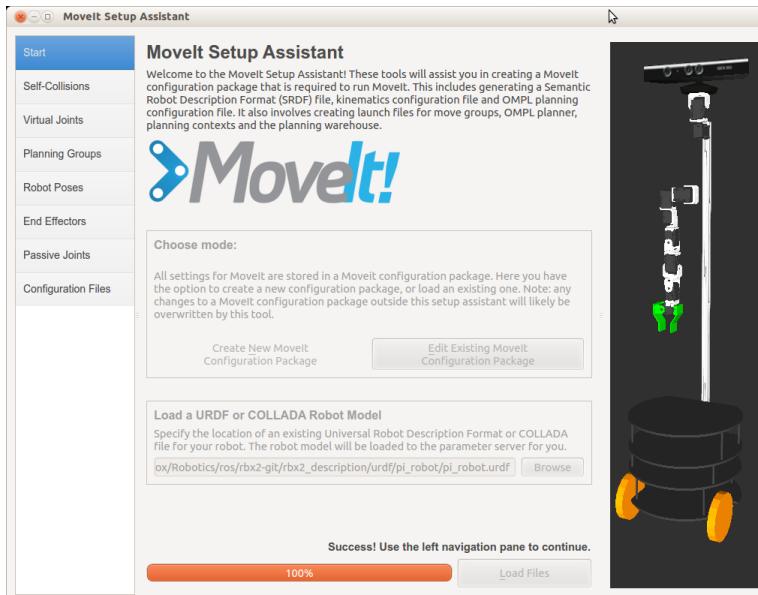


Now proceed with the following steps.

11.10.1 Load the robot's URDF model

The setup process begins by loading the static URDF model we created earlier.

- To create a new MoveIt configuration package, click the button labeled **Create New MoveIt Configuration Package**.
- Under the section labeled "Load a URDF or Collada Robot Model", click the **Browse** button and navigate to the URDF file we created above:
`rbx2_description/urdf/pi_robot/pi_robot.urdf`
- With your model chosen, click the **Load Files** button.
- If all goes well, you should see an image of Pi Robot in the right hand panel. (The black background makes certain parts of the model a little hard to see.) You can rotate, move and zoom the model using your mouse just like in **RViz**.



NOTE: You might find that the setup assistant crashes at this point and aborts back to the command line. This is usually due to a crash in the assistant's `RViz` plugin as it tries to come up. If this happens to you, first simply try again. If you keep getting crashes, try running `RViz`'s GL engine in software-only mode by setting the following environment variable:

```
$ export LIBGL_ALWAYS_SOFTWARE=1
```

Then run the setup assistant again.

11.10.2 Generate the collision matrix

The collision matrix lists all the links that can never be in collision with each other. For example, the gripper can never collide with one of the wheels because the arm is not long enough. Having this table pre-computed can save valuable CPU time when it comes to planning movements of the arm. Follow these steps to generate the collision matrix for the loaded robot model.

- click on the **Self-Collisions** tab on the left panel
- click on the large button labeled **Regenerate Default Collision Matrix**. You can generally leave the sampling density set to the default value of 10000.)

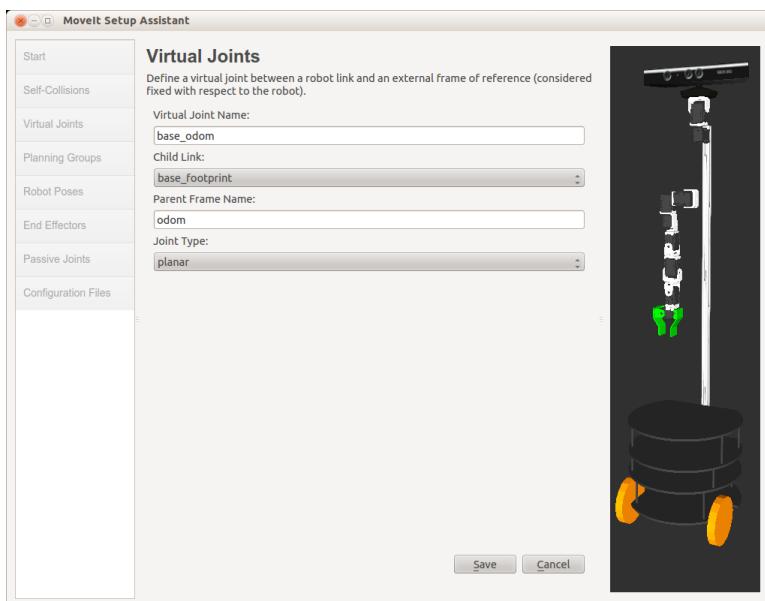
The result should be a list of link pairs that cannot collide with one another because of the geometric constraints of the robot.

11.10.3 Add the `base_odom` virtual joint

Virtual joints are used primarily to attach the robot to the world. In our case, we want to connect the robot's `base_footprint` frame to the `odom` frame using a `planar` joint which allows the two links to slide over each other in a 2D plane. (If your robot uses the [`robot_pose_ekf`](#) package to combine wheel odometry information with data from an IMU or gyro, then you will connect the robot to the `odom_combined` frame.)

Create a virtual link called `base_odom` as follows:

- click on the **Virtual Joints** tab on the left panel
- click on the **Add Virtual Joint** button
- in the text box labeled **Virtual Joint Name**, enter `base_odom`
- set the **Child Link** to `base_footprint`
- in the text box labeled **Parent Frame Name** enter `odom` or `odom_combined` if using the [`robot_pose_ekf`](#) package.
- set the **Joint Type** to `planar`
- finally, click the **Save** button

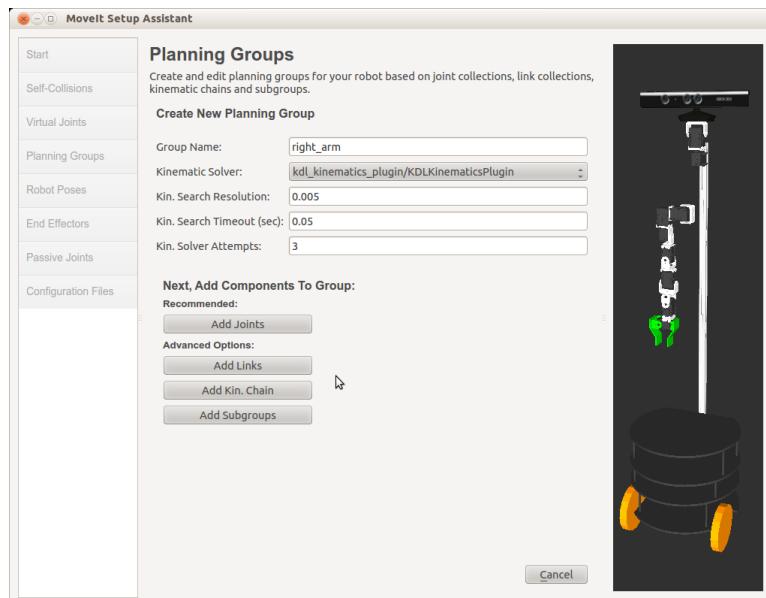


11.10.4 Adding the right arm planning group

Planning groups are used to describe different semantic parts of your robot, such as the right arm, the right gripper, or the head. Begin by adding a group for the right arm as follows.

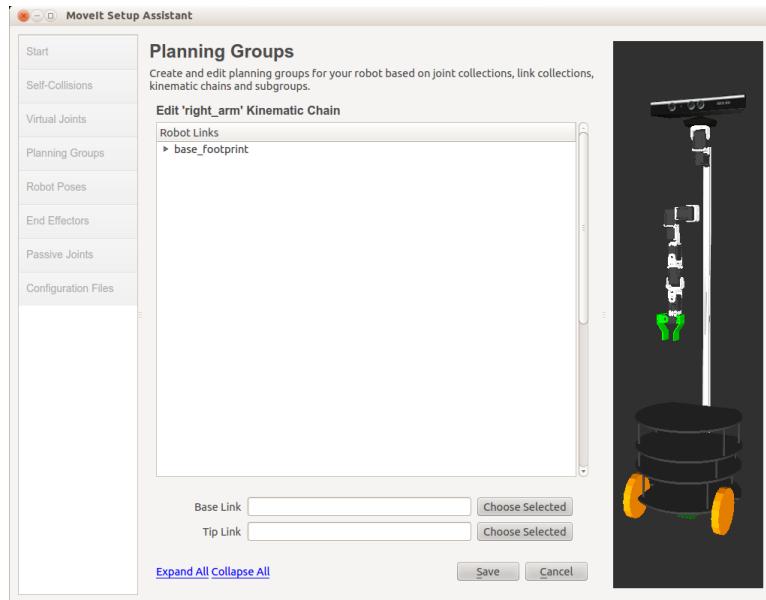
- click on the **Planning Groups** tab on the left panel
- click the **Add Group** button
- next, fill in the other boxes as follows:
 - **Group Name:** right_arm
 - **Kinematic Solver:** kdl_kinematics_plugin/KDLKinematicsPlugin
 - **Kin. Search Resolution:** 0.005
 - **Kin. Search Timeout (sec):** 0.05
 - **Kin. Solver Attempts:** 3

So far the screen should look like this:

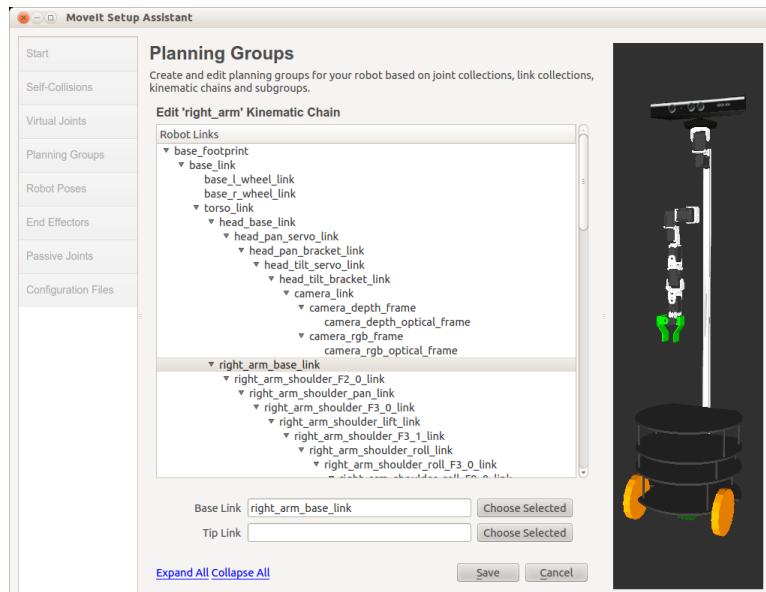


We will add the arm's joints to the planning group as a serial kinematic chain as this is required when using the KDL plugin. Therefore, click the **Add Kin. Chain** button.

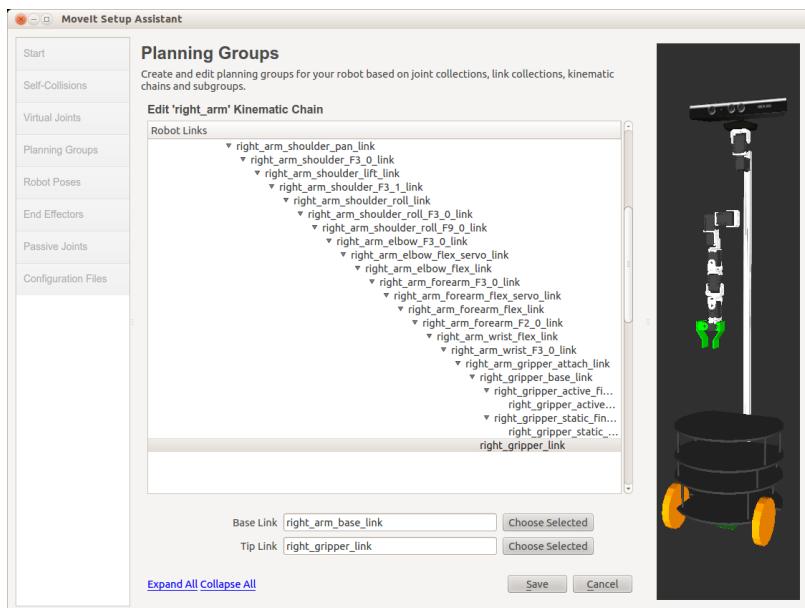
The next screen should look like the following:



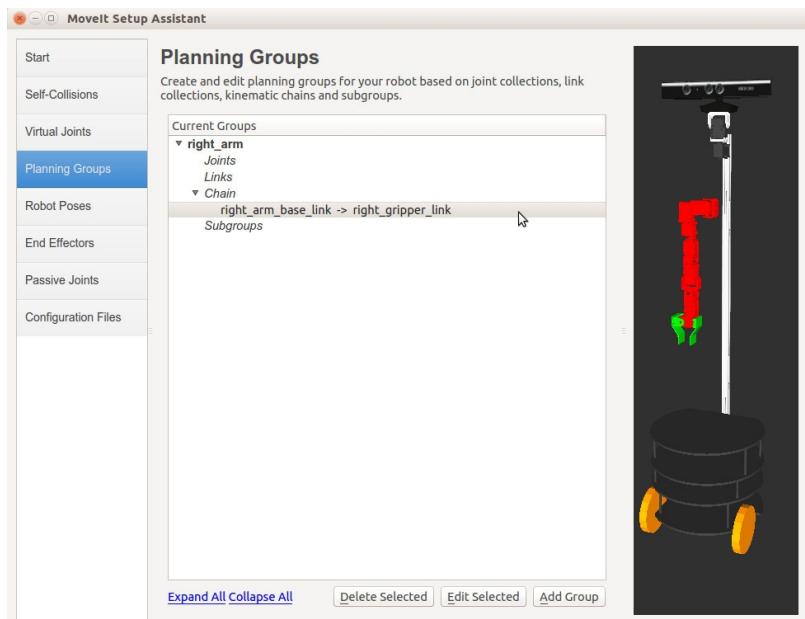
Click the link labeled **Expand All** near the bottom of the window to see the entire link tree of the robot as shown below. Then select the `right_arm_base_link` in the tree and click the **Choose Selected** button next to the **Base Link** text box. The result should look like this:



Now select the `right_gripper_link` for the **Tip Link** and the result should look like this:



Finally, click the **Save** button. The final screen should look like this:



Note how we have selected the right arm chain which is then highlighted in red in the image.

The last link in the chain is often referred to as the "tip" and typically corresponds to the end-effector. Note how in this case, we are using the virtual `right_gripper_link` that was created in the URDF in between the two gripper fingers to make it easier to specify a desired pose for the end-effector.

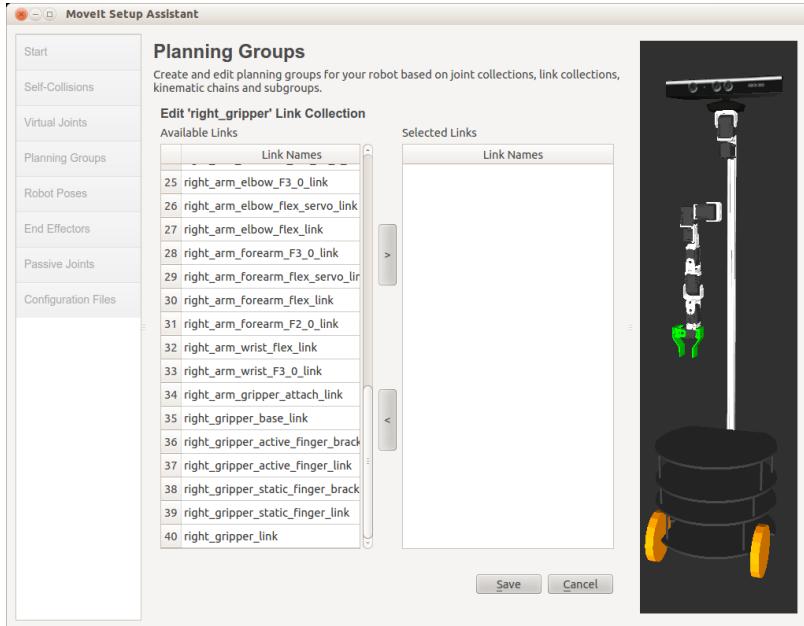
You can continue to add planning groups if desired. For example, if your robot has a telescoping torso, you could define a group that includes the torso and the arm. That way the prismatic torso joint could be used to move the whole arm up and down when solving a given inverse kinematics problem.

11.10.5 Adding the right gripper planning group

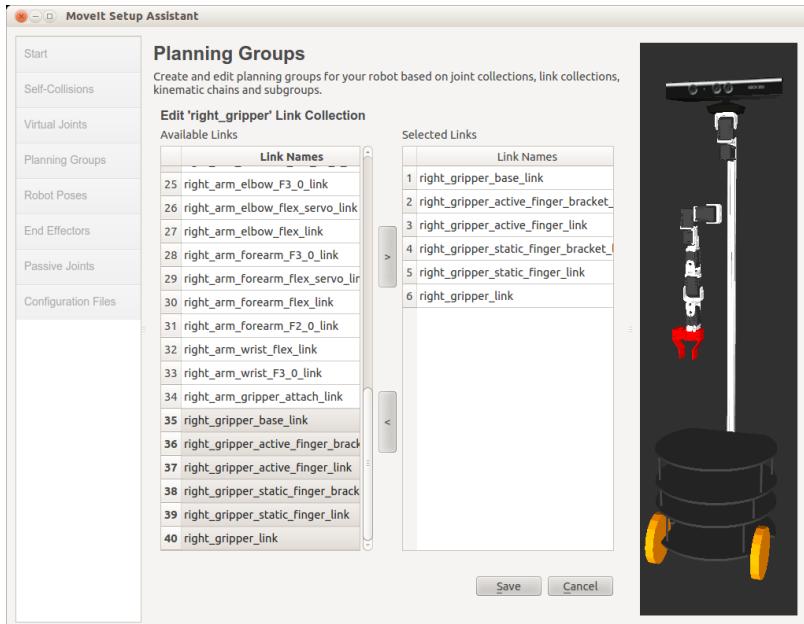
Next we'll add a planning group for the right gripper. Note that the procedure is slightly different than what we just did for the arm. Start by clicking the **Add Group** button in the previous screen. Then fill in the top section as follows:

- **Group Name:** right_gripper
- **Kinematic Solver:** None
- **Kin. Search Resolution:** 0.005
- **Kin. Search Timeout (sec):** 0.05
- **Kin. Solver Attempts:** 3

This time, click the **Add Links** button. The screen should look like the following:



Check that the left and right panels are labeled "Link Names". In the left panel, select all links whose name begins with `right_gripper`, then click the right arrow button to move the links into the right panel. The result should look like this:



Note how the gripper is now highlighted in red in the image on the right. Finally, click the **Save** button.

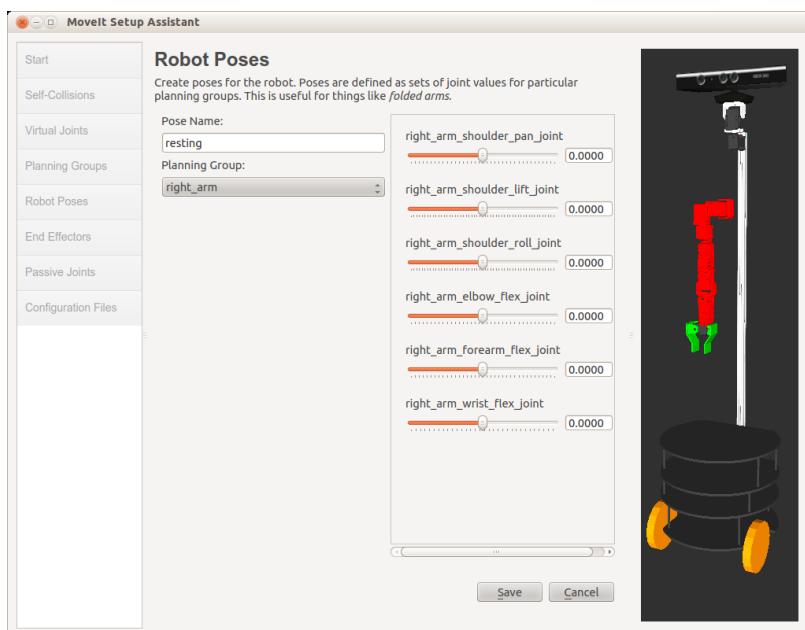
11.10.6 Defining robot poses

It will become convenient later on to define a few named poses for the robot arm. For example, we will use the name "resting" for the default pose where the arm hangs straight down. We will also create another pose called "straight_forward" where the arm is pointing straight forward and yet another called "wave" where the arm is held upward as if waving to someone.

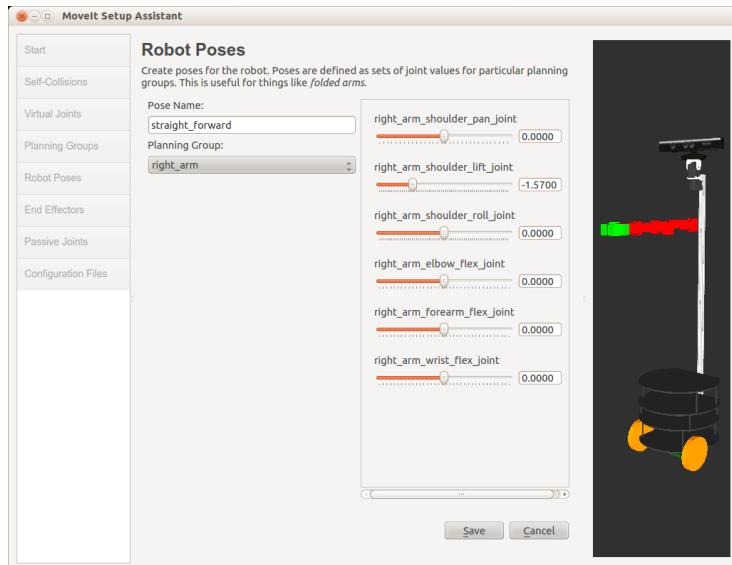
To define a named pose, proceed as follows:

- click on the **Robot Poses** tab on the left
- click on the **Add Pose** button
- enter a name for the pose in the **Pose Name** text box.
- Choose the planning group for this pose
- use the slider controls to position the arm the way you want for the given pose

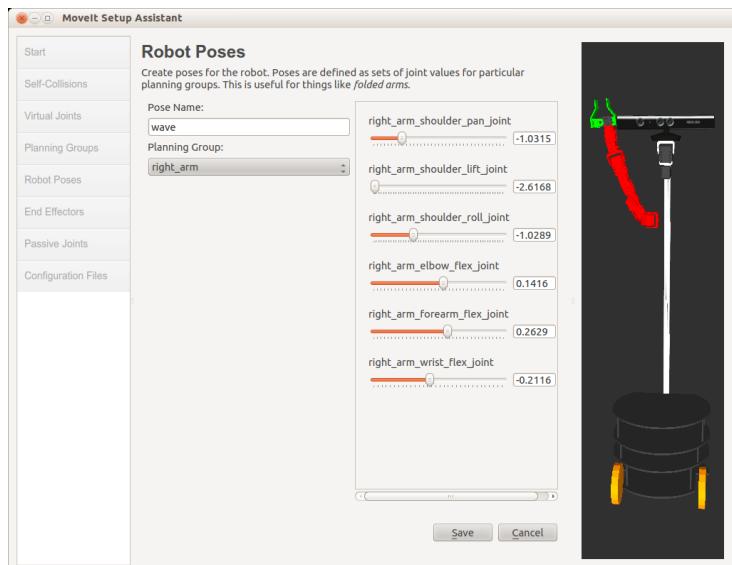
The image below shows the screen for the "resting" pose of the right arm:



When the arm is positioned the way you want, click the **Save** button. To add another pose, click the **Add Pose** button again. In the following screen shot, we have defined a pose called `straight_forward` with the arm pointing forward and parallel to the floor.



To add the "wave" pose, position the arm as follows:



When finished, click the **Save** button.

11.10.7 Defining end effectors

While we have already created the right gripper planning group, we now have to identify it as an end-effector which gives it special properties in MoveIt! for actions like pick-and-place. Create the `right_end_effector` object as follows:

- click the **End Effectors** tab on the left
- click the **Add End Effector** button
- enter `right_end_effector` for the **End Effector Name**
- select `right_gripper` in the **End Effector Group** pull down menu
- in the **Parent Link** pull down menu, select `right_gripper_link`
- leave the **Parent Group** blank
- click the **Save** button

Note that we are again using the virtual `right_gripper_link` instead of a physical link. For grasping operations in particular, it is very helpful to use a reference frame located in between the gripper's finger links when planning the approach of the gripper toward the object to be grasped.

11.10.8 Defining passive joints

Passive joints like caster wheels are not controlled by the robot and are entered here to let MoveIt! know they can't be planned for. Although Pi Robot does have a caster wheel in real life, it was left off the URDF model for simplicity so we have nothing to add here.

11.10.9 Generating the configuration files

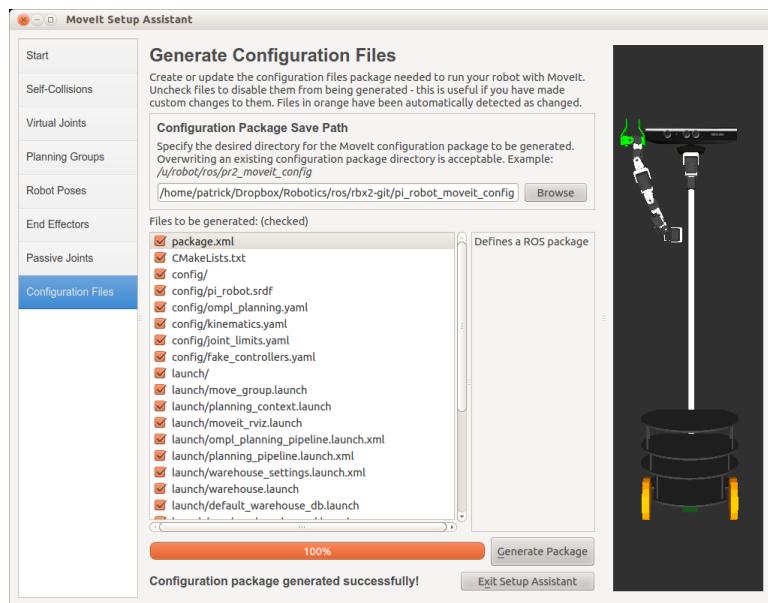
The final step will automatically generate a ROS package containing all the configuration files for your robot. Proceed with the following steps:

- click the **Configuration Files** tab on the left
- click the **Browse** button
- navigate to your `catkin/src` directory and create a new folder to hold your robot's MoveIt configuration files. Most people choose a folder name of the form `robot_name_moveit_config`. So for Pi Robot, we use `pi_robot_moveit_config`. This will also become the package name for

your configuration. Click **Open** in the file browser to move into the newly created folder.

- click the **Generate Package** button

If all files are successfully created, you should see the status message **Configuration package generated successfully!** at the bottom of the window and the screen should look something like this:



To exit the assistant, click the button labeled **Exit Setup Assistant**.

Finally, to ensure that ROS can find your new package, run the following command:

```
$ rospack profile
```

We only need to run this command this one time. New terminals will automatically pick up your new package.

11.11 Configuration Files Created by the MoveIt! Setup Assistant

The MoveIt! Setup Assistant creates all the configuration files needed to test arm navigation in demo mode. These files are stored in the `config` subdirectory of the robot's MoveIt! package. For instance, the config files for Pi Robot can be found in the `rbx2` repository under the directory `pi_robot_moveit_config/config`. Let's take a look at the four files that will concern us the most in this volume.

11.11.1 The SRDF file (`robot_name.srdf`)

The MoveIt! Setup Assistant needs a place to store the configurations you have chosen for your robot's arm(s), end-effectors, named poses and so on. A standard URDF file does not have the flexibility required to store this information so a new file using the [Semantic Robot Description Format](#) is used for this purpose. The name of the file is `robot_name.srdf`, so for Pi Robot, the file is called `pi_robot.srdf`.

The SRDF file is plain text and easy to read so you can bring it up in a text editor to view its contents. The key tags and definitions are as follows:

- A `<group>` element for each planning group listing the chains, joints or links making up that group. For example, the `<group>` element for Pi Robot's right arm looks like this:

```
<group name="right_arm">
  <chain base_link="right_arm_base_link" tip_link="right_gripper_link">
</group>
```

- A `<group_state>` element that stores the joint positions for any named poses you created. The element that stores the "wave" pose for Pi Robot's arm looks like this:

```
<group_state name="wave" group="right_arm">
  <joint name="right_arm_elbow_flex_joint" value="0.1416" />
  <joint name="right_arm_forearm_flex_joint" value="0.2629" />
  <joint name="right_arm_shoulder_lift_joint" value="-2.6168" />
  <joint name="right_arm_shoulder_pan_joint" value="-1.0315" />
  <joint name="right_arm_shoulder_roll_joint" value="-1.0288" />
  <joint name="right_arm_wrist_flex_joint" value="-0.2116" />
</group_state>
```

- An `<end_effector>` element indicating the group and parent link for the end-effector. Here is the element for Pi Robot's right gripper:

```
<end_effector name="right_end_effector" parent_link="right_gripper_link"
group="right_gripper" />
```

- A `<virtual_joint>` element for each virtual joint created by the Setup Assistant. Here is the definition of the planar virtual joint we created between the `odom` and `base_footprint` frames:

```
<virtual_joint name="base_odom" type="planar" parent_frame="odom"
child_link="base_footprint" />
```

- Finally, at the end of the SRDF file is a long list of `<disable_collisions>` elements. These lines were created when we generated the collision matrix in the Setup Assistant. The Setup Assistant sorts through the robot's URDF model and figures out which pairs of links can never be in collision such as the `base_footprint` and the `head_pan_bracket_link` which would be represented like this:

```
<disable_collisions link1="base_footprint"
link2="head_pan_bracket_link" reason="Never" />
```

By storing this information in a static file, the MoveIt! motion planners can save a significant amount of time by not checking these link pairs for collisions whenever a new arm motion is planned.

11.11.2 The `fake_controllers.yaml` file

To run MoveIt! in demo model, a set of fake joint trajectory controllers is used. The fake controllers are configured in the file `fake_controllers.yaml` which for Pi Robot looks like this:

```
controller_list:
  - name: fake_right_arm_controller
    joints:
      - right_arm_shoulder_pan_joint
      - right_arm_shoulder_lift_joint
      - right_arm_shoulder_roll_joint
      - right_arm_elbow_flex_joint
      - right_arm_forearm_flex_joint
      - right_arm_wrist_flex_joint

  - name: fake_right_gripper_controller
    joints:
      - right_gripper_finger_joint
```

Note that we have a controller for each planning group which in this case includes the right arm and the gripper. We will learn more about this file later in the chapter when dealing with real controllers so for now it is enough to notice that each fake controller simply lists the set of joints it controls.

The MoveIt! fake controllers take the joint configurations computed by the motion planners and simply republish them on the `/joint_states` topic. In other words, if a particular joint trajectory for the arm is computed by the motion planner, the fake controller for the arm updates the joint positions on the `/joint_states` topic with the values computed for each point along the trajectory. If we were using a real joint controller, the computed trajectory would be sent to the controller to be rendered as servo commands that move the arm in the desired way. The `/joint_states` topic would then reflect the actual motions that result, and how well the arm actually moves depends on the details of the controller and the physics of the arm.

11.11.3 The `joint_limits.yaml` file

As we know from the chapter on URDF models, the range of each joint can be restricted by defining upper and lower position limits. The velocity of a joint can also be limited in the URDF file using the `vlimit` parameter. The MoveIt! Setup Assistant creates the `joint_limits.yaml` configuration file that sets the maximum velocity for each joint based on any `vlimit` values found in the original URDF file. It also includes parameters for setting maximum joint *acceleration*. Here are the first two joints in the `joint_limits.yaml` file for Pi Robot:

```
joint_limits:  
  right_arm_shoulder_pan_joint:  
    has_velocity_limits: true  
    max_velocity: 3.14  
    has_acceleration_limits: false  
    max_acceleration: 0  
  
  right_arm_shoulder_lift_joint:  
    has_velocity_limits: true  
    max_velocity: 3.14  
    has_acceleration_limits: false  
    max_acceleration: 0
```

The `has_velocity_limits` parameter is set to `true` if there is a velocity limit specified in the original URDF model. The value for the `maximum_velocity` parameter is then taken from the URDF file. The parameter `has_acceleration_limits` is set to `false` by default for each joint and the `max_acceleration` is set to 0.

If you find that your robot's arm does not move quickly enough, try changing the parameters in this file. We will see later how we can always *slow down* the motion even without making changes here. But to get faster motion, these are the parameters to tweak. Start by setting the `has_acceleration_limits` parameters to `true` and setting the `max_acceleration` value to something greater than 0. Start with low values like something in between 1.0 and 3.0. After the changes, you will need to

terminate any running `move_group.launch` file and then start it up again to load the new values.

Finally, note that the underlying joint trajectory controller might also impose velocity limits of its own. For example, as we learned in Chapter 5, the ArbotiX controller accepts a `max_speed` parameter for each joint and these values will override any values set in the `joint_limits.yaml` file.

11.11.4 The `kinematics.yaml` file

MoveIt! profiles a numeric kinematics solver (KDL) that should work with any kinematic chain you define for your robot. The kinematic parameters used with the solver are stored in the file `kinematics.yaml` that, like the other configuration files, is found in the `config` subdirectory of your robot's MoveIt! package. Generally speaking, you won't have to touch this file since it is already customized for your robot. The file is loaded by the `planning_context.launch` file which in turn is run when launching your robot's `move_group.launch` file.

The syntax of the `kinematics.yaml` file is fairly simple and assigns an IK plugin plus a few parameters for each planning group created when running the Setup Assistant. For a single-arm robot like Pi Robot, the file would look like this:

```
right_arm:  
  kinematics_solver: kdl_kinematics_plugin/KDLKinematicsPlugin  
  kinematics_solver_attempts: 3  
  kinematics_solver_search_resolution: 0.005  
  kinematics_solver_timeout: 0.05
```

Recall that the name of the planning group (here called `right_arm`) as well as the kinematics solver and its parameters were chosen when running the Setup Assistant. There is usually no need to change these and if you do, it is usually better to re-run the Setup Assistant rather than editing the file manually.

Here is another example `kinematics.yaml` file, this time for the PR2 which has two arms and telescoping torso:

```
right_arm:  
  kinematics_solver: pr2_arm_kinematics/PR2ArmKinematicsPlugin  
  kinematics_solver_attempts: 3  
  kinematics_solver_search_resolution: 0.001  
  kinematics_solver_timeout: 0.05  
right_arm_and_torso:  
  kinematics_solver: kdl_kinematics_plugin/KDLKinematicsPlugin  
  kinematics_solver_attempts: 3  
  kinematics_solver_search_resolution: 0.001  
  kinematics_solver_timeout: 0.05  
left_arm:
```

```

kinematics_solver: pr2_arm_kinematics/PR2ArmKinematicsPlugin
kinematics_solver_attempts: 3
kinematics_solver_search_resolution: 0.001
kinematics_solver_timeout: 0.05
left_arm_and_torso:
  kinematics_solver: kdl_kinematics_plugin/KDLKinematicsPlugin
  kinematics_solver_attempts: 3
  kinematics_solver_search_resolution: 0.001
  kinematics_solver_timeout: 0.05

```

In this case, the left and right arms use a custom kinematics solver created specifically for the PR2 that is much faster than the stock KDL solver. Two additional planning groups are also defined which include each arm and the torso. These planning groups use the KDL solver.

From the ROS Wiki we learn the following details regarding the parameters used above:

- `kinematics_solver`:
 - `kdl_kinematics_plugin/KDLKinematicsPlugin`: The KDL kinematics plugin wraps around the numerical inverse kinematics solver provided by the [Orcos KDL](#) package.
 - This is the default kinematics plugin currently used by MoveIt!
 - It obeys joint limits specified in the URDF (and will use the safety limits if they are specified in the URDF).
- `kinematics_solver_search_resolution`: This specifies the resolution that a solver might use to search over the redundant space for inverse kinematics, e.g. using one of the joints for a 7 DOF arm specified as the redundant joint.
- `kinematics_solver_timeout`: This is a default timeout specified (in seconds) for each internal iteration that the inverse kinematics solver may perform. A typical iteration (e.g. for a numerical solver) will consist of a random restart from a seed state followed by a solution cycle (for which this timeout is applicable). The solver may attempt multiple restarts - the default number of restarts is defined by the `kinematics_solver_attempts` parameter below.
- `kinematics_solver_attempts`: The number of random restarts that will be performed on the solver. Each solution cycle after the restart will have a timeout defined by the `kinematics_solver_timeout` parameter above. In general, it is better to set the timeout low and fail quickly in an individual solution cycle.

While the default KDL kinematics solver will suffice for most of our applications, later in the chapter we will learn how to create a faster analytic solver for our arm using

[OpenRAVE](#). An analytic solver is less likely to miss a valid solution than a numerical solver. Furthermore, it tends to run much faster which can be critical if the arm is required to track a moving target or respond quickly to changes in the environment.

11.12 The move_group Node and Launch File

The `move_group` node is at the heart of the MoveIt! engine. The MoveIt! Setup Assistant creates a corresponding launch file called `move_group.launch` in the launch subdirectory of your robot's MoveIt! configuration package. This launch file runs the main `move_group` node and loads a number of parameters and plugins which in turn cause the `move_group` node to activate a variety of topics and services to handle kinematics, the planning scene, collision checking, sensor input and joint trajectory control.

Running the `move_group.launch` file is all that is required to have access to the full set of MoveIt! ROS topics and services. It also enables us to interact with MoveIt! programmatically using either the Python or C++ API. The Setup Assistant also creates a `demo.launch` file that runs the `move_group.launch` file with a set of fake joint controllers and a pre-configured `RViz` session. This allows us to test the MoveIt! setup without having a real robot or even a simulator. Let us turn to that next.

11.13 Testing MoveIt! in Demo Mode

The MoveIt! configuration package we just created includes a `demo.launch` file that will enable us to test the setup. Run the following command to launch all the MoveIt! nodes and services in demo mode:

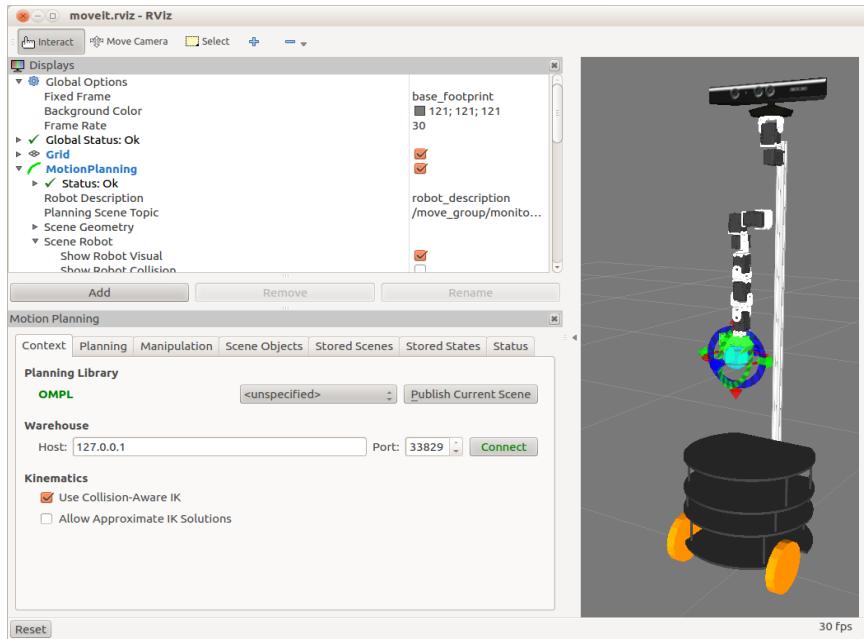
```
$ rosrun pi_robot_moveit_config demo.launch
```

NOTE: If the demo launch file aborts, type `Ctrl-C` and try launching again. If it continues to crash, the culprit is most likely `RViz`. Turning off hardware acceleration for `RViz` usually helps which can be done by issuing the following command before launching the `demo.launch` file:

```
$ export LIBGL_ALWAYS_SOFTWARE=1
```

If this solves the crashing problem, it might be a good idea to put the above command at the end of your `~/.bashrc` file so it is run automatically in new terminal sessions.

Once `RViz` successfully appears, you should see Pi Robot in the right panel and the **Motion Planning** panel on the left as shown below:

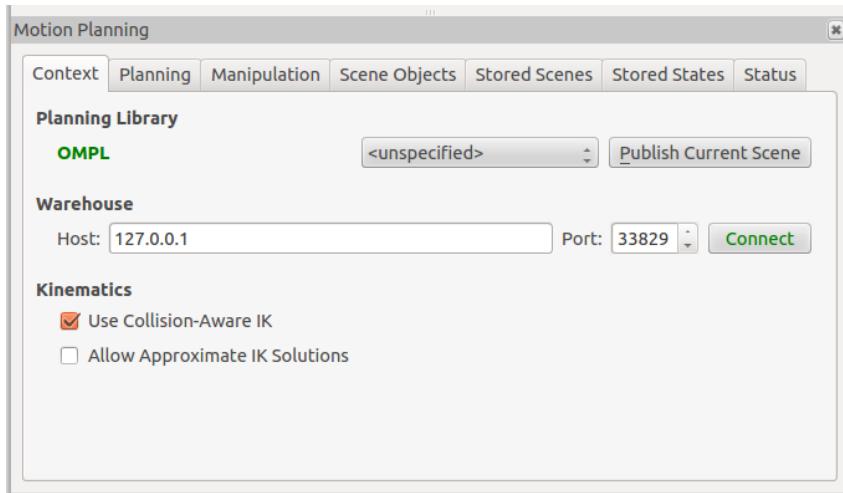


Confirm or change the settings in `RViz` to match the following:

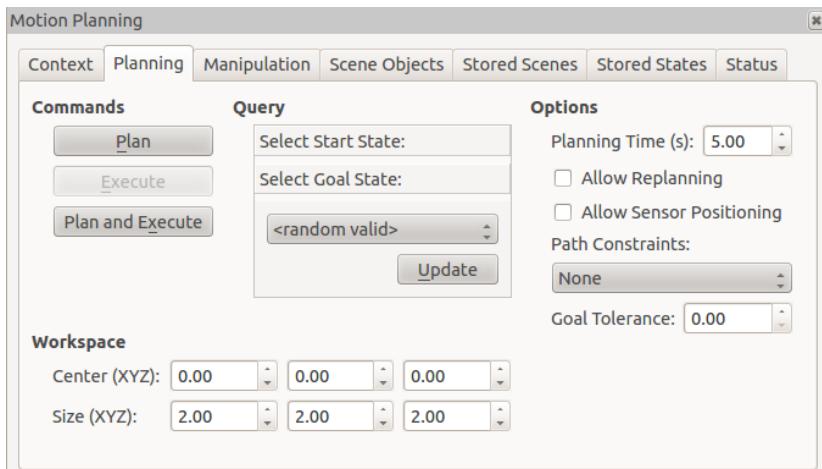
- **Global Option → Fixed Frame:** `base_footprint`
- **Motion Planning → Scene Robot** → check **Show Robot Visual**
- **Motion Planning → Planning Request → Planning Group** → `right_arm`
- **Motion Planning → Planning Request** → check **Query Start State**
- **Motion Planning → Planning Request** → check **Query Goal State**
- **Motion Planning → Planned Path** → check **Show Robot Visual**
- **Motion Planning → Planned Path** → un-check **Loop Animation**
- **Motion Planning → Planned Path** → un-check **Show Trail**
- **File → Save Config**

You can now try setting arm navigation goals for Pi Robot. For a complete description of how to do this using interactive markers, see the [Quick Start](#) guide on the MoveIt! Wiki. Below is a brief summary of the steps. Note that by default, the starting configuration of the arm is shown in green while the goal configuration is colored orange.

- Click on the **Context** tab in the Motion Planning panel and verify that **Use Collision-Aware IK** is checked as shown below:



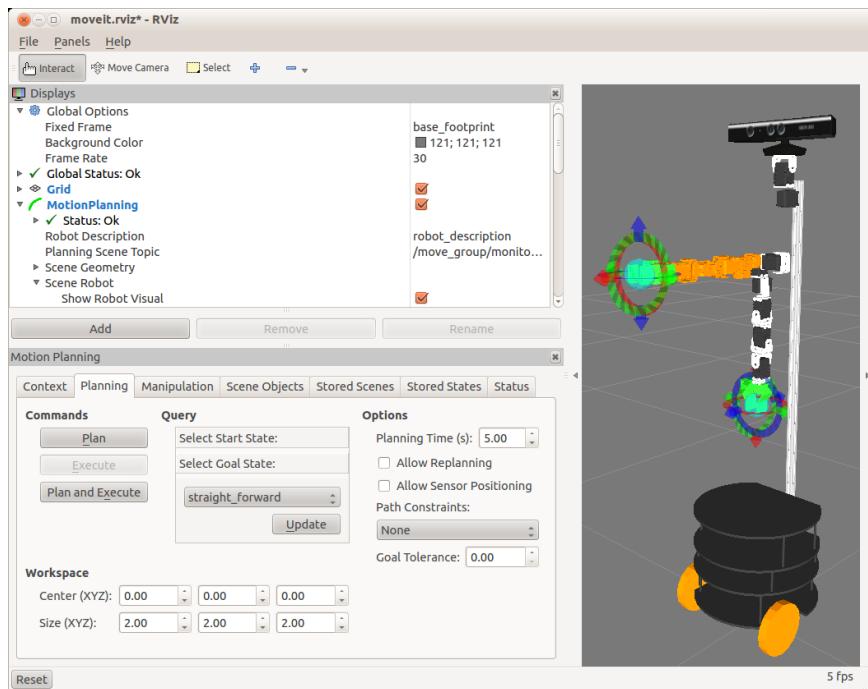
- Next, click on the **Planning** tab in the Motion Planning panel and you should see the following screen:



Click on **Select Start State** under the **Query** column. Most of the time, you will simply leave this selected on <current> but you can also use the pull down menu to select a different starting configuration of the arm. If you defined some named poses for the arm when running the Setup Assistant, they will be listed in the menu.

You can also choose a random arm configuration. When you have made your selection, click the **Update** button. The starting configuration will appear as a green arm in RViz. (If the starting configuration is <current>, the green color will probably be hidden underneath the normal display of the arm.) You can also use the interactive markers at the end of the arm to set the start position using your mouse. Note that when using the interactive markers, you do *not* click the **Update** button when you are finished positioning the arm.

- Once you have your starting configuration set, click on **Select Goal State** under the **Query** column. Use the pull down menu to select the goal configuration of the arm, then click the **Update** button. The goal configuration will appear as an orange arm in RViz. The image below indicates that we have selected the "straight_forward" configuration:



Alternatively, you can use the interactive markers on the orange colored arm to set the goal position. As noted earlier, when using the interactive markers, you do not click the **Update** button when you are finished positioning the arm. (You can also select a pre-defined pose, click **Update**, then tweak the pose with the interactive markers.)

- When you are satisfied with the start and goal states, click on the **Plan and Execute** button. The arm should move between the start and goal configurations.
- It can be instructive to choose a number of <random valid> states for the **Goal State** which will give you an idea of the range and types of trajectories to expect from your arm.

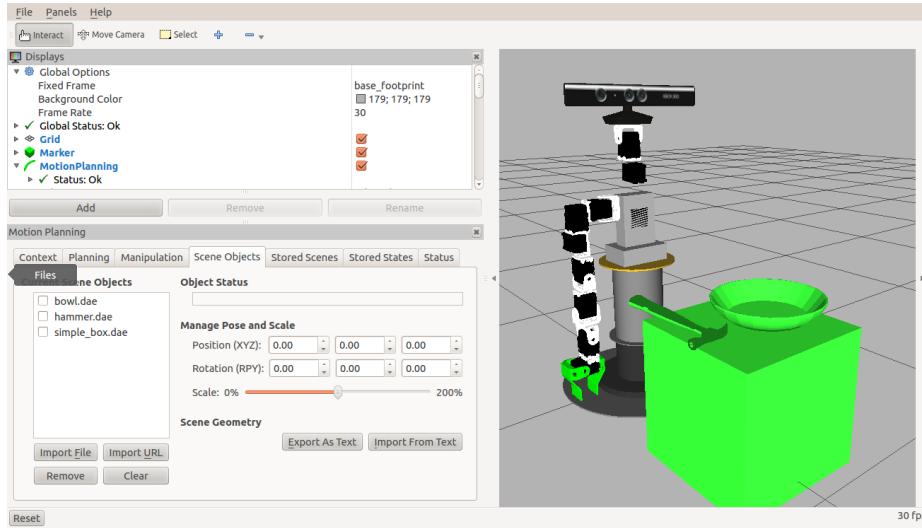
To exit demo mode at any time, type `Ctrl-C` in the terminal window in which you ran the `demo.launch` file.

11.13.1 Exploring additional features of the Motion Planning plugin

The **RViz Motion Planning** plugin has additional features that we do not have space to explore in this Volume. For example, the **Manipulation** tab is designed to be connected to an object detection pipeline so that one can run a "pick-and-place" operation from within the GUI. While we won't attempt to set up the GUI in this way, we will cover the basics of pick-and-place later in the chapter using virtual target objects and then again in the next chapter with a real robot and real objects.

The **Scene Objects** tab allow you to add mesh objects to the scene from local files or URLs and then store the result for later use. One place to look for compatible mesh files is the Gazebo database at <http://gazebosim.org/models>. For example, if you click on the model directory for `bowl` and then on the `meshes` subdirectory, you will find the mesh file `bowl.dae`. Right-click on this file and select "copy link address" or equivalent. Then use the **Import URL** button in the **Scene Objects** tab and paste the URL into the pop-up box. The bowl will appear in **Rviz**—it will probably be shown in green and rolled over on its side somewhere near or intersecting with the robot's base. You can move it around with the interactive marks or the **Position** and **Rotation** controls on the **Scene Objects** panel. You can also change the bowl's size with the **Scale** slider. Add other objects in the same way and then click on the **Export As Text** button to save the scene to a text file. You import this scene later on using the **Import From Text** button.

The next screen shot shows a scene created by adding three meshes from the `gazebosim.org` site including a box, a bowl and a hammer. The objects were then scaled and moved into position using the **Scale** slider and interactive markers on each object.



The **Stored Scenes** and **Stored States** tabs work with a Mongo database that can be launched using the `warehouse.launch` file in your robot's MoveIt! configuration package. To access the database first click the **Connect** button on the **Context** tab. If the connection is successful, the **Connect** button will turn into a **Disconnect** button. Scenes and robot states can then be saved in the database for later use. We will return to scene objects later in the chapter and show how MoveIt! can control the motion of the arm to either avoid or interact with these objects.

11.13.2 Re-running the Setup Assistant at a later time

Note that you can always re-run the Setup Assistant at a later time if you have made changes to your robot model or you want to make other modifications, such as adding more named poses. Simply run the assistant with the command:

```
$ rosrun moveit_setup_assistant setup_assistant.launch
```

Then click the button labeled **Edit Existing MoveIt Configuration Package**. A file dialog box will open. Navigate to your MoveIt! configuration directory, and click the **Open** button. Then click the **Load Files** button. Your current configuration will be loaded and you can make any changes you like. When you are finished, click the **Configuration Files** tab on the left and then click the **Generate Package** button to save your changes.

NOTE: Don't forget to recreate the static version of your robot's URDF file before re-running the Setup Assistant if you make changes to any of your model's component Xacro/URDF files.

11.14 Testing MoveIt! from the Command Line

To test the basic functionality of our robot arm, we can interact with MoveIt! from the command line using the `moveit_commander cmdline.py` utility. If you do not already have the MoveIt! `demo.launch` file running from earlier, bring it up now:

```
$ rosrun pi_robot_moveit_config demo.launch
```

Next, **un**-check the boxes beside **Query Start State** and **Query Goal State** as follows:

- **Motion Planning → Planning Request** → *un*-check **Query Start State**
- **Motion Planning → Planning Request** → *un*-check **Query Goal State**

With these settings we can more easily view the movement of the arm when sending motion commands to the MoveIt! motion planner.

Now bring up the MoveIt! command line interface by running:

```
$ rosrun moveit_commander moveit_commander.py
```

You should be placed at the command prompt:

```
Waiting for commands. Type 'help' to get a list of known commands.
```

```
>
```

All commands in MoveIt! operate on a move group as defined in the configuration files for the robot in use. In the case of Pi Robot, first select the `right_arm` group with the `use` command:

```
> use right_arm
```

After a brief delay, you should see an output similar to the following and a new command prompt for the `right_arm` group:

```
[ INFO] [1369528317.243702965]: Ready to take MoveGroup commands for
group right_arm.
OK
right_arm>
```

With the robot visible in `RViz`, let's begin by moving the arm to a random set of joint angles using the `go rand` command:

```
right_arm> go rand
```

MoveIt! will check the validity of the selected angles against the kinematic model of the arm. The target pose will be considered invalid if (1) the selected joint angles fall outside the joint limits set in the model, (2) the arm would collide with another part of the robot such as the head or torso, or (3) the motion would cause the arm to strike any nearby objects. (More on this later.) If any of these checks fail, a message similar to the following will appear:

```
[ INFO] [1369529066.236768000]: ABORTED: No motion plan found. No  
execution attempted.  
Failed while moving to random target [-0.785861312784 -0.777854562622  
-1.90569032181 -0.927753414307 -0.32391361976 0.711703415429]
```

On the other hand, if the joint goal represents a reachable configuration of the arm, MoveIt! will plan a trajectory from the starting configuration to the target configuration and pass that trajectory on to the arm's trajectory controller. The controller will then move the arm smoothly into the new position. In this case, you will see an output like the following:

```
right_arm> go rand  
Moved to random target [-1.33530406365 0.898460602916 0.184664419448  
1.20867797686 0.171914381348 1.08589163089]  
right_arm>
```

Try issuing the `go rand` command a few times to see how the arm responds to different target configurations.

Let's explore a few of the other commands. To see the current joint values and the pose of the end-effector, use the `current` command:

```
right_arm> current
```

The output should look something like the following:

```

joints = [0.0135323487558 0.544921628281 2.26598694909 0.598703264228
-0.8361580286 1.32167751045]
right_gripper_link pose = [
header:
  seq: 0
  stamp:
    secs: 1369490199
    nsecs: 690589904
  frame_id: odom
pose:
  position:
    x: 0.0117037066685
    y: -0.197628755962
    z: 0.537961695072
  orientation:
    x: 0.182884025394
    y: 0.0589242746894
    z: -0.764116809095
    w: 0.615797746964 ]
right_gripper_link RPY = [0.14494326196452958, 0.3597715164230441,
-1.7585397976479231]

```

At the top of the output we see a list of the joint angles in radians. The joints are listed in the same order as they are linked in the arm, starting at the shoulder and working toward the gripper. Next comes the pose (position and orientation) of the end-effector (gripper) relative to the `/odom` reference frame. At the end of the output is the orientation of the end-effector given in terms of roll, pitch and yaw (RPY) around the *x*, *y* and *z* axes respectively.

To record the current configuration into a variable called 'c' use the command:

```

right_arm> record c
Remembered current joint values under the name c
right_arm>

```

The goal joint values can now be tweaked using the appropriate index. For example, to set the second joint value (`right_arm_lift_joint`) to 0.2, use the command:

```
right_arm> c[1] = 0.2
```

To set the goal to the changed configuration, use the command:

```
right_arm> goal = c
```

Finally, to move the arm to the new goal configuration, make sure the robot is visible in RViz, then send the goal to the arm using the command:

```
right_arm> go goal
```

Remember that the new goal may fail if it does not pass all the checks described earlier.

If you defined one or more named poses in the Setup Assistant, you can use them in the Commander. For Pi Robot, you can send the "resting", "straight_forward" or "wave" pose:

```
right_arm> go resting
right_arm> go straight_forward
right_arm> go wave
```

To see the other commands supported by the MoveIt! Commander, type `help` at the command prompt:

```
right_arm> help
```

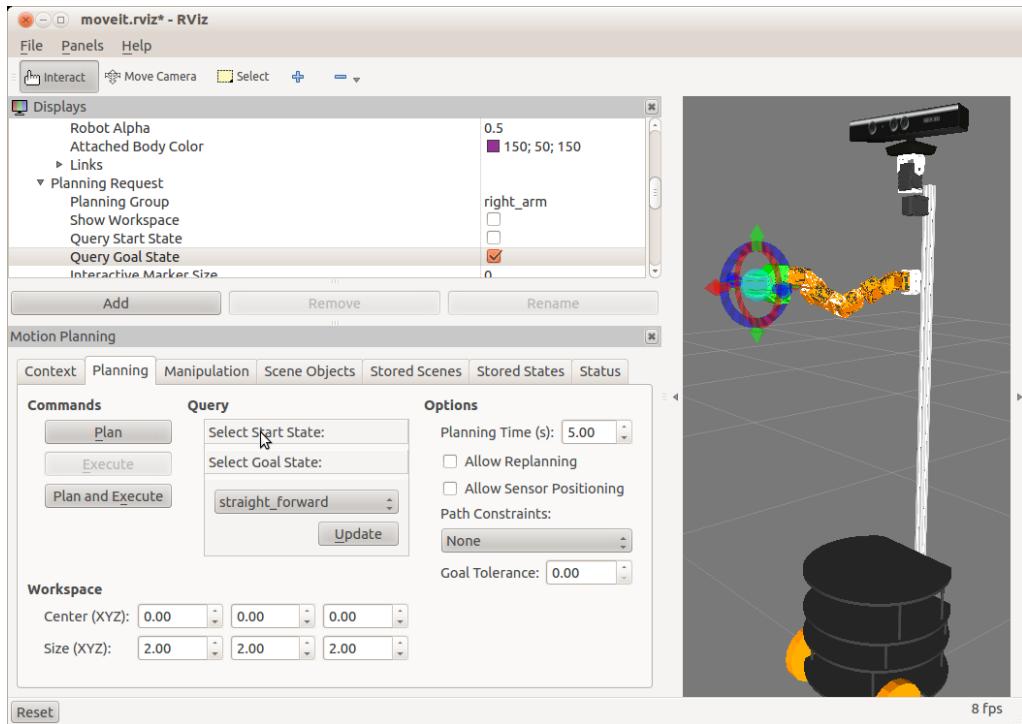
Now that we know that MoveIt! is talking to the arm, it's time to make it do something useful.

11.15 Determining Joint Configurations and End Effector Poses

Later on in the chapter, we will be writing our own Python nodes to set goal configurations for the arm or target poses for the end-effector. For goal configurations we need to know the desired joint angles, and for end-effector poses, we need to know the target position and orientation of the end-effector (e.g. gripper). These values are not always easy to guess or visualize so we need a method for obtaining them more accurately.

One option is to use MoveIt! in demo mode together with the command line interface. Using the interactive markers in `RViz`, we can position the arm or end-effector the way we want, then read off the resulting joint angles or end-effector pose using the `current` command in the MoveIt! command line interface as we did earlier.

For example, suppose we have positioned the arm using the interactive markers in `RViz` as shown below and we want to get either the joint angles or end-effector pose so we can use it later as a goal in one of our programs.



Simply bring up the MoveIt! command line interface, select the `right_arm` move group and issue the `current` command as follows:

```
$ rosrun moveit_commander moveit_commander_cmdline.py
```

```
> use right_arm
right_arm> current
```

And the output should be similar to the following:

```

joints = [-0.395309951307 -1.03650332222 -2.40611416266 1.22550181669
-1.11578483203 0.829686311888]
right_gripper_link pose = [
header:
  seq: 0
  stamp:
    secs: 1401584103
    nsecs: 673882961
  frame_id: /odom
pose:
  position:
    x: 0.130429148535
    y: -0.211274182496
    z: 0.849751498818
  orientation:
    x: 0.688193773223
    y: 0.000192406815258
    z: -0.000249462081726
    w: 0.725526864592 ]
right_gripper_link RPY = [1.517993195677934, 0.0006225491696134483,
-9.715655858643511e-05]

```

The joint angles are in the order they appear in the kinematic chain beginning with the shoulder pan joint, and the pose refers to the pose of the end-effector relative to the `/odom` frame. (This works also for the `/base_footprint` frame because of the planar virtual joint we defined connecting the `/odom` frame and the `/base_footprint` frame in the MoveIt! Setup Assistant). These numbers can be copied into your script and used as target joint angles or a target pose for the end-effector.

You can also determine the end-effector pose directly in `RViz` by expanding the **Links** tree under the **Scene Robot** and then expanding the **right_gripper** link to reveal its current position and orientation. However, using the technique above yields both the joints angles and end-effector pose and does not require all that clicking in `RViz`.

Finally, you can use the handy script called `get_arm_pose.py` found in the `rbx2_arm_nav/scripts` directory. The script uses the MoveIt! API so the robot's `move_group.launch` file must already be running. Run the script with the command:

```
$ rosrun rbx2_arm_nav get_arm_pose.py
```

The output includes the list of active joints and their current positions as well as the pose of the end-effector. These values are in a format that allows them to be copy-and-pasted into a script to be used as target values.

11.16 Using the ArbotiX Joint Trajectory Action Controllers

While it is possible to test a lot of code in MoveIt!'s demo mode, we will get better results if we use a proper joint trajectory controller. In Chapter 5 we learned how to configure the ArbotiX package to control both individual joints and joint chains that make up an arm or a pan-and-tilt head. In that chapter we did not go beyond the individual joint controllers. Now we turn our attention to the multi-joint trajectory action controllers.

Since we can run the ArbotiX node in fake mode, we can use the simulated ArbotiX controllers rather than MoveIt!'s fake controllers to test the programming samples presented in the rest of the chapter. This approach has the additional advantage of allowing us to run nearly the same code with real servos later on.

Before diving into the MoveIt! sample code, let's test the more basic function of the ArbotiX joint trajectory action controllers for the arm and head.

11.16.1 Testing the ArbotiX joint trajectory action controllers in simulation

Recall that joint trajectory action controllers respond to messages of type `FollowJointTrajectoryGoal` which are a little too long and complex to comfortably publish manually on the command line. So we will use a simple Python script instead to illustrate the process.

The script is called `trajectory_demo.py` and it can be found in the `rbx2_arm_nav/scripts` directory. Before looking at the code, let's try it out.

If you still have the MoveIt! `demo.launch` file running from an earlier session, terminate it now. Then begin by launching the one-arm version of Pi Robot in fake mode (`sim:=true`):

```
$ roslaunch rbx2_bringup pi_robot_with_gripper.launch sim:=true
```

You should see the following output on the screen:

```
process[arbotix-1]: started with pid [11850]
process[right_gripper_controller-2]: started with pid [11853]
process[robot_state_publisher-3]: started with pid [11859]
[INFO] [WallTime: 1401976945.363225] ArbotiX being simulated.
[INFO] [WallTime: 1401976945.749586] Started FollowController
(right_arm_controller). Joints: ['right_arm_shoulder_pan_joint',
'right_arm_shoulder_lift_joint', 'right_arm_shoulder_roll_joint',
'right_arm_elbow_flex_joint', 'right_arm_forearm_flex_joint',
'right_arm_wrist_flex_joint'] on C1
[INFO] [WallTime: 1401976945.761165] Started FollowController (head_controller).
Joints: ['head_pan_joint', 'head_tilt_joint'] on C2
```

The key items to note are highlighted in bold above: a controller for the right gripper is started; the ArbotiX controller is running in simulation mode; and the joint trajectory action controllers (referred to as FollowControllers by the arbotix node) are started for the right arm and head.

Next, bring up RViz with the `arm_sim.rviz` config file from the `rbx2_arm_nav` package:

```
$ rosrun rviz rviz -d `rospack find rbx2_arm_nav`/config/arm_sim.rviz
```

Now run the `trajectory_demo.py` script. Note that we are not using MoveIt! at all at this point. The script simply sends a pair of joint trajectory requests (one for the arm and one for the head) to the trajectory action controllers that were started by our launch file above. If the controllers succeed in implementing the requested trajectories, the arm and head should move upward and to the right smoothly and at the same time:

```
$ rosrun rbx2_arm_nav trajectory_demo.py _reset:=false _sync:=true
```

To move the arm and head back to their original configurations, run the command with the `reset` parameter set to `true`:

```
$ rosrun rbx2_arm_nav trajectory_demo.py _reset:=true _sync:=true
```

To move the arm first and then the head, run the script with the `sync` parameter set to `false`:

```
$ rosrun rbx2_arm_nav trajectory_demo.py _reset:=false _sync:=false
```

Let's now take a look at the code.

Link to source: [trajectory_demo.py](#)

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

```

4 import actionlib
5
6 from control_msgs.msg import FollowJointTrajectoryAction
7 from control_msgs.msg import FollowJointTrajectoryGoal
8 from trajectory_msgs.msg import JointTrajectory, JointTrajectoryPoint
9
10 class TrajectoryDemo():
11     def __init__(self):
12         rospy.init_node('trajectory_demo')
13
14         # Set to True to move back to the starting configurations
15         reset = rospy.get_param('~reset', False)
16
17         # Set to False to wait for arm to finish before moving head
18         sync = rospy.get_param('~sync', True)
19
20         # Which joints define the arm?
21         arm_joints = ['right_arm_shoulder_pan_joint',
22                         'right_arm_shoulder_lift_joint',
23                         'right_arm_shoulder_roll_joint',
24                         'right_arm_elbow_flex_joint',
25                         'right_arm_forearm_flex_joint',
26                         'right_arm_wrist_flex_joint']
27
28         # Which joints define the head?
29         head_joints = ['head_pan_joint', 'head_tilt_joint']
30
31     if reset:
32         # Set the arm back to the resting position
33         arm_goal = [0, 0, 0, 0, 0, 0]
34
35         # Re-center the head
36         head_goal = [0, 0]
37     else:
38         # Set a goal configuration for the arm
39         arm_goal = [-0.3, -2.0, -1.0, 0.8, 1.0, -0.7]
40
41         # Set a goal configuration for the head
42         head_goal = [-1.3, -0.1]
43
44         # Connect to the right arm trajectory action server
45         rospy.loginfo('Waiting for right arm trajectory controller...')
46
47         arm_client =
actionlib.SimpleActionClient('right_arm_controller/follow_joint_trajectory',
FollowJointTrajectoryAction)
48
49         arm_client.wait_for_server()
50
51         rospy.loginfo('...connected.')
52
53         # Connect to the head trajectory action server
54         rospy.loginfo('Waiting for head trajectory controller...')
55

```

```

56         head_client =
57     actionlib.SimpleActionClient('head_controller/follow_joint_trajectory',
58     FollowJointTrajectoryAction)
59
60     head_client.wait_for_server()
61
62     # Create an arm trajectory with the arm_goal as the end-point
63     arm_trajectory = JointTrajectory()
64     arm_trajectory.joint_names = arm_joints
65     arm_trajectory.points.append(JointTrajectoryPoint())
66     arm_trajectory.points[0].positions = arm_goal
67     arm_trajectory.points[0].velocities = [0.0 for i in arm_joints]
68     arm_trajectory.points[0].accelerations = [0.0 for i in arm_joints]
69     arm_trajectory.points[0].time_from_start = rospy.Duration(3.0)
70
71     # Send the trajectory to the arm action server
72     rospy.loginfo('Moving the arm to goal position...')
73
74     # Create an empty trajectory goal
75     arm_goal = FollowJointTrajectoryGoal()
76
77     # Set the trajectory component to the goal trajectory created above
78     arm_goal.trajectory = arm_trajectory
79
80     # Specify zero tolerance for the execution time
81     arm_goal.goal_time_tolerance = rospy.Duration(0.0)
82
83     # Send the goal to the action server
84     arm_client.send_goal(arm_goal)
85
86     if not sync:
87         # Wait for up to 5 seconds for the motion to complete
88         arm_client.wait_for_result(rospy.Duration(5.0))
89
90     # Create a head trajectory with the head_goal as the end-point
91     head_trajectory = JointTrajectory()
92     head_trajectory.joint_names = head_joints
93     head_trajectory.points.append(JointTrajectoryPoint())
94     head_trajectory.points[0].positions = head_goal
95     head_trajectory.points[0].velocities = [0.0 for i in head_joints]
96     head_trajectory.points[0].accelerations = [0.0 for i in head_joints]
97     head_trajectory.points[0].time_from_start = rospy.Duration(3.0)
98
99     # Send the trajectory to the head action server
100    rospy.loginfo('Moving the head to goal position...')
101
102    head_goal = FollowJointTrajectoryGoal()
103    head_goal.trajectory = head_trajectory
104    head_goal.goal_time_tolerance = rospy.Duration(0.0)
105
106    # Send the goal
107    head_client.send_goal(head_goal)
108
```

```

109     # Wait for up to 5 seconds for the motion to complete
110     head_client.wait_for_result(rospy.Duration(5.0))
111
112     rospy.loginfo('...done')

```

Let's break this down line by line:

```

6  from control_msgs.msg import FollowJointTrajectoryAction
7  from control_msgs.msg import FollowJointTrajectoryGoal
8  from trajectory_msgs.msg import JointTrajectory, JointTrajectoryPoint

```

First we need to import a number of message and action types for use with trajectories. We have already met the `FollowJointTrajectoryAction` action and `FollowJointTrajectoryGoal` message earlier in the chapter. You will recall that the key component of these objects is the specification of a joint trajectory in terms of joint positions, velocities, accelerations and efforts. We therefore also need to import the `JointTrajectory` and `JointTrajectoryPoint` message types.

```

14      # Set to True to move back to the starting configurations
15      reset = rospy.get_param('~reset', False)
16
17      # Set to False to wait for arm to finish before moving head
18      sync = rospy.get_param('~sync', True)

```

The `reset` parameter allows us to move the arm and head back to their starting positions. The `sync` parameter determines whether we run the arm and head trajectories simultaneously or one after the other.

```

20      # Which joints define the arm?
21      arm_joints = ['right_arm_shoulder_pan_joint',
22                     'right_arm_shoulder_lift_joint',
23                     'right_arm_shoulder_roll_joint',
24                     'right_arm_elbow_flex_joint',
25                     'right_arm_forearm_flex_joint',
26                     'right_arm_wrist_flex_joint']
27
28      # Which joints define the head?
29      head_joints = ['head_pan_joint', 'head_tilt_joint']

```

Next we create two lists containing the joint names for the arm and head respectively. We will need these joint names when specifying a trajectory goal.

```

31      if reset:
32          # Set the arm back to the resting position
33          arm_goal = [0, 0, 0, 0, 0, 0]
34

```

```

35         # Re-center the head
36         head_goal = [0, 0]
37     else:
38         # Set a goal configuration for the arm
39         arm_goal = [-0.3, -2.0, -1.0, 0.8, 1.0, -0.7]
40
41         # Set a goal configuration for the head
42         head_goal = [-1.3, -0.1]

```

Here we define two goal configurations in terms of joint positions, one for the arm and one for the head. There is nothing special about these particular values and you can try other joint angles as you like. However, remember that the joint positions are listed in the same order as the joint names defined above. If the `reset` parameter is set to `True` on the command line, then the goal positions are set back to their neutral positions with the arm hanging straight down and the head centered.

```

47     arm_client =
actionlib.SimpleActionClient('right_arm_controller/follow_joint_trajectory',
FollowJointTrajectoryAction)
48
49     arm_client.wait_for_server()

```

Next we create a simple action client that connects to the joint trajectory action server for the right arm. Recall that the namespace for this controller was defined in the configuration file for the `arbotix` controllers using the `action_name` parameter.

```

56     head_client =
actionlib.SimpleActionClient('head_controller/follow_joint_trajectory',
FollowJointTrajectoryAction)
57
58     head_client.wait_for_server()

```

And here we do the same for the head trajectory action server. We can now use the `arm_client` and `head_client` objects to send trajectory goals to these two joint groups.

```

62     # Create an arm trajectory with the arm_goal as a single end-point
63     arm_trajectory = JointTrajectory()
64     arm_trajectory.joint_names = arm_joints
65     arm_trajectory.points.append(JointTrajectoryPoint())
66     arm_trajectory.points[0].positions = arm_goal
67     arm_trajectory.points[0].velocities = [0.0 for i in arm_joints]
68     arm_trajectory.points[0].accelerations = [0.0 for i in arm_joints]
69     arm_trajectory.points[0].time_from_start = rospy.Duration(3.0)

```

To create a joint trajectory goal for the arm, we are actually going to use just a single point (joint configuration); namely, the goal positions we stored earlier in the `arm_goal` variable. In other words, we are going to define the trajectory by specifying only its end point. The reason we can get away with this is that a joint trajectory action server will interpolate additional joint configurations in between the starting configuration and the goal configuration we send it. While it is certainly possible to specify additional arm configurations along the way, why not let the action server do the work for us?

Line 63 above creates an empty `JointTrajectory` object. In Line 64 we fill in the joint names for this trajectory with the names of the arm joints we enumerated earlier and stored in the variable `arm_joints`. Recall that a joint trajectory consists of an array (or list in Python) of trajectory points. So in Line 65 we append an empty `JointTrajectoryPoint` that we will fill in with our goal configuration. This point has index 0 in the list of trajectory points. In Line 66, we set the position values for this point to our goal positions as stored in the `arm_goal` variable. We then set the velocities and accelerations to 0.0 for each joint since this is the end point for our trajectory so the arm will be stopped. Finally, in Line 69 we set the `time_from_start` for this point to be 3.0 seconds. This means we want the trajectory to pass through this point 3 seconds after starting the trajectory. But since this is the end-point of the trajectory, then it also means we want the entire trajectory to execute in about 3 seconds.

With our single-point goal trajectory created, we are ready to send it to the action server:

```
74     # Create an empty trajectory goal
75     arm_goal = FollowJointTrajectoryGoal()
76
77     # Set the trajectory component to the goal trajectory created above
78     arm_goal.trajectory = arm_trajectory
79
80     # Specify zero tolerance for the execution time
81     arm_goal.goal_time_tolerance = rospy.Duration(0.0)
82
83     # Send the goal to the action server
84     arm_client.send_goal(arm_goal)
85
86     if not sync:
87         # Wait for up to 5 seconds for the motion to complete
88         arm_client.wait_for_result(rospy.Duration(5.0))
```

In Line 75 we create an empty `FollowJointTrajectoryGoal` message. Then in Line 78 we set the goal's trajectory component to the arm trajectory we just created above. In Line 81 we state that we want the trajectory to finish on time, which was 3.0 seconds according to our `time_from_start` setting. If it is OK for the arm to reach its destination a little later, you can increase the tolerance here. Line 84 sends the actual trajectory goal to the action server using the `send_goal()` method on the `arm_client` object. Finally, if we are not running the arm and head trajectories at the same time, Line 88 waits up to 5 seconds for the trajectory to finish or terminate with an error.

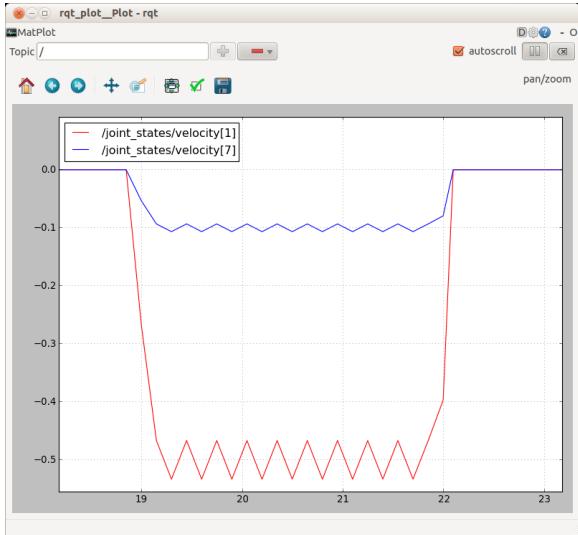
The rest of the script simply repeats the procedure for the head trajectory.

There are a few things worth noting about the trajectories generated by the script:

- Even though we only specified a single trajectory point (the ending configuration for either the arm or head joints), the `arbotix` joint trajectory action controller created intermediate joint configurations thereby enabling the arm or head to move smoothly between the starting and ending configurations.
- Each joint (either in the arm or the pan-and-tilt head) stops moving at the same time. You can confirm this visually by watching the fake robot in `RViz` or you can use `rqt_plot` to test it graphically. For example, the following command will plot the joint velocities for the `head_pan` and `head_tilt` joints over the course of the motion:

```
$ rqt_plot /joint_states/velocity[1]:velocity[7]
```

The graph below shows the pan and tilt joint velocities while running the `trajectory_demo.py` script:



Note how the motions of the two joints begin and end at the same time.

- Recall that we set the `time_from_start` for both the arm and head trajectories to be 3 seconds. In the figure above, the numbers along the x-axis represent time in seconds. You can therefore see that the head trajectory took just a little over 3 seconds.

11.16.2 Testing the ArbotiX joint trajectory controllers with real servos

For an arm configured exactly like Pi Robot's, the `trajectory_demo.py` script can be run on the real servos without modification. However, since not everyone has access to a real Dynamixel arm, let's test the simpler case of a pan-and-tilt head.

The script `head_trajectory_demo.py` in the `rbx2_dynamixels/scripts` directory is essentially the same as the `trajectory_demo.py` script that we just examined but only includes the code relevant to the head trajectory. Before running the script, we have to connect to the real servos. Terminate any fake or real Dynamixel related launch files you might already have running and bring up the head-only version of Pi Robot, or use your own launch file if you have one:

```
$ roslaunch rbx2_bringup pi_robot_head_only.launch sim:=false
```

Note how we set the `sim` argument to `false` in the command above.

Assuming your servos are detected properly by the `arbotix` driver, run the head trajectory script with the `reset` parameter set to `false`:

```
$ rosrun rbx2_dynamixels head_trajectory_demo.py _reset:=false
```

The head should rotate smoothly to the right (clockwise) and look up and it should take about 3 seconds to execute the trajectory. Re-center the servos again, but this time with a duration of 10 seconds:

```
$ rosrun rbx2_dynamixels head_trajectory_demo.py _reset:=true \
 _duration:=10.0
```

If your robot has a multi-jointed arm like Pi Robot's, you can try the `trajectory_demo.py` scripts as we did with the fake robot. Of course, you will need to modify the joint names and position values in the script assuming your arm configuration is different than Pi's.

Now that we have tested the basic function of the ArbotiX joint trajectory controllers, we are ready to configure MoveIt! to work with them.

11.17 Configuring MoveIt! Joint Controllers

There are two parts to setting up real controllers with MoveIt: (1) configuring the lower level joint trajectory controller that is used with your particular hardware (e.g. Dynamixel servos); (2) configuring the appropriate MoveIt! controller plugin that provides a more abstract connection between your physical controller and the MoveIt! API. (You can find a more in-depth [overview](#) on the MoveIt! Wiki.)

We have already learned how the ArbotiX package provides a joint trajectory action controller to use with Dynamixel servos. This controller uses a ROS [FollowJointTrajectoryAction](#) action to accept a desired joint trajectory and ensure that it is followed within a given set of tolerances. Fortunately, MoveIt! already includes a controller plugin called the [moveit_simple_controller_manager](#) that works with the `FollowJointTrajectoryAction` interface. It is also compatible with the [GripperCommandAction](#) action used by the ArbotiX package to control a variety of robot grippers. This means all we need to do is provide a link between MoveIt!'s controller configuration and the topics and action names used by the ArbotiX package.

A good reference for configuring MoveIt! joint controllers can be found on the MoveIt! [controller configuration](#) wiki page. We will provide a brief summary here.

Interfacing MoveIt! with real controllers requires these two steps:

- Creating a `controllers.yaml` file in the `config` subdirectory of the MoveIt! configuration directory for your robot.

- Creating a controller manager launch file that runs the [simple MoveIt controller manager plugin](#) and loads the parameters from the controllers.yaml file.

11.17.1 Creating the controllers.yaml file

The controllers.yaml file for Pi Robot looks like this:

```
controller_list:
  - name: right_arm_controller
    action_ns: follow_joint_trajectory
    type: FollowJointTrajectory
    default: true
    joints:
      - right_arm_shoulder_pan_joint
      - right_arm_shoulder_lift_joint
      - right_arm_shoulder_roll_joint
      - right_arm_elbow_flex_joint
      - right_arm_forearm_flex_joint
      - right_arm_wrist_flex_joint
  - name: right_gripper_controller
    action_ns: gripper_action
    type: GripperCommand
    default: true
    joints:
      - right_gripper_finger_joint
```

As you can see, this file lists a controller for each of the MoveIt! planning groups we defined using the Setup Assistant. In Pi's case, we have one for the `right_arm` group and one for the `right_gripper` group. For each group, the file includes a name for the controller, the namespace it operates in, the type of ROS action it implements, whether or not it is the default controller for this group (almost always set to `true`) and the list of joints in that group.

Let's break this down for each of our two planning groups. For the right arm we have:

```
name: right_arm_controller
action_ns: follow_joint_trajectory
```

These two parameters are taken together and specify that the name of the ROS topic that will accept commands for this controller is called `/right_arm_controller/follow_joint_trajectory`. To fully understand what this means, we need the next parameter:

```
type: FollowJointTrajectory
```

The `type` parameter indicates that the arm will be controlled using a ROS `FollowJointTrajectory` action. You can view the full definition of a `FollowJointTrajectory` action using the command:

```
$ rosmsg show control_msgs/FollowJointTrajectoryAction | less
```

which produces a rather long output beginning with:

```
control_msgs/FollowJointTrajectoryActionGoal action_goal
  std_msgs/Header header
    uint32 seq
    time stamp
    string frame_id
  actionlib_msgs/GoalID goal_id
    time stamp
    string id
control_msgs/FollowJointTrajectoryGoal goal
  trajectory_msgs/JointTrajectory trajectory
    std_msgs/Header header
      uint32 seq
      time stamp
      string frame_id
    string[] joint_names
  trajectory_msgs/JointTrajectoryPoint[] points
    float64[] positions
    float64[] velocities
    float64[] accelerations
    float64[] effort
    duration time_from_start
...
...
```

This first part of the message definition shows us the essence of what we need to know: namely, that controlling the arm requires the specification of a **joint trajectory** which in turn consists of an array of positions, velocities, accelerations and efforts at each point along the way for each joint in the arm. The parameter `time_from_start` specifies the desired timing of each joint configuration along the trajectory relative to the time stamp in the header.

Putting it all together, MoveIt! expects our `right_arm` planning group to be controlled by `FollowJointTrajectoryAction` messages published on the `/right_arm_controller/follow_joint_trajectory` action topics. This in turns means that our actual joint controller for the right arm must implement a ROS action server that consumes these types of messages and implements a callback to map `FollowJointTrajectoryGoal` messages into actual joint trajectories. Both the `arbotix_ros` and `dynamixel_motor` packages provide action servers that work with Dynamixel servos.

Turning now to the `right_gripper` group we have:

```
name: right_gripper
action_ns: gripper_action
```

```
type: GripperCommand
```

Gripper control is much simpler than controlling the rest of the arm as it typically amounts to a simple squeezing of one or two finger joints activated by one or two servos. From the three parameters listed above, MoveIt! expects the gripper to be controlled by `GripperCommandAction` goals sent to the topic `/right_gripper/gripper_action`. To see the first part of the `GripperCommandAction` definition, run the command:

```
$ rosmsg show GripperCommandAction | less
```

which should produce an output beginning with the lines:

```
control_msgs/GripperCommandActionGoal action_goal
  std_msgs/Header header
    uint32 seq
    time stamp
    string frame_id
  actionlib_msgs/GoalID goal_id
    time stamp
    string id
  control_msgs/GripperCommandGoal goal
    control_msgs/GripperCommand command
      float64 position
      float64 max_effort
```

Near the bottom of the output above we see that the gripper goal is given as a simple `float position` value specifying the desired distance between the fingers, and a `max_effort` for how hard to squeeze.

A real gripper controller must therefore define an action server that can consume `GripperCommandGoal` messages and translate the desired finger position(s) into rotation angles of the attached servos.

11.17.2 Creating the controller manager launch file

In addition to the `controllers.yaml` file, we also have to create a launch file for the overall MoveIt! controller manager. This file lives in the `launch` subdirectory of the robot's MoveIt! package and is given the name

`robot_name_moveit_controller_manager.launch.xml`. The MoveIt! Setup Assistant will create an empty version of this file so it is up to us to fill in the details. In Pi Robot's case, we edit the file

`pi_robot_moveit_controller_manager.launch.xml` to look like this:

```
<launch>
  <!-- Set the param that trajectory_execution_manager needs to find the
  controller plugin -->
```

```

<arg name="moveit_controller_manager"
default="moveit_simple_controller_manager/MoveItSimpleControllerManager" />
  <param name="moveit_controller_manager" value="$(arg
moveit_controller_manager)"/>

<!-- load controller_list -->
<rosparam file="$(find pi_robot_moveit_config)/config/controllers.yaml"/>
</launch>

```

This launch file sets the parameter `moveit_controller_manager` to the generic manager included with the MoveIt! plugin called [MoveItSimpleControllerManager](#). It then reads in our `controllers.yaml` file to set the individual controller parameters we have already defined.

Now that we have the higher level MoveIt! controller interface configured, let's turn to the joint action trajectory controllers we will use with real servos.

11.18 The MoveIt! API

The primary MoveIt! API is referred to as the **Move Group Interface** and is available in [C++](#) and [Python](#). The examples in this book use the Python API but the same concepts apply to the C++ version. At the time of this writing, the Python API is still incomplete but does include the most important functions for our purposes.

The general pattern for most of our nodes is the same:

- connect to the planning group we want to control; e.g. `right_arm`
- set either a joint space goal or a Cartesian goal (position and orientation) for the end-effector
- optionally set constraints on the motion
- ask MoveIt! to plan a trajectory to the goal
- optionally modify the trajectory such as changing its speed
- execute the planned trajectory

Let's begin with a node that uses forward kinematics to reach a goal in joint space.

11.19 Forward Kinematics: Planning in Joint Space

Sometimes you already know the joint positions you want the arm to assume. MoveIt! makes it easy to move the arm into the desired configuration. The following four lines of Python are enough to move our 6-DOF arm into a given joint configuration:

```

1 right_arm = MoveGroupCommander("right_arm")
2 joint_positions = [0.2, -0.5, 1.57, -1.0, -0.4, 0.5]
3 right_arm.set_joint_value_target(joint_positions)
4 right_arm.go()

```

The first line defines the move group to be the right arm. Then we define a list of joint positions (radians for revolute or continuous joints, meters for prismatic joints). The order of the joints is the same as the link order in the arm, starting with the base of the arm. We then use the `set_joint_value_target()` function to set the target angles to the motion planner followed by the `go()` function to execute the plan.

To get us started, take a look at the script `moveit_fk_demo.py` in the directory `rbx2_arm_nav/scripts`. This script performs the following steps:

- moves the arm to the named configuration called "resting" that was defined in the Setup Assistant and stored in the SRDF file
- opens the gripper to a neutral position
- moves the arm to a joint configuration defined by setting target angles for each joint
- assigns the target configuration a name ("saved_config") so we can easily use it again later
- closes the gripper as if grasping a small object
- moves the arm to the named configuration called "straight_forward" that was defined in the Setup Assistant and stored in the SRDF file
- moves the arm to the "saved_config" configuration saved earlier
- opens the gripper as if to release an object
- returns the arm to the named configuration called "resting" that was defined in the Setup Assistant and stored in the SRDF file
- sets the gripper back to a neutral position

Before looking at the code, let's try it out with Pi Robot in the ArbotiX simulator. Be sure to start with a clean slate so first terminate any MoveIt! launch files or RViz sessions you might already have running.

Next bring up the simulated version of Pi Robot using the ArbotiX joint controllers:

```
$ rosrun pi_robot_with_gripper.launch sim:=true
```

Note how we run the launch file with the `sim` argument set to `true`. Later on we will use the same launch file with real servos by simply setting the `sim` argument to `false`. After running the launch file you should see the following output:

```
process[arbotix-1]: started with pid [11850]
process[right_gripper_controller-2]: started with pid [11853]
process[robot_state_publisher-3]: started with pid [11859]
[INFO] [WallTime: 1401976945.363225] ArbotiX being simulated.
[INFO] [WallTime: 1401976945.749586] Started FollowController
(right_arm_controller). Joints: ['right_arm_shoulder_pan_joint',
'right_arm_shoulder_lift_joint', 'right_arm_shoulder_roll_joint',
'right_arm_elbow_flex_joint', 'right_arm_forearm_flex_joint',
'right_arm_wrist_flex_joint'] on C1
[INFO] [WallTime: 1401976945.761165] Started FollowController
(head_controller). Joints: ['head_pan_joint', 'head_tilt_joint'] on C2
```

The key items to note are highlighted in bold above: the right gripper controller is started; the `arbotix` driver is running in simulation mode; and the trajectory controllers for the right arm and head have been launched.

Next, we need to run Pi Robot's `move_group.launch` file created by the MoveIt! Setup Assistant. As we know from earlier in the chapter, this launch file brings up all the nodes and services required to interact with the robot using the MoveIt! API:

```
$ rosrun pi_robot_moveit_config move_group.launch
```

If `RViz` is already running, shut it down then bring it up again using the `arm_nav.rviz` config file from the `rbx2_arm_nav` package:

```
$ rosrun rviz rviz -d `rospack find rbt2_arm_nav`/config/arm_nav.rviz
```

Finally, open another terminal and run the forward kinematics demo script while keeping an eye on `RViz`:

```
$ rosrun rbt2_arm_nav moveit_fk_demo.py
```

You should see the robot's arm and gripper move to the various joint poses defined in the script.

Let's now look at the code.

Link to source: [moveit_fk_demo.py](#)

```

1 #!/usr/bin/env python
2
3 import rospy, sys
4 import moveit_commander
5 from control_msgs.msg import GripperCommand
6
7 class MoveItDemo:
8     def __init__(self):
9         # Initialize the move_group API
10        moveit_commander.roscpp_initialize(sys.argv)
11
12        # Initialize the ROS node
13        rospy.init_node('moveit_demo', anonymous=True)
14
15        # Set three basic gripper openings
16        GRIPPER_OPEN = [0.05]
17        GRIPPER_CLOSED = [-0.03]
18        GRIPPER_NEUTRAL = [0.01]
19
20        # Connect to the right_arm move group
21        right_arm = moveit_commander.MoveGroupCommander('right_arm')
22
23        # Connect to the right_gripper move group
24        right_gripper = moveit_commander.MoveGroupCommander('right_gripper')
25
26        # Get the name of the end-effector link
27        end_effector_link = right_arm.get_end_effector_link()
28
29        # Display the name of the end_effector link
30        rospy.loginfo("The end effector link is: " + str(end_effector_link))
31
32        # Start the arm in the "resting" configuration stored in the SRDF file
33        right_arm.set_named_target("resting")
34
35        # Plan a trajectory to the goal configuration
36        traj = right_arm.plan()
37
38        # Execute the planned trajectory
39        right_arm.execute(traj)
40
41        # Pause for a moment
42        rospy.sleep(1)
43
44        # Set the gripper to a neutral position using a joint value target
45        right_gripper.set_joint_value_target(GRIPPER_NEUTRAL)
46
47        # Plan and execute a trajectory to the goal configuration
48        right_gripper.go()
49        rospy.sleep(1)
50
51        # Set target joint values for the arm: joints are in the order they
52        # appear in the kinematic tree.
53        joint_positions = [-0.0867, 1.274, 0.02832, 0.0820, -1.273, -0.003]
54

```

```

55      # Set the arm's goal configuration to the be the joint positions
56      right_arm.set_joint_value_target(joint_positions)
57
58      # Plan and execute a trajectory to the goal configuration
59      right_arm.go()
60
61      # Pause for a moment
62      rospy.sleep(1)
63
64      # Save this configuration for later
65      right_arm.remember_joint_values('saved_config', joint_positions)
66
67      # Set the gripper target to a partially closed position
68      right_gripper.set_joint_value_target(GRIPPER_CLOSED)
69
70      # Plan and execute the gripper motion
71      right_gripper.go()
72      rospy.sleep(1)
73
74      # Set the arm target to the "straight_forward" pose from the SRDF file
75      right_arm.set_named_target("straight_forward")
76
77      # Plan and execute the motion
78      right_arm.go()
79      rospy.sleep(1)
80
81      # Set the goal configuration to the named configuration saved earlier
82      right_arm.set_named_target('saved_config')
83
84      # Plan and execute the motion
85      right_arm.go()
86
87      # Set the gripper to the open position
88      right_gripper.set_joint_value_target(GRIPPER_OPEN)
89
90      # Plan and execute the motion
91      right_gripper.go()
92      rospy.sleep(1)
93
94      # Return the arm to the named "resting" pose stored in the SRDF file
95      right_arm.set_named_target("resting")
96      right_arm.go()
97
98      # Return the gripper target to neutral position
99      right_gripper.set_joint_value_target(GRIPPER_NEUTRAL)
100     right_gripper.go()
101
102     # Cleanly shut down MoveIt!
103     moveit_commander.roscpp_shutdown()
104
105     # Exit the script
106     moveit_commander.os._exit(0)

```

Since this is our first MoveIt! script, let's take it line by line.

```
4 import moveit_commander
```

First we import the Python wrapper to the overall `moveit_commander` interface. This will give us access to the functions and objects we need to control the robot's arm and gripper.

```
5 from control_msgs.msg import GripperCommand
```

To open or close the gripper, we need the `GripperCommand` message type from the `control_msgs` package.

```
10 moveit_commander.roscpp_initialize(sys.argv)
```

When running the MoveIt! Python API, this command initializes the underlying `moveit_commander` C++ system and should always be run near the top of your script.

```
16 GRIPPER_OPEN = [0.05]
17 GRIPPER_CLOSED = [-0.03]
18 GRIPPER_NEUTRAL = [0.01]
```

Here we set three basic gripper openings. The values are in meters and specify the desired spacing in between the finger tips with `0.0` representing the half-way point. It is up to the gripper controller to map the desired opening into joint angles using the geometry of the gripper.

```
21 right_arm = moveit_commander.MoveGroupCommander('right_arm')
```

The primary object we will use from the `moveit_commander` library is the `MoveGroupCommander` class for controlling a specific move group (i.e. planning group) defined in the robot's SRDF configuration. In the case of one-arm Pi Robot, we have a move group for the right arm and another for the right gripper. Recall that we defined the names for the robot's planning groups when we ran the MoveIt! Setup Assistant. Here we initialize the right arm group by passing its name to the `MoveGroupCommander` class.

```
24 right_gripper = moveit_commander.MoveGroupCommander('right_gripper')
```

In a similar manner, we create the move group object for the right gripper.

```
27 end_effector_link = right_arm.get_end_effector_link()
```

Using the `right_arm` move group object, we can call a number of MoveIt! functions directly on the arm. In the line above, we use the function `get_end_effector_link()` to get the name of the end effector link.

```
30     rospy.loginfo("The end effector link is: " + str(end_effector_link))
```

And then we display the name on the terminal.

```
33     right_arm.set_named_target("resting")
```

Next we use the `set_named_target()` function to prepare the arm to move to the configuration called "resting" stored in the SRDF file. This particular configuration has the arm hanging straight down. Note that we haven't actually told the arm to move yet.

```
36     traj = right_arm.plan()
```

With a target configuration set, we now use the `plan()` function to compute a trajectory from the current configuration to the target configuration. The `plan()` function returns a MoveIt! `RobotTrajectory()` object that includes a joint trajectory for moving the arm toward the goal. We store this trajectory in the variable named `traj`.

```
39     right_arm.execute(traj)
```

Finally, we use the `execute()` command to actually move the arm along the planned trajectory. Notice how we pass the pre-computed trajectory to the `execute()` function as an argument. The job of actually implementing the trajectory is passed along to the underlying joint trajectory action controller which in our case is the arbotix controller. MoveIt! includes another command called `go()` that combines the `plan()` and `execute()` command into one. So instead of using:

```
traj = right_arm.plan()
right_arm.execute(traj)
```

we can simply use :

```
right_arm.go()
```

Notice how the `go()` command does not use the intermediary trajectory object. While this simplifies the code, we sometimes want to modify the planned trajectory before executing it. We will therefore usually use the `go()` command unless we need to apply some transformation to the trajectory before execution and in those cases, we will use `plan()` and `execute()`.

```
45     right_gripper.set_joint_value_target(GRIPPER_NEUTRAL)
```

Next we set a joint value target for the right gripper to the "neutral" position defined earlier in the script. Note that we haven't commanded the gripper to move yet—we do that next:

```
48     right_gripper.go()
```

Here we use the `go()` command to plan and execute the gripper trajectory.

```
53     joint_positions = [-0.0867, 1.274, 0.02832, 0.0820, -1.273, -0.003]
```

Next we prepare to move the arm into a configuration defined by a specific set of positions for each joint. First we set an array of joint positions, one for each of the six degrees of freedom of the arm. The order of the joints is the same as the order in which they are linked together in the URDF model. In the case of Pi Robot's arm, the order is:

```
['right_arm_shoulder_pan_joint', 'right_arm_shoulder_lift_joint',
'right_arm_shoulder_roll_joint', 'right_arm_elbow_flex_joint',
'right_arm_forearm_flex_joint', 'right_arm_wrist_flex_joint']
```

If you want to display the active joint names in the correct order as we have done above, or store their names in an array, you can use the `get_active_joints()` command for the planning group in question. For example, the following line would store the active joint names for the `right_arm` group in an array named `right_arm_active_joints`:

```
right_arm_active_joints = right_arm.get_active_joints()
```

With the desired joint configuration stored in the variable `joint_positions`, we are ready to set it as a goal configuration for the arm:

```
56     right_arm.set_joint_value_target(joint_positions)
```

Here we use the `set_joint_value_target()` function like we did with the gripper, only this time we are passing in values for six joints instead of one.

```
59     right_arm.go()
```

Plan and execute the arm motion.

```
62     rospy.sleep(2)
```

Pause for a moment before moving again.

```
65     right_arm.remember_joint_values('saved_config', joint_positions)
```

Here we use the `remember_joint_values()` function to store the values from the `joint_positions` array as a named configuration that we have called 'saved_config'. We will then use this named configuration later in the script to move the arm back to this position. Note that instead of the fixed `joint_positions` array, use could read the current joint values using the function `get_current_joint_values()`. This means that at any time you would like to create a named configuration, you can use a pair of commands like the following:

```
current_positions = right_arm.get_current_joint_values()
right_arm.remember_joint_values('named_target', current_positions)
```

Note that named configurations created dynamically like this are not stored in the SRDF file and will only last as long as the script is running.

```
68     right_gripper.set_joint_value_target(GRIPPER_CLOSED)
```

Close the gripper as if grasping a small object.

```
71     right_gripper.go()
```

Plan and execute the gripper motion.

```
74     # Set the arm target to the "straight_forward" pose from the SRDF file
75     right_arm.set_named_target("straight_forward")
76
77     # Plan and execute the motion
78     right_arm.go()
```

Set the arm configuration to the named "straight_forward" pose stored in the SRDF file and then plan and execute the motion.

```
81     # Set the goal configuration to the named configuration saved earlier
82     right_arm.set_named_target('saved_config')
83
84     # Plan and execute the motion
85     right_arm.go()
```

Recall that we stored one of the earlier arm configurations under the name 'saved_config'. Here we use the `set_named_target()` function to return to that pose.

```
87     right_gripper.set_joint_value_target(GRIPPER_OPEN)
88     right_gripper.go()
89     rospy.sleep(1)
```

Open the gripper as if letting go of an object.

```
94     # Return the arm to the named "resting" pose stored in the SRDF file
95     right_arm.set_named_target("resting")
96     right_arm.go()
```

Return the arm to the named "resting" pose.

```
99     right_gripper.set_joint_value_target(GRIPPER_NEUTRAL)
100    right_gripper.go()
```

Set the gripper back to the neutral position.

```
102    # Cleanly shut down MoveIt!
103    moveit_commander.roscpp_shutdown()
104
105    # Exit the script
106    moveit_commander.os._exit(0)
```

Cleanly shut down both the `moveit_commander` and the script using a pair of utility commands. It is a good idea to always end your scripts with these two commands.

11.20 Inverse Kinematics: Planning in Cartesian Space

Most of the time, we will not know the target joint angles ahead of time. Instead, we will want to specify a target pose or trajectory for the robot's gripper or end-effector in normal Cartesian space and then let the IK solver figure out the arm's joint trajectory that will place the end-effector in the desired pose.

MoveIt! allows us to plan an arm movement in any reference frame connected to the robot. For example, suppose we want to position the gripper 20 cm (0.20 meters) forward of the center of the base, 10 cm (0.1 m) to the right of center line, and 85 cm (0.85 meters) above the ground. Furthermore, we want the orientation of the gripper to be essentially horizontal. Since the z-axis of the gripper frame points upward from the plane of the gripper, this means we can just use the unit quaternion for the orientation components. The following code snippet would accomplish the task:

```
right_arm = MoveGroupCommander("right_arm")
end_effector_link = right_arm.get_end_effector_link()
```

```

target_pose = PoseStamped()
target_pose.header.frame_id = 'base_footprint'
target_pose.header.stamp = rospy.Time.now()
target_pose.pose.position.x = 0.20
target_pose.pose.position.y = -0.1
target_pose.pose.position.z = 0.85
target_pose.pose.orientation.x = 0.0
target_pose.pose.orientation.y = 0.0
target_pose.pose.orientation.z = 0.0
target_pose.pose.orientation.w = 1.0

right_arm.set_pose_target(target_pose, end_effector_link)

right_arm.go()

```

In this code snippet we begin by connecting to the right arm move group. We then get the name of the end-effector link that we will use later for setting its pose. Next, we set the target pose of the end-effector as a `PoseStamped` message relative to the `base_footprint` frame. Finally, we use the `set_pose_target()` function to set the desired end-effector pose.

To see this in action, we will use the script `moveit_ik_demo.py` in the directory `rbx2_arm_nav/scripts`. This script performs the following steps:

- moves the arm to the named configuration called "resting" that was defined in the Setup Assistant and is stored in the SRDF file
- sets the target pose described above for the end-effector relative to the `base_footprint` frame using the `set_pose_target()` function
- runs the `go()` command which uses the inverse kinematics of the arm to plan and execute a trajectory that places the end-effector in the desired pose
- shifts the end-effector 5 cm to the right using the `shift_pose_target()` function with a position argument
- rotates the gripper 90 degrees using the `shift_pose_target()` function with an orientation argument
- stores this new pose of the end-effector using the `get_current_pose()` function so we can come back to it later
- moves the arm to the named configuration called "wave" that was defined in the Setup Assistant and stored in the SRDF file

- calls `set_pose_target()` on the pose saved earlier followed by the `go()` command to move the gripper back to the saved posed
- returns the arm to the named configuration called "resting" that was defined in the Setup Assistant and stored in the SRDF file

Before looking at the code, let's try it in the ArbotiX simulator. If you don't already have the Pi Robot's launch file running from the previous section, fire it up now:

```
$ rosrun arbotiX_pi pi_robot.launch sim:=true
```

If you don't already have the robot's `move_group.launch` file running, run it now as well:

```
$ rosrun arbotiX_pi pi_robot_moveit_config move_group.launch
```

And if you don't have `RViz` running with the `arm_nav.rviz` config file, run the following command:

```
$ rosrun rviz rviz -d `rospack find arbotiX_pi`/config/arm_nav.rviz
```

Finally, run the inverse kinematics demo script:

```
$ rosrun arbotiX_pi moveit_ik_demo.py
```

You should see the robot's arm move to the various poses defined in the script as well as any messages displayed in the terminal window.

If you observe the terminal window where we ran the `move_group.launch` file, you will see messages regarding the number of planning attempts and the time it took to find a solution for each arm movement. For example, the following messages are displayed when computing the IK solution required to move the arm from the initial resting position to the first pose:

```

[ INFO] [1403875537.220921508]: No planner specified. Using default.
[ INFO] [1403875537.221527386]: RRTConnect: Starting with 1 states
[ INFO] [1403875537.734205788]: RRTConnect: Created 4 states (2 start +
2 goal)
[ INFO] [1403875537.734287309]: Solution found in 0.513139 seconds
[ INFO] [1403875537.737509841]: Path simplification took 0.003135
seconds
[ INFO] [1403875537.745549132]: Received new trajectory execution
service request...
[ INFO] [1403875540.335332624]: Execution completed: SUCCEEDED

```

The line highlighted in bold above indicates that it took over half a second to compute the IK solution using the generic KDL solver. Typical solution times for different poses range from 0.15 to 0.6 seconds on my i5 laptop. At the end the chapter we will learn to how to create a custom IK solver for the arm that reduces these computation times by a factor of 10 or more.

Let's now look at the `move_ik_demo.py` script in detail.

Link to source: [moveit_ik_demo.py](#)

```

1  #!/usr/bin/env python
2
3  import rospy, sys
4  import moveit_commander
5  from moveit_msgs.msg import RobotTrajectory
6  from trajectory_msgs.msg import JointTrajectoryPoint
7
8  from geometry_msgs.msg import PoseStamped, Pose
9  from tf.transformations import euler_from_quaternion, quaternion_from_euler
10
11 class MoveItDemo:
12     def __init__(self):
13         # Initialize the move_group API
14         moveit_commander.roscpp_initialize(sys.argv)
15
16         rospy.init_node('moveit_demo')
17
18         # Initialize the MoveIt! commander for the right arm
19         right_arm = moveit_commander.MoveGroupCommander('right_arm')
20
21         right_arm.set_end_effector_link('right_gripper_link')
22
23         # Get the name of the end-effector link
24         end_effector_link = right_arm.get_end_effector_link()
25
26         # Set the reference frame for pose targets
27         reference_frame = 'base_footprint'
28
29         # Set the right arm reference frame accordingly
30         right_arm.set_pose_reference_frame(reference_frame)

```

```

31      # Allow replanning to increase the odds of a solution
32      right_arm.allow_replanning(True)
33
34      # Allow some leeway in position (meters) and orientation (radians)
35      right_arm.set_goal_position_tolerance(0.01)
36      right_arm.set_goal_orientation_tolerance(0.1)
37
38      # Start the arm in the "resting" pose stored in the SRDF file
39      right_arm.set_named_target("resting")
40      right_arm.go()
41      rospy.sleep(2)
42
43      # Set the target pose for the end-effector
44      target_pose = PoseStamped()
45      target_pose.header.frame_id = reference_frame
46      target_pose.header.stamp = rospy.Time.now()
47      target_pose.pose.position.x = 0.20
48      target_pose.pose.position.y = -0.1
49      target_pose.pose.position.z = 0.85
50      target_pose.pose.orientation.x = 0.0
51      target_pose.pose.orientation.y = 0.0
52      target_pose.pose.orientation.z = 0.0
53      target_pose.pose.orientation.w = 1.0
54
55      # Set the start state to the current state
56      right_arm.set_start_state_to_current_state()
57
58      # Set the goal pose of the end effector to the stored pose
59      right_arm.set_pose_target(target_pose, end_effector_link)
60
61      # Plan a trajectory to the target pose
62      traj = right_arm.plan()
63
64      # Execute the planned trajectory
65      right_arm.execute(traj)
66
67      # Pause for a second
68      rospy.sleep(1)
69
70      # Shift the end-effector to the right 10cm
71      right_arm.shift_pose_target(1, -0.1, end_effector_link)
72      right_arm.go()
73      rospy.sleep(1)
74
75      # Rotate the end-effector 90 degrees
76      right_arm.shift_pose_target(3, -1.57, end_effector_link)
77      right_arm.go()
78      rospy.sleep(1)
79
80      # Store this pose for later use
81      saved_target_pose = right_arm.get_current_pose(end_effector_link)
82
83      # Move to the named pose "wave"
84

```

```

85     right_arm.set_named_target("wave")
86     right_arm.go()
87     rospy.sleep(1)
88
89     # Go back to the stored target
90     right_arm.set_pose_target(saved_target_pose, end_effector_link)
91     right_arm.go()
92     rospy.sleep(1)
93
94     # Finish up in the resting position
95     right_arm.set_named_target("resting")
96     right_arm.go()
97     rospy.sleep(2)
98
99     # Shut down MoveIt! cleanly
100    moveit_commander.roscpp_shutdown()
101
102    # Exit MoveIt!
103    moveit_commander.os._exit(0)
104
105 if __name__ == "__main__":
106     MoveItDemo()

```

The first part of the script is similar to the `moveit_fk_demo.py` node so we will skip ahead to what is new.

```

26     # Set the reference frame for pose targets
27     reference_frame = 'base_footprint'
28
29     # Set the right arm reference frame accordingly
30     right_arm.set_pose_reference_frame(reference_frame)

```

It is a good idea to explicitly set the reference frame relative to which you will set pose targets for the end-effector. In this case we use the `base_footprint` frame.

```
33     right_arm.allow_replanning(True)
```

If the `allow_replanning()` function is set to `True`, MoveIt! will try up to five different plans to move the end-effector to the desired pose. If set to `False`, it will try only one. If it is especially important that the end-effector make it to the target pose, then set this to `True`.

```

36     right_arm.set_goal_position_tolerance(0.01)
37     right_arm.set_goal_orientation_tolerance(0.05)

```

When asking the IK solver to come up with a solution for the desired end-effector pose, we can make the problem easier or harder by setting position and orientation tolerances that we are willing to accept on the final pose. In the first line above we have set a fairly tight tolerance of 1.0 cm (0.01 meters) on the position of the end effector. The second line sets an orientation tolerance of about 3 degrees. Choose your tolerances to match the task at hand.

```
45     target_pose = PoseStamped()
46     target_pose.header.frame_id = reference_frame
47     target_pose.header.stamp = rospy.Time.now()
48     target_pose.pose.position.x = 0.20
49     target_pose.pose.position.y = -0.1
50     target_pose.pose.position.z = 0.85
51     target_pose.pose.orientation.x = 0.0
52     target_pose.pose.orientation.y = 0.0
53     target_pose.pose.orientation.z = 0.0
54     target_pose.pose.orientation.w = 1.0
```

Here we set a target pose for the end-effector relative to the `reference_frame` we set earlier. This particular pose has the gripper oriented horizontally and positioned 0.85 meters off the ground, 0.10 meters to the right of the torso and 0.20 meters forward of the center of the base.

```
57     right_arm.set_start_state_to_current_state()
```

Before sending the target pose to the IK solver, it is a good idea to explicitly set the start state to the current state of the arm.

```
57     right_arm.set_pose_target(target_pose, end_effector_link)
```

Next we use the `set_pose_target()` function to set the target pose for the end-effector. This function takes a pose as the first argument and the name of the end effector link which we stored earlier in the script in the variable `end_effector_link`. As with the `set_joint_value_target()` function we used in the FK demo, the `set_pose_target()` function does not initiate planning or cause the arm to move so we take care of that next.

```
62     # Plan a trajectory to the target pose
63     traj = right_arm.plan()
64
65     # Execute the planned trajectory
66     right_arm.execute(traj)
```

First we call the `plan()` function to get back a trajectory that will move the end-effector to the pose set by the previous `set_pose_target()` statement. Then we call the `execute()` function on the returned trajectory. If the planner or IK solver fails to find a valid trajectory, the variable `traj` will be empty (`None`) and the `execute()` command will fail with a message similar to:

```
[ WARN] [1398085882.470952540]: Fail: ABORTED: No motion plan found. No execution attempted.
```

The planner will attempt to find a solution up to 5 times if the `allow_replanning()` function is set to `True` earlier in the script. Otherwise it will abort entirely after the first failed attempt. If you want to test to see if all plans have failed, simply check that the variable `traj` is not `None` before running `execute()`.

```
71      # Shift the end-effector to the right 5cm  
72      right_arm.shift_pose_target(1, -0.05, end_effector_link)
```

Occasionally it is useful to shift the position or orientation of the end-effector without specifying an entirely new pose. For example, perhaps the gripper is already nearly in the correct pose to pick up an object but we want to shift it a little to the right. The `shift_pose_target()` command shown above attempts to move the end-effector 0.05 meters (5 cm) to the right of the current location. The first argument determines which axis (translation or rotation) the shift should be applied to. The axes are defined in the order $0, 1, 2, 3, 4, 5 \rightarrow x, y, z, r, p, y$ where r, p, y stand for roll, pitch and yaw. So the command above calls for a translational shift along the y axis.

```
73      right_arm.go()
```

To execute the shift, we use the `go()` command. As before, we could call the `plan()` and `execute()` command separately or combine them as we have here.

```
74      right_arm.shift_pose_target(3, -1.57, end_effector_link)  
75      right_arm.go()
```

Here we use the `shift_pose_target()` function again but this time we rotate the end-effector 90 degrees around the roll axis. (The roll axis is the same as the x -axis in the frame of the end-effector.)

```
82      saved_target_pose = right_arm.get_current_pose(end_effector_link)
```

Now that we have shifted the end-effector in position and orientation, we can assign the new pose to a variable so that we can get back to the same pose later on. Here we use the `get_current_pose()` function to save the current pose to the variable `saved_target_pose`. Note that instead of saving the pose of the end-effector, we could use the `remember_joint_values()` function as we did in the previous demo to save the joint angles for the entire arm. The advantage of saving the pose rather than the joint angles is that if the environment changes, the IK solver might be able to find a different set of joint angles that places the end-effector in the same pose.

```
85     right_arm.set_named_target("wave")
86     right_arm.go()
```

Here we move the arm to a named configuration called "wave" stored in the SRDF file.

```
90     right_arm.set_pose_target(saved_target_pose, end_effector_link)
91     right_arm.go()
```

Next we use the `set_pose_target()` function to set the target pose to the saved pose. Running the `go()` command then calls the IK solver to find a corresponding set of joint angles and executes a trajectory that will place the end-effector in that pose.

```
95     right_arm.set_named_target("resting")
96     right_arm.go()
```

Finally, we return the arm to the "resting" configuration.

11.21 Pointing at or Reaching for a Visual Target

For a more extended example of using inverse kinematics to position the end-effector, take a look at the `arm_tracker.py` node found in the `rbx2_arm_nav/nodes` directory. This script works in a way similar to the `head_tracker.py` node we examined in the chapter on 3D head tracking, only this time instead of rotating the camera to look at the object, we will move the arm so that the robot effectively "points" or reaches toward the target.

The `arm_tracker.py` script should be fairly easy to follow given the in-line comments and its similarity to head tracking so we won't list it out here. The basic strategy is to subscribe to the `/target_pose` topic and then use the arm's inverse kinematics to position the end-effector to be as close as possible to the target. At the same time, the gripper is kept relatively horizontal to facilitate grasping in case that is the ultimate goal. If the target object is moved, a new IK solution is computed and the arm moves the gripper accordingly.

One difference between head tracking and arm tracking is that we can't update the arm position as quickly as we can the head's pan-and-tilt angles. The reason is that computing IK solutions takes a significant amount of time as we saw in the previous section. So the script checks to see if the target has moved far enough away from its last position to make it worth moving the arm again. If the target moves too far out of reach, the robot lowers its arm to the resting position. Although the script should work using MoveIt!'s default KDL IK solver, we can obtain faster and more reliable results using a custom IK solver for the robot's arm which is covered in the last section of this chapter.

The `arm_tracker.py` program also illustrates how to use the ROS `tf` library to transform target poses between different frames of reference. For example, to determine the direction in space that the arm should point or reach, we can transform the pose of the target relative to a reference frame attached to the robot's shoulder rather than the base frame. We then position the gripper somewhere along the line connecting the shoulder to the target and at a distance equal to the distance of the target to the shoulder or the length of the arm, whichever is less.

Let's test the arm tracker node using the ArbotiX simulator and the fake 3D target we used with head tracking. If you don't already have it running, bring up the simulated one-arm version of Pi Robot:

```
$ rosrun rbt2_bringup pi_robot_with_gripper.launch sim:=true
```

And if you don't already have Pi Robot's `move_group.launch` file running, fire it up now:

```
$ rosrun pi_robot_moveit_config move_group.launch
```

Now bring up `RViz` with the `fake_target.rviz` config file from the `rbt2_utils` package:

```
$ rosrun rviz rviz -d `rospack find rbt2_utils`/fake_target.rviz
```

Next, run the fake 3D target at a relatively slow moving speed:

```
$ rosrun rbt2_utils pub_3d_target.launch speed:=0.2
```

The yellow balloon should appear in `RViz` and move slowly in front of the robot. Recall that the `pub_3d_target.py` node publishes the pose of the balloon on the `/target_pose` topic.

Just for the fun of it, let's bring up the head tracker node in simulation mode so that Pi's head will track the target:

```
$ rosrun rbx2_dynamixels head_tracker.launch sim:=true
```

After a few moments, the head should start tracking the balloon in RViz.

Finally, start up the `arm_tracker.py` node:

```
$ rosrun rbx2_arm_nav arm_tracker.py
```

If all goes well, Pi should periodically update the position of his arm so that the gripper reaches toward the balloon.

11.22 Setting Constraints on Planned Trajectories

In our scripts so far, we have used the `set_pose_target()` function followed by `plan()` and `execute()` or simply `go()` to move the end-effector from one pose to another. The specific trajectory that the end-effector takes through space has been left up to the underlying planner. This trajectory can sometimes take a rather roundabout way to get the gripper from point A to point B, partly due to the random sampling methods inherent in the planning process.

In this section, we will illustrate two basic techniques for gaining more control over the path the end-effector takes through space. First we will explore the use of Cartesian paths and waypoints. Then we will look at applying constraints on the position and/or orientation of the end-effector along the trajectory.

11.22.1 Executing Cartesian Paths

Suppose we want the robot's end-effector to follow straight line trajectories through space as it moves through a sequence of poses or waypoints. MoveIt! includes the `compute_cartesian_path()` function that does the trick. Of course, the individual waypoint poses must be reachable by the IK solver or the process will fail. In the simplest case, we can use just two waypoints: the starting pose and a goal pose. The `compute_cartesian_path()` function can then be used to move the end-effector in a straight line between the two poses.

To see how it works, take a look at the script called `moveit_cartesian_demo.py` found in the `rbx2_arm_nav/scripts` directory. The script begins by moving the arm to the "straight_forward" pose stored in the SRDF. Note that we use the regular `go()` command for this first motion so we do not expect the path to be straight. We then set

this pose as our starting waypoint. Next, we set a waypoint that is located 0.2 meters back and to the right of the starting pose. The next waypoint is down, to the left and forward again. The final waypoint is back to the "straight_forward" pose. Finally, we move the arm back to the "resting" configuration using the regular `go()` command.

The waypoints are given to the `compute_cartesian_path()` function as arguments and, if all goes well, the gripper should move along a straight line between each successive pair of waypoint poses forming a kind of triangle in space.

To try it out, make sure you have the fake version of Pi Robot running in the ArbotiX simulator if it isn't already:

```
$ rosrun rbt2_bringup pi_robot_with_gripper.launch sim:=true
```

If you are not already running Pi Robot's MoveIt! launch file, bring it up now:

```
$ rosrun pi_robot_moveit_config move_group.launch
```

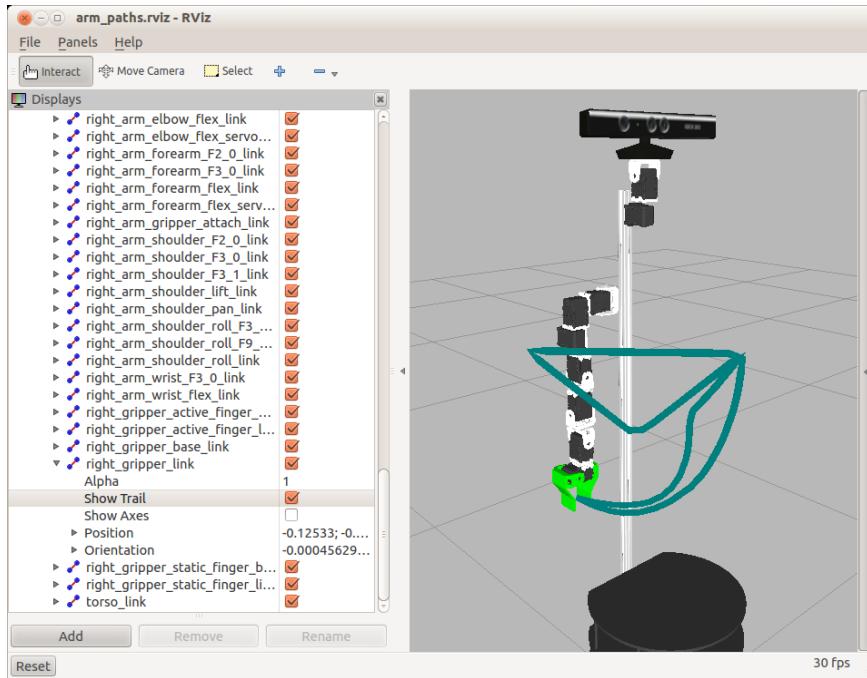
If `RViz` is already running, terminate it now then fire it up again with the `arm_paths.rviz` configuration file. (Alternatively, use the **Open Config** option under the **File** menu and navigate to `rbt2_arm_nav/config/arm_paths.rviz`.)

```
$ rosrun rviz rviz -d `rospack find rbt2_arm_nav`/config/arm_paths.rviz
```

Finally, run the Cartesian paths demo script with the `cartesian` parameter set to `true`:

```
$ rosrun rbt2_arm_nav moveit_cartesian_demo.py _cartesian:=true
```

The result in `RViz` should look something like the following:



The straight line triangle in the image is formed by the paths taken by the gripper when following the computed Cartesian path. The two curved paths are the trajectories followed when using the regular `go()` command.

NOTE: If the script fails with a message like:

```
Path planning failed with only 0.25 success after 100 attempts.
```

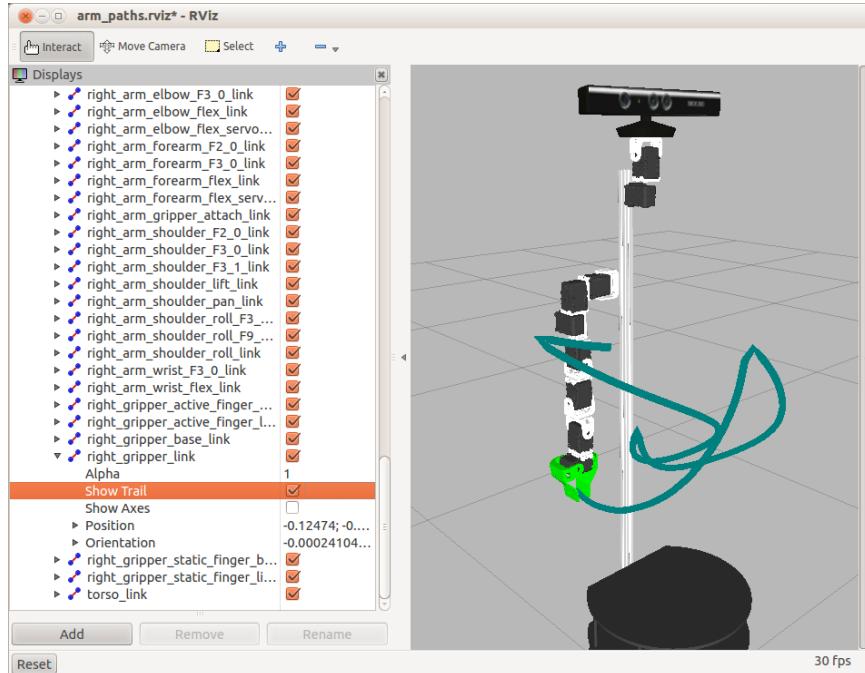
then simply run the script again. The Cartesian path planner requires that an IK solution can be found for *each* pose along the path. If the solver fails to find a solution for one or more of the waypoints, then the path is aborted. Running the script again usually finds a solution due to the random nature of the KDL IK solver.

The path is displayed in `RViz` because the `arm_paths.rviz` config checks the box beside **Show Trail** beside the `right_gripper_link` in the **Planned Path** list of links as shown in the image.

Next, try running the script with the `cartesian` parameter set to `false`. This will move the gripper to the all target poses using the usual `go()` function without first computing straight line trajectories. Before running the following command, toggle the checkbox beside the right gripper's **Show Trail** setting to clear the current path if it is still displayed. Then run:

```
$ rosrun rbx2_arm_nav moveit_cartesian_demo.py _cartesian:=false
```

This time, the view in RViz should look something like this.



Notice that the all the trajectories are now curved. Note also that the gripper no longer stays horizontal along the triangular portion of the trajectory. What's more, if you run the command again, you will get a potentially different set of trajectories in between the same waypoints. This highlights another useful feature of the `compute_cartesian_path()` function: it can move the gripper the same way time after time.

Let's now take a look at the key lines of the script.

Link to source: [moveit_cartesian_demo.py](#)

```
16    cartesian = rospy.get_param('~cartesian', True)
```

First we read in the parameter called `cartesian` that is `True` by default. If the parameter is `True`, we calculate Cartesian paths between the set of waypoints. If it is set to `False`, we use the regular `set_pose_target()` and `go()` commands to move the arm. This way we can compare the straight line trajectories with those computed the normal way.

```
33      # Get the name of the end-effector link
34      end_effector_link = right_arm.get_end_effector_link()
35
36      # Start in the "straight_forward" configuration
37      right_arm.set_named_target("straight_forward")
```

Next we get the name of the end-effector link and set the arm to the "straight_forward" configuration that we defined when running the MoveIt! Setup Assistant.

```
43      # Remember this pose so we can add it as a waypoint
44      start_pose = right_arm.get_current_pose(end_effector_link).pose
45
46      # Set the end pose to be the same as the resting pose
47      end_pose = deepcopy(start_pose)
```

After moving the arm to the "straight_forward" pose, we set the `start_pose` and `end_pose` variables to be the same as the resting pose so we can use these as the first and last waypoints.

```
49      # Initialize the waypoints list
50      waypoints = []
51
52      # Set the first waypoint to be the starting pose
53      wpose = deepcopy(start_pose)
54
55      if cartesian:
56          # Append the pose to the waypoints list
57          waypoints.append(deepcopy(wpose))
```

Next we initialize a list to hold the waypoints and set the `wpose` variable to a copy of the `start_pose`. If we are executing a Cartesian path, we then append the first waypoint to the `waypoints` list. Note that we aren't yet moving the arm—we are simply building up a list of waypoints.

```
59      # Set the next waypoint back and to the right
60      wpose.position.x -= 0.2
61      wpose.position.y -= 0.2
```

```

62     if cartesian:
63         # Append the pose to the waypoints list
64         waypoints.append(deepcopy(wpose))
65     else:
66         right_arm.set_pose_target(wpose)
67         right_arm.go()
68         rospy.sleep(2)

```

Here we set the next waypoint pose 0.2 meters back and 0.2 meters to the right of the starting pose. If the `cartesian` parameter is set to `True`, then we append this new pose to the `waypoints` list. Otherwise, we move the end-effector to the new pose right away and in whatever manner the planner chooses by using the standard `set_pose_target()` and `go()` commands.

The next few lines (that we'll skip here) simply add another waypoint that positions the end-effector down and back toward the mid-line of the robot. The final waypoint is set to the starting pose ("straight_forward").

```

95     fraction = 0.0
96     maxtries = 100
97     attempts = 0
98
99     right_arm.set_start_state_to_current_state()

```

Before we compute and execute the Cartesian path, we set a counter for the number of attempts we will allow and a variable called `fraction` that will be explained below. We also set the arm's internal start state to the current state.

```

101    # Plan the Cartesian path connecting the waypoints
102    while fraction < 1.0 and attempts < maxtries:
103        (plan, fraction) = right_arm.compute_cartesian_path(
104            waypoints,          # waypoint poses
105            0.01,               # eef_step
106            0.0,                # jump_threshold
107            True)              # avoid_collisions
108        attempts += 1
109
110        if attempts % 100 == 0:
111            rospy.loginfo("Still trying after " + str(attempts) + " "
112                           "attempts...")

```

The key statement here begins on Line 103 above and continues to Line 107. Here we invoke the function `compute_cartesian_path()` which takes four arguments:

- `waypoints` – the list of poses we want the end-effector to pass through

- `eef_step` – maximum end-effector step (in meters) allowed between consecutive positions of the end-effector along the trajectory. We have set this to 0.01 meters or 1 cm.
- `jump_threshold` – how little we want the arm to jump in configuration space. Setting this to 0.0 effectively disables this check.
- `avoid_collisions` – whether or not planning should avoid collisions. The default value for this argument is `True`.

The `compute_cartesian_path()` returns two values: the computed `plan` (trajectory) for that attempt and the `fraction` of waypoints successfully reached using that plan. For a successful plan, we want the `fraction` to be 1.0 (100%) so we wrap the `compute_cartesian_path()` function in a while loop until either the returned `fraction` is 1.0 or we run out of attempts as specified by the `maxtries` value. (While `maxtries` is set to 100 in the script, you could also pass it in as a ROS parameter.)

```

112      # If we have a complete plan, execute the trajectory
113      if fraction == 1.0:
114          rospy.loginfo("Path computed successfully. Moving the arm.")
115          right_arm.execute(plan)
116      else:
117          rospy.loginfo("Path planning failed with only " + str(fraction) +
" success after " + str(maxtries) + " attempts.")

```

Finally, if we obtain a plan that passes through 100% of the waypoints, we execute the plan. Otherwise, we display a "plan failed" message.

11.22.2 Setting other path constraints

Suppose the robot's task is to grasp and move an object while keeping it upright. For example, the object might be a container of liquid like a cup of coffee so that the cup has to be kept from tipping and spilling the fluid. MoveIt! allows us to specify path constraints for the planned trajectory that can accomplish this type of goal.

In this case, we need to constrain the orientation of the gripper so that it stays level during the motion. The script `moveit_constraints_demo.py` in the `rbx2_arm_nav/scripts` directory illustrates how it works. This script performs the following actions:

- starts the arm in the "resting" position
- moves the arm to an initial pose with the gripper about shoulder height and horizontal

- creates and sets an orientation constraint specifying a horizontal orientation for the gripper
- sets a target pose for the gripper in front of the robot and down around table height
- plans and executes a path to the target pose while satisfying the orientation constraint
- clears the constraints
- moves the arm back to the "resting" position

Keep in mind that when using path constraints, both the starting pose and target pose must satisfy the constraints or planning will fail.

Before looking at the code, let's try it out. First bring up the one arm version of Pi Robot in the ArbotiX simulator:

```
$ rosrun rbt2_bringup pi_robot_with_gripper.launch sim:=true
```

Now fire up the MoveIt! nodes for Pi:

```
$ rosrun pi_robot_moveit_config move_group.launch
```

Next, bring up RViz with the `arm_nav.rviz` config file:

```
$ rosrun rviz rviz -d `rospack find rbt2_arm_nav`/config/arm_nav.rviz
```

Finally, run the `moveit_constraints_demo.py` script:

```
$ rosrun rbt2_arm_nav moveit_constraints_demo.py
```

NOTE: Don't be alarmed if the robot appears to take a long time to perform this task or fails altogether. If the motion is successful, the gripper should move vertically downward in front of the robot keeping the gripper horizontal. However, you might notice quite a long pause (possibly as long as 2 minutes on an i5 computer) while the planner attempts to compute a path for the gripper that satisfies the constraints. Furthermore, it is not unusual for the planner to fail to find a solution at all, even after the default 5 attempts. If this happens to you, run the script again until you see the desired motion. If you monitor the terminal window in which we ran the `move_group.launch` file, you can see the status of each attempt.

The slowness of this task is due to the random sampling algorithm used by the KDL solver that requires an IK solution for each point along the computed path. Fortunately, much faster and more reliable results can be obtained by using the IKFast solver that we will explore at the end of the chapter.

Let's now look at the key lines in the `moveit_constraints_demo.py` script:

Link to source: [moveit_constraints_demo.py](#)

```
from moveit_msgs.msg import Constraints, OrientationConstraint
```

To use orientation constraints, we first import the general `Constraints` message type as well as the `OrientationConstraint` message from the MoveIt! messages package. (There is also a `PositionConstraint` message type that works in a similar way.)

```
# Increase the planning time since constraint planning can take a while
right_arm.set_planning_time(15)
```

Here we set the planning time per attempt to 15 seconds since computing IK solutions for each point along the path can take a while.

```
# Create a constraints list and give it a name
constraints = Constraints()
constraints.name = "Keep gripper horizontal"

# Create an orientation constraint for the right gripper
orientation_constraint = OrientationConstraint()
orientation_constraint.header = start_pose.header
orientation_constraint.link_name = right_arm.get_end_effector_link()
orientation_constraint.orientation.w = 1.0
orientation_constraint.absolute_x_axis_tolerance = 0.1
orientation_constraint.absolute_y_axis_tolerance = 0.1
orientation_constraint.absolute_z_axis_tolerance = 3.14
orientation_constraint.weight = 1.0
```

To create a path constraint, we first initialize a `Constraints` list and give it a name. We then start adding constraints to the list. In this case, we are going to add a single orientation constraint.

The constraint `header` is typically the same as the `header` of the pose of the arm just before applying the constraint. The `link_name` refers to the link we want to constrain which in our case is the gripper (end-effector).

We then set the desired orientation to be the unit quaternion (the x, y, and z components are zero by default.) The unit quaternion in the planning frame (`base_footprint`) refers to a horizontal orientation of the gripper with the fingers pointing in the direction of the x-axis (i.e. in the same direction that the robot faces.) If we just want to keep the gripper level and we don't care if it rotates around the z-axis during the motion, we can set the `absolute_z_axis_tolerance` to 3.14 radians which is what we have done above. However, to keep the gripper from pitching or rolling around the x and y axes, we set these tolerances fairly low.

Finally, we give this constraint a weight which would have more relevance if we were applying multiple constraints at the same time.

```
# Append the constraint to the list of constraints
constraints.orientation_constraints.append(orientation_constraint)

# Set the path constraints on the right_arm
right_arm.set_path_constraints(constraints)
```

In these next two lines, we first append the orientation constraint to the list of constraints and then, since that is the only constraint we are going to apply, we use the `set_path_constraints()` function to apply the constraints to the `right_arm` move group.

```
# Set a target pose for the arm
target_pose = PoseStamped()
target_pose.header.frame_id = REFERENCE_FRAME
target_pose.pose.position.x = 0.173187824708
target_pose.pose.position.y = -0.0159929871606
target_pose.pose.position.z = 0.692596608605
target_pose.pose.orientation.w = 1.0

# Set the start state and target pose, then plan and execute
right_arm.set_start_state_to_current_state()
right_arm.set_pose_target(target_pose, end_effector_link)
right_arm.go()
rospy.sleep(1)
```

We then set the next target pose and execute the trajectory as usual. The planner sees that we have applied the constraint on the `right_arm` group and therefore incorporates the constraint into the planning.

```
right_arm.clear_path_constraints()
```

When you want to go back to regular planning, clear all constraints using the `clear_path_constraints()` function.

11.23 Adjusting Trajectory Speed

At the time of this writing, MoveIt! does not provide a direct way to control the speed at which a trajectory is executed. As we explained earlier in the chapter, trajectory speed is determined more indirectly using the maximum joint speeds set in the robot's URDF model and the `max_velocity` and `max_acceleration` values set in the `joint_limits.yaml` configuration found in the `config` subdirectory of your robot's MoveIt! package. But since these are maximum values, they are set once for all trajectories.

It is relatively straightforward to implement our own trajectory speed control. Essentially all we need to do is scale the velocities, accelerations and `time_from_start` for each point along the trajectory. The only limitation is that we can only *slow down* trajectory speeds this way since the default speed is already as fast as the `joint_limits.yaml` file will allow.

We will create a function called `scale_trajectory_speed()` and store it in a separate Python module so that it can be used in any script it might be needed. The module is the file `arm_utils.py` located in the directory `rbx2_arm_nav/src/rbx2_arm_nav`. Simply import this module whenever you want access to the `scale_trajectory_speed()` function. (Details below.)

To see the function in action, first launch the fake version of Pi Robot if it is not already running:

```
$ roslaunch rbt2_bringup pi_robot_with_gripper.launch sim:=true
```

Next, bring up Pi Robot's `move_group.launch` file if it is not already running:

```
$ roslaunch pi_robot_moveit_config move_group.launch
```

Now bring up RViz with the `arm_nav.rviz` config file:

```
$ rosrun rviz rviz -d `rospack find rbt2_arm_nav`/config/arm_nav.rviz
```

Finally, run the `moveit_speed_demo.py` script:

```
$ rosrun rbt2_arm_nav moveit_speed_demo.py
```

The robot should first move the arm to the straight out position at normal speed, then back to the resting position at roughly $\frac{1}{4}$ normal speed.

Let's now look at the code:

Link to source: [arm_utils.py](#)

```
1 #!/usr/bin/env python
2
3 import rospy
4 from moveit_msgs.msg import RobotTrajectory
5 from trajectory_msgs.msg import JointTrajectoryPoint
6 from geometry_msgs.msg import PoseStamped, Pose
7
8 def scale_trajectory_speed(traj, scale):
9     # Create a new trajectory object
10    new_traj = RobotTrajectory()
11
12    # Initialize the new trajectory to be the same as the input trajectory
13    new_traj.joint_trajectory = traj.joint_trajectory
14
15    # Get the number of joints involved
16    n_joints = len(traj.joint_trajectory.joint_names)
17
18    # Get the number of points on the trajectory
19    n_points = len(traj.joint_trajectory.points)
20
21    # Store the trajectory points
22    points = list(traj.joint_trajectory.points)
23
24    # Cycle through all points and joints and scale the time from start,
25    # speed and acceleration
26    for i in range(n_points):
27        point = JointTrajectoryPoint()
28
29        # The joint positions are not scaled so pull them out first
30        point.positions = traj.joint_trajectory.points[i].positions
31
32        # Next, scale the time_from_start for this point
33        point.time_from_start =
traj.joint_trajectory.points[i].time_from_start / scale
34
35        # Get the velocities for each joint for this point
36        point.velocities = list(traj.joint_trajectory.points[i].velocities)
37
38        # Get the accelerations for each joint for this point
39        point.accelerations =
list(traj.joint_trajectory.points[i].accelerations)
40
41        # Scale the velocity and acceleration for each joint at this point
42        for j in range(n_joints):
43            point.velocities[j] = point.velocities[j] * scale
44            point.accelerations[j] = point.accelerations[j] * scale * scale
45
46        # Store the scaled trajectory point
47        points[i] = point
48
49    # Assign the modified points to the new trajectory
```

```

50         new_traj.joint_trajectory.points = points
51
52     # Return the new trajecotry
53     return new_traj

```

Let's break down the code line by line.

```

4  from moveit_msgs.msg import RobotTrajectory
5  from trajectory_msgs.msg import JointTrajectoryPoint

```

First we import the `RobotTrajectory` and `JointTrajectoryPoint` objects. As we know from earlier in the chapter, a robot trajectory is made up of a number of trajectory points and each point holds the position, velocity, acceleration and effort for each joint at that point.

```

8  def scale_trajectory_speed(traj, scale):
9      # Create a new trajectory object
10     new_traj = RobotTrajectory()
11
12     # Initialize the new trajectory to be the same as the input trajectory
13     new_traj.joint_trajectory = traj.joint_trajectory

```

Here we begin the definition of the function `scale_trajectory_speed()` that takes a trajectory and scale factor as arguments. First we create a new trajectory called `new_traj` that will hold the modified trajectory and we initialize the trajectory points with those from the input trajectory.

```

15     # Get the number of joints involved
16     n_joints = len(traj.joint_trajectory.joint_names)
17
18     # Get the number of points on the trajectory
19     n_points = len(traj.joint_trajectory.points)
20
21     # Store the trajectory points
22     points = list(traj.joint_trajectory.points)

```

Next we get a count of the number of joints and the number of points in the trajectory. We convert the trajectory points to a Python list so we can enumerate through them.

```

26     for i in range(n_points):
27         point = JointTrajectoryPoint()
28
29         # The joint positions are not scaled so pull them out first
30         point.positions = traj.joint_trajectory.points[i].positions
31
32         # Next, scale the time_from_start for this point
33         point.time_from_start =
traj.joint_trajectory.points[i].time_from_start / scale

```

For each point on the trajectory, we start by initializing a new trajectory point to hold the modifications. Scaling does not affect the joint positions along the trajectory so we pull them from the original trajectory unmodified. Then we adjust the `time_from_start` by the scale factor. The smaller the scale factor (the slower we want the trajectory to be executed), the longer the `time_from_start` for each point which is why we divide rather than multiply by the scale.

```
35         # Get the joint velocities for this point
36         point.velocities = list(traj.joint_trajectory.points[i].velocities)
37
38         # Get the joint accelerations for this point
39         point.accelerations =
40     list(traj.joint_trajectory.points[i].accelerations)
```

We then pull out the list of joint velocities and accelerations for this point so that we can scale them next.

```
42     for j in range(n_joints):
43         point.velocities[j] = point.velocities[j] * scale
44         point.accelerations[j] = point.accelerations[j] * scale * scale
```

Here we scale the joint velocities and accelerations for this trajectory point. Note how accelerations get multiplied by the square of the scale since acceleration is the rate of change of velocity.

```
47     points[i] = point
```

With the scaling complete, we store the modified trajectory point as the i^{th} point in the new trajectory.

```
50     new_traj.joint_trajectory.points = points
```

When all points have been scaled, we assign the entire list to the new trajectory.

```
53     return new_traj
```

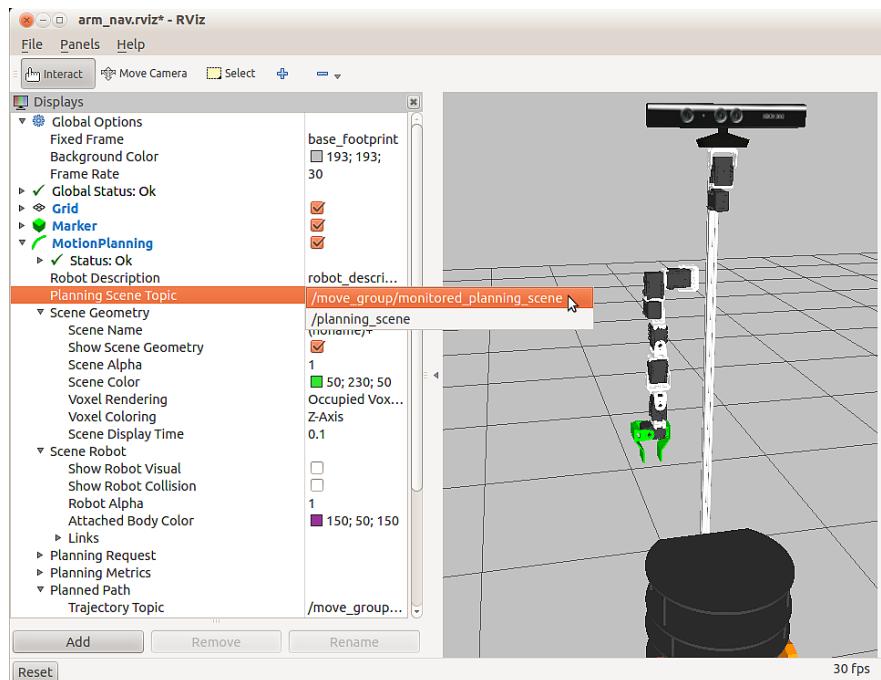
Finally, we return the new trajectory to the calling program.

11.24 Adding Obstacles to the Planning Scene

One of the more powerful features of MoveIt! is the ability to plan arm motions around obstacles. Bear in mind that to plan a successful reaching trajectory, MoveIt! must not

allow *any* part of the arm to strike an object or another part of the robot. Think of reaching for the butter at the dinner table without bumping a wine glass near your elbow. This is a complex problem in geometry and inverse kinematics but MoveIt! makes it relatively easy for us to implement.

Our next demo script, `moveit_obstacles_demo.py`, begins by placing some simulated obstacles into the planning scene. The key to viewing such obstacles in RViz is to select the `/move_group/monitored_planning_scene` topic for the **Planning Scene Topic** in the **Motion Planning** display as shown below:



The obstacles initially appear all the same color in RViz which is determined by the setting under the **Motion Planning** display named **Scene Geometry → Scene Color** also shown above. Our script sets different colors for individual objects. Real obstacles can be added to the planning scene either from existing knowledge of the robot's environment, or from sensor data such as laser scans and point clouds as we will see later in the chapter.

Before we look at the code, let's run the simulation. If you're not already running the one-arm version of Pi Robot in the ArbotiX simulator, bring it up now:

```
$ roslaunch rbx2_bringup pi_robot_with_gripper.launch sim:=true
```

Next, bring up Pi Robot's move_group.launch file if it is not already running:

```
$ roslaunch pi_robot_moveit_config move_group.launch
```

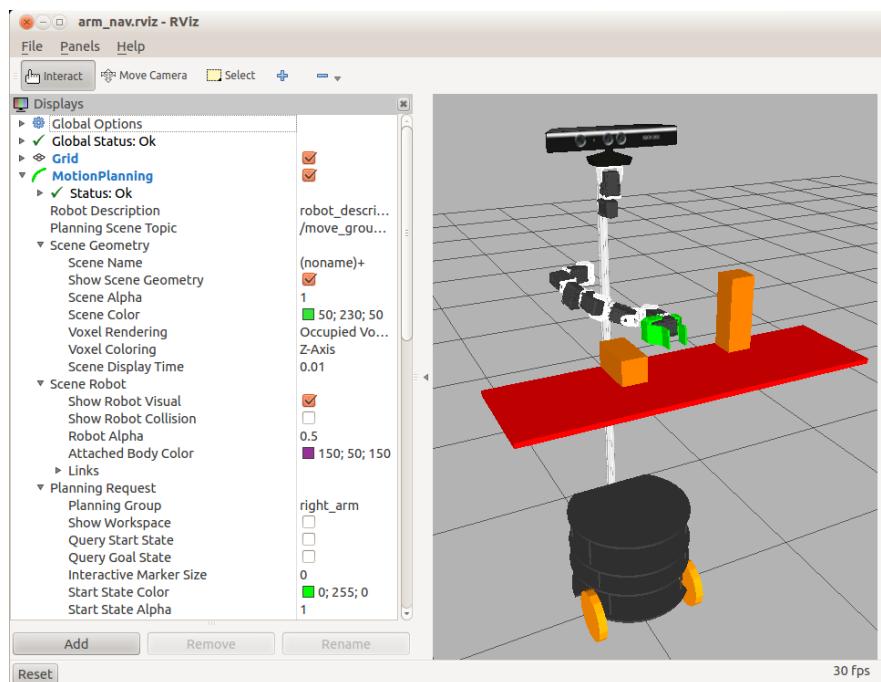
Make sure RViz is running with the arm_nav.rviz config file:

```
$ rosrun rviz rviz -d `rospack find rbx2_arm_nav`/config/arm_nav.rviz
```

Now run the moveit_obstacles.demo.py script with the following command:

```
$ rosrun rbx2_arm_nav moveit_obstacles_demo.py
```

The script first adds a floating table top and two boxes to the scene. It then moves the arm to the "resting" position in case it is not already there. Next we set a target pose for the gripper to place it in between the two boxes and a few centimeters above the table. MoveIt! then deftly controls the arm to move the gripper to the target pose while avoiding any collisions with any part of the arm with the obstacles. Finally, we move the arm back to the resting position. The view in RViz should something look like this part way through the sequence:



Let's now take a look at the code.

Link to source: [moveit_obstacles_demo.py](#)

```
1 #!/usr/bin/env python
2
3 import rospy, sys
4 import moveit_commander
5 from moveit_commander import MoveGroupCommander, PlanningSceneInterface
6 from moveit_msgs.msg import PlanningScene, ObjectColor
7 from geometry_msgs.msg import PoseStamped, Pose
8
9 class MoveItDemo:
10     def __init__(self):
11         # Initialize the move_group API
12         moveit_commander.roscpp_initialize(sys.argv)
13
14         rospy.init_node('moveit_demo')
15
16         # Construct the initial scene object
17         scene = PlanningSceneInterface()
18
19         # Create a scene publisher to push changes to the scene
20         self.scene_pub = rospy.Publisher('planning_scene', PlanningScene)
21
22         # Create a dictionary to hold object colors
23         self.colors = dict()
24
25         # Take a breath...
26         rospy.sleep(1)
27
28         # Initialize the MoveIt! commander for the right arm
29         right_arm = MoveGroupCommander('right_arm')
30
31         # Get the name of the end-effector link
32         end_effector_link = right_arm.get_end_effector_link()
33
34         # Allow some leeway in position (meters) and orientation (radians)
35         right_arm.set_goal_position_tolerance(0.01)
36         right_arm.set_goal_orientation_tolerance(0.05)
37
38         # Allow replanning to increase the odds of a solution
39         right_arm.allow_replanning(True)
40
41         # Set the reference frame for pose targets
42         reference_frame = 'base_footprint'
43
44         # Set the right arm reference frame accordingly
45         right_arm.set_pose_reference_frame(reference_frame)
46
47         # Allow 5 seconds per planning attempt
48         right_arm.set_planning_time(5)
49
50         # Give each of the scene objects a unique name
```

```

51     table_id = 'table'
52     box1_id = 'box1'
53     box2_id = 'box2'
54
55     # Remove leftover objects from a previous run
56     scene.remove_world_object(box1_id)
57     scene.remove_world_object(box2_id)
58     scene.remove_world_object(table_id)
59
60     # Give the scene a chance to catch up
61     rospy.sleep(1)
62
63     # Start the arm in the "resting" pose stored in the SRDF file
64     right_arm.set_named_target("resting")
65     right_arm.go()
66
67     rospy.sleep(2)
68
69     # Set the height of the table off the ground
70     table_ground = 0.75
71
72     # Set the length, width and height of the table and boxes
73     table_size = [0.2, 0.7, 0.01]
74     box1_size = [0.1, 0.05, 0.05]
75     box2_size = [0.05, 0.05, 0.15]
76
77     # Add a table top and two boxes to the scene
78     table_pose = PoseStamped()
79     table_pose.header.frame_id = reference_frame
80     table_pose.pose.position.x = 0.26
81     table_pose.pose.position.y = 0.0
82     table_pose.pose.position.z = table_ground + table_size[2] / 2.0
83     table_pose.pose.orientation.w = 1.0
84     scene.add_box(table_id, table_pose, table_size)
85
86     box1_pose = PoseStamped()
87     box1_pose.header.frame_id = reference_frame
88     box1_pose.pose.position.x = 0.21
89     box1_pose.pose.position.y = -0.1
90     box1_pose.pose.position.z = table_ground + table_size[2] +
box1_size[2] / 2.0
91     box1_pose.pose.orientation.w = 1.0
92     scene.add_box(box1_id, box1_pose, box1_size)
93
94     box2_pose = PoseStamped()
95     box2_pose.header.frame_id = reference_frame
96     box2_pose.pose.position.x = 0.19
97     box2_pose.pose.position.y = 0.15
98     box2_pose.pose.position.z = table_ground + table_size[2] +
box2_size[2] / 2.0
99     box2_pose.pose.orientation.w = 1.0
100    scene.add_box(box2_id, box2_pose, box2_size)
101
102    # Make the table red and the boxes orange

```

```

103     self.setColor(table_id, 0.8, 0, 0, 1.0)
104     self.setColor(box1_id, 0.8, 0.4, 0, 1.0)
105     self.setColor(box2_id, 0.8, 0.4, 0, 1.0)
106
107     # Send the colors to the planning scene
108     self.sendColors()
109
110     # Set the target pose in between the boxes and above the table
111     target_pose = PoseStamped()
112     target_pose.header.frame_id = reference_frame
113     target_pose.pose.position.x = 0.2
114     target_pose.pose.position.y = 0.0
115     target_pose.pose.position.z = table_pose.pose.position.z +
table_size[2] + 0.05
116     target_pose.pose.orientation.w = 1.0
117
118     # Set the target pose for the arm
119     right_arm.set_pose_target(target_pose, end_effector_link)
120
121     # Move the arm to the target pose (if possible)
122     right_arm.go()
123
124     # Pause for a moment...
125     rospy.sleep(2)
126     # Exit MoveIt! cleanly
127     moveit_commander.roscpp_shutdown()
128
129     # Exit the script
130     moveit_commander.os._exit(0)
131
132     # Set the color of an object
133     def setColor(self, name, r, g, b, a = 0.9):
134         # Initialize a MoveIt! color object
135         color = ObjectColor()
136
137         # Set the id to the name given as an argument
138         color.id = name
139
140         # Set the rgb and alpha values given as input
141         color.color.r = r
142         color.color.g = g
143         color.color.b = b
144         color.color.a = a
145
146         # Update the global color dictionary
147         self.colors[name] = color
148
149     # Actually send the colors to MoveIt!
150     def sendColors(self):
151         # Initialize a planning scene object
152         p = PlanningScene()
153
154         # Need to publish a planning scene diff
155         p.is_diff = True

```

```

156
157     # Append the colors from the global color dictionary
158     for color in self.colors.values():
159         p.object_colors.append(color)
160
161     # Publish the scene diff
162     self.scene_pub.publish(p)
163
164 if __name__ == "__main__":
165     try:
166         MoveItDemo()
167     except KeyboardInterrupt:
168         raise

```

Let's examine the key lines of the script:

```

5  from moveit_commander import MoveGroupCommander, PlanningSceneInterface
6  from moveit_msgs.msg import PlanningScene, ObjectColor

```

Here we import the `PlanningSceneInterface` that enables us to add and remove objects to and from the scene. We will also give the objects different colors by using the `ObjectColor` message type and we will publish object updates to the `planning_scene` topic using the `PlanningScene` message type.

```

17     scene = PlanningSceneInterface()

```

We create an instance of the `PlanningSceneInterface` class and assign it to the variable `scene`.

```

20     self.scene_pub = rospy.Publisher('planning_scene', PlanningScene)

```

Here we define the scene publisher that will be used to publish object color information back to the planning scene.

```

41     # Set the reference frame for pose targets
42     reference_frame = 'base_footprint'
43
44     # Set the right arm reference frame accordingly
45     right_arm.set_pose_reference_frame(reference_frame)

```

We will set the object poses relative to the `base_footprint` frame so we assign it to the variable `reference_frame` so that it can be reused throughout the script. We also set the arm's pose reference frame to the same frame to keep things simple. You can use different frames for scene objects and arm poses if it makes more sense for a given situation.

```
51     table_id = 'table'  
52     box1_id = 'box1'  
53     box2_id = 'box2'
```

Each scene object requires a unique name which we assign to the variables `table_id`, `box1_id` and `box2_id`.

```
56     scene.remove_world_object(box1_id)  
57     scene.remove_world_object(box2_id)  
58     scene.remove_world_object(table_id)
```

Since we might want to run the script more than once in succession, we use the `remove_world_object()` function to remove any scene objects that would have been added by a previous run.

```
69     # Set the height of the table off the ground  
70     table_ground = 0.75  
71  
72     # Set the length, width and height of the table and boxes  
73     table_size = [0.2, 0.7, 0.01]  
74     box1_size = [0.1, 0.05, 0.05]  
75     box2_size = [0.05, 0.05, 0.15]
```

Here we set the height of the floating table off the ground as well as the dimensions (in meters) of the table and the two boxes.

```
78     table_pose = PoseStamped()  
79     table_pose.header.frame_id = reference_frame  
80     table_pose.pose.position.x = 0.26  
81     table_pose.pose.position.y = 0.0  
82     table_pose.pose.position.z = table_ground + table_size[2] / 2.0  
83     table_pose.pose.orientation.w = 1.0
```

Next we set the pose for the table, placing it 0.26 meters in front of the reference frame (`/base_footprint`) and at a height determined by the `table_ground` variable (set to 0.75 meters earlier) plus half the height of the table itself.

```
84     scene.add_box(table_id, table_pose, table_size)
```

Now we can add the table to the planning scene using the `add_box()` function. This function takes a string argument for the name of the object followed by the pose and size of the box.

We will skip the lines where we set the poses of the two boxes and add them to the scene the same way we did for the table.

```

102     # Make the table red and the boxes orange
103     self.setColor(table_id, 0.8, 0, 0, 1.0)
104     self.setColor(box1_id, 0.8, 0.4, 0, 1.0)
105     self.setColor(box2_id, 0.8, 0.4, 0, 1.0)
106
107     # Send the colors to the planning scene
108     self.sendColors()

```

While certainly not necessary, it is nice to give the objects different colors so that they appear more distinct in RViz. Here we use a convenience function defined later in the script to color the table red and the two boxes orange.

```

111     target_pose = PoseStamped()
112     target_pose.header.frame_id = reference_frame
113     target_pose.pose.position.x = 0.2
114     target_pose.pose.position.y = 0.0
115     target_pose.pose.position.z = table_pose.pose.position.z +
table_size[2] + 0.05
116     target_pose.pose.orientation.w = 1.0

```

Here we set the target pose for the end effector to be in between the two boxes and 5 cm above the table. The target orientation is to have the gripper horizontal.

```

118     # Set the target pose for the arm
119     right_arm.set_pose_target(target_pose, end_effector_link)
120
121     # Move the arm to the target pose (if possible)
122     right_arm.go()

```

Finally, we set the target pose for the arm and run the `go()` command to plan and execute a trajectory that avoids the obstacles.

The script finishes up with a couple of auxiliary functions for setting the object colors. These functions were borrowed from Mike Ferguson's [moveit_python](#) package. The `setColor()` function is fairly self explanatory and uses the `ObjectColor` message type from the `moveit_msgs` package. However, the `sendColors()` function could use a little explanation so let's look at it now:

```

150     def sendColors(self):
151         # Initialize a planning scene object
152         p = PlanningScene()
153
154         # Need to publish a planning scene diff
155         p.is_diff = True
156
157         # Append the colors from the global color dictionary
158         for color in self.colors.values():

```

```
159         p.object_colors.append(color)
160
161     # Publish the scene diff
162     self.scene_pub.publish(p)
```

The key idea here is that we are making an update to the planning scene rather than creating one from scratch. The objects have already been added earlier in the script and we just want to paint them a different color. The `PlanningScene` message includes a field called `is_diff` and if this is set to `True` as we do above, then the overall planning scene will be updated rather than replaced by the information we publish. The last line in the function uses the scene publisher that we defined near the top of the script to actually publish the object colors.

11.25 Attaching Objects and Tools to the Robot

Suppose your robot is holding a tool or other object (maybe a light saber?) and we want motion planning to incorporate this object when performing collision checking. MoveIt! makes this easy by providing the `attach_box()` and `attach_mesh()` functions to attach the desired object to the robot—usually to the end-effector, but it can be attached to any part of the robot. Once the object is attached, any further motion commands will take into account the size and shape of the object.

The demo script `moveit_attached_object_demo.py` shows how to attach an elongated tool to Pi Robot's gripper and then move the arm to the "straight_forward" pose followed by the "resting" pose while avoiding contact between the tool and the floating table.

To test it out, make sure you have the fake version of Pi Robot running, as well as Pi's `move_group.launch` file. If `RViz` is already running, exit it now and bring it back up with the `attached_object.rviz` config file:

```
$ rosrun rviz rviz -d `rospack find \
rbx2_arm_nav`/config/attached_object.rviz
```

Now run the demo script:

```
$ rosrun rbt2_arm_nav moveit_attached_object_demo.py
```

The attached object will turn purple in color when the attachment is successful. It will then turn green again at the end when it is detached. Note how the movement of the arm compensates for the attached object and prevents it from striking any part of the table or the base of the robot.

NOTE: The rather jerky motion of the arm in this demo is due to some as yet unidentified bug or limitation in the update rate of the MoveIt! RViz plugin. A real arm would move smoothly.

The first part of the script is the same as the `moveit_obstacles_demo.py` so we won't repeat it here. The key lines for attaching the object to the end-effector are as follows:

```
60      # Set the length, width and height of the object to attach
61      tool_size = [0.3, 0.02, 0.02]
62
63      # Create a pose for the tool relative to the end-effector
64      p = PoseStamped()
65      p.header.frame_id = end_effector_link
66
67      # Place the end of the object within the grasp of the gripper
68      p.pose.position.x = tool_size[0] / 2.0 - 0.025
69      p.pose.position.y = 0.0
70      p.pose.position.z = 0.0
71
72      # Align the object with the gripper (straight out)
73      p.pose.orientation.x = 0
74      p.pose.orientation.y = 0
75      p.pose.orientation.z = 0
76      p.pose.orientation.w = 1
77
78      # Attach the tool to the end-effector
79      scene.attach_box(end_effector_link, 'tool', p, tool_size)
```

In Line 62 we set the dimensions of the box-shaped tool and give it a length of 0.3 meters (30 cm) and a width and height of 0.02 meters (2 cm).

Lines 64–76 define the pose we want the object to have relative to the end-effector. The pose used here orients the long dimension of the tool to be parallel to the gripper fingers and pointing out from the arm. To position the tool so that it appears to be grasped by the gripper at one end, we slide it $\frac{1}{2}$ the length of the tool so that its end is positioned just at the finger tips, then bring it back 2.5 cm so the gripper appears to have a good hold on it.

Finally, in Line 79, we use the `attach_box()` function on the `scene` object to attach our box-shaped tool to the end-effector. This function takes four arguments: the link to which we want to attach the object; the name (given as a string) assigned to the object we are attaching (so it can be referred to later); the pose defining the object's relation to the attachment link; the size of the box we are attaching given as a triple specifying length, width and height. You can also supply an optional fifth argument which is the

list of "touch links" this object is allowed to contact without it being considered a collision.

That's all there is to it. The rest of the script simply places the arm in the "straight_forward" pose followed by the "resting" pose. MoveIt! takes care of the motion planning with the object attached to the end-effector.

Instead of the `attach_box()` function used above, you can also use the `attach_mesh()` function to attach an STL or Collada mesh object stored in a file. The order of the arguments in this case is: `attach_mesh(link, object_name, pose, filename, touch_links=[])`.

Toward the end of the script, we detach the tool from the end-effector with the statement:

```
102     scene.remove_attached_object(end_effector_link, 'tool')
```

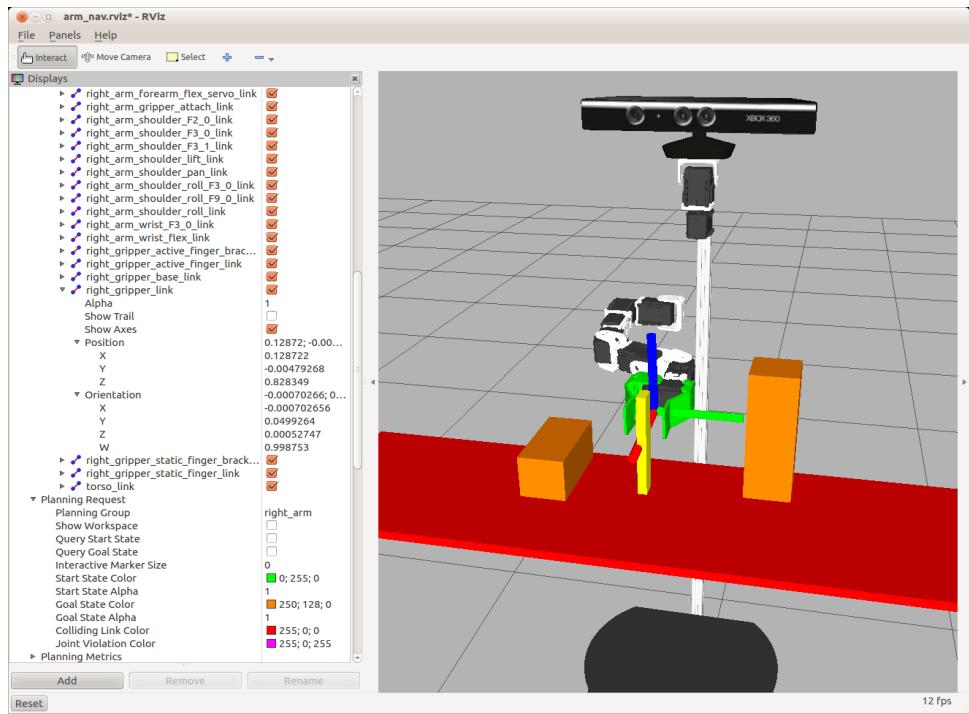
Note how we reference the tool by the name we gave it earlier during the attach operation. Note also that like the `attach_box()` function, the `remove_attached_object()` function is defined on the `scene` object, not the move group for the arm.

11.26 Pick and Place

The next demonstration adds another object to the table that we would like the robot to grasp with its gripper. The object will appear in RViz as a narrow yellow box positioned in between the two orange boxes. We already know from the earlier obstacle demo that we can move the gripper in between the two larger boxes if we send it a pre-calculated pose. The more realistic challenge is to derive the grasping pose from the pose of the target object. MoveIt! does not yet have a function to do this automatically for us. Fortunately, others have provided utilities to make our job easier.

The basic task at hand is to generate a collection of reasonable gripper poses that *might* do the job. Since we know the pose of the target object, we can try a series of gripper poses centered on the object and aligned with its orientation. These trial poses will be given a range of pitch, yaw and (perhaps) roll values until MoveIt! lets us know that a given pose can be reached while not colliding with the target or other obstacles.

The image below shows the orientation and position of the virtual frame attached to the `right_gripper_link`:



Remember that we defined this frame for planning purposes and placed the axes in between the two finger pads which is where we would typically grasp an object. From the image you can see that we want to align this frame with the target object (the narrow yellow block) and place the origin of the gripper frame near the center of the object.

Before looking at the code, let's try it out in simulation. If you're not already running Pi Robot in the ArbotiX simulator, bring it up now:

```
$ rosrun rbox2_bringup pi_robot_with_gripper.launch sim:=true
```

Next, bring up Pi Robot's `move_group.launch` file if it is not already running:

```
$ rosrun pi_robot_moveit_config move_group.launch
```

Now run RViz with the `pick_and_place.rviz` config file:

```
$ rosrun rviz rviz -d `rospack find \`rbox2_arm_nav`/config/pick_and_place.rviz`
```

Finally, run the `moveit_pick_and_place.demo.py` script:

```
$ rosrun rbx2_arm_nav moveit_pick_and_place_demo.py
```

Like the obstacles demo, the script first adds a floating table top and two boxes to the scene. It also adds a narrow box in between the first two boxes and this is set as the target for the pick-and-place operation. After generating a number of test grasps, the arm is moved to place the gripper in the correct position to grasp the target. The gripper then closes around the target and the target is placed at a new location on the table.

NOTE 1: You will notice that the movement of the grasped object seems to lag the movement of the gripper. This is an artifact of some as yet unidentified bug or limitation in the MoveIt! RViz plugin. You can see smooth motion of the object at the expense of jerky movement of the arm by changing the following settings in RViz:

- under **Scene Robot**, change the **Alpha** value from 0 to 1
- un-check the **RobotModel** display

Now run the pick-and-place demo again. This time the arm motion will not be as smooth but the grasped object will move in sync with the gripper.

NOTE 2: You might find that either the pick operation or the place operation fails after the five attempts allowed in the script. This can sometimes happen when the IK solver fails to find a solution for moving either the gripper (during pick) or the grasped object (during place). We'll discuss methods for increasing the chances of success below. For now, simply run the script again until both the pick and place operations succeed.

Since the `moveit_pick_and_place_demo.py` script is similar to the `moveit_obstacles_demo.py` script that we looked at in the previous section, we will focus only on what is new.

Link to source: [moveit_pick_and_place_demo.py](#)

Let's start with the new import statements:

```
from moveit_msgs.msg import Grasp, GripperTranslation, MoveItErrorCodes
```

First we need to import the `Grasp`, `GripperTranslation` and `MoveItErrorCodes` message types. The `Grasp` message type will be used to specify a list of possible grasp poses to try when the gripper is in range of the target object. The `GripperTranslation` message type allows us to specify an approach direction as well as minimum and desired approach distances. And the `MoveItErrorCodes` will allow us to test for the success of a pick or place operations. More details will follow below.

```
from tf.transformations import quaternion_from_euler
```

We also import the `quaternion_from_euler` function from the `tf` library since we will be converting some Euler angles to quaternions.

```
GROUP_NAME_ARM = 'right_arm'
GROUP_NAME_GRIPPER = 'right_gripper'

GRIPPER_FRAME = 'right_gripper_link'

GRIPPER_CLOSED = [-0.06]
GRIPPER_OPEN = [0.05]
GRIPPER_NEUTRAL = [0.0]

GRIPPER_JOINT_NAMES = ['right_gripper_finger_joint']

GRIPPER EFFORT = [1.0]

REFERENCE_FRAME = 'base_footprint'
```

For convenience, we define a number of global variables to hold the names of various links, frames and joints. These could also be read in as ROS parameters. The values listed for the three gripper positions (closed, open and neutral) were determined somewhat empirically.

```
self.gripper_pose_pub = rospy.Publisher('gripper_pose', PoseStamped)
```

During debugging, it is sometimes helpful to visualize the gripper poses that are attempted when trying to pick up the target object. Here we define a publisher that will allow us to publish the various poses on the "gripper_pose" topic. We can then subscribe to this topic in RViz to view the poses.

```
# Set a limit on the number of pick attempts before bailing
max_pick_attempts = 5

# Set a limit on the number of place attempts
max_place_attempts = 5
```

The MoveIt! pick-and-place operation is not guaranteed to succeed on the first try so we specify a maximum number of attempts for the pick operation and the place operation separately.

```
target_size = [0.02, 0.01, 0.12]
```

Like we did for the table and boxes, we define a length, width and height for the target object. In this case we are creating a narrow box target that is 12cm high and 2cm x 1cm on its sides.

```
# Set the target pose in between the boxes and on the table
target_pose = PoseStamped()
target_pose.header.frame_id = REFERENCE_FRAME
target_pose.pose.position.x = 0.22
target_pose.pose.position.y = 0.0
target_pose.pose.position.z = table_ground + table_size[2] + target_size[2] /
2.0
target_pose.pose.orientation.w = 1.0

# Add the target object to the scene
scene.add_box(target_id, target_pose, target_size)
```

Here we set the pose of the target so that it sits on the table in between the two boxes. We then add the target to the scene using the `add_box()` function.

```
# Set the support surface name to the table object
right_arm.set_support_surface_name(table_id)
```

With the objects added to the scene, we set the table as the support surface for pick and place operations. This allows MoveIt! to ignore a collision warning when placing the object back on the table.

```
# Specify a pose to place the target after being picked up
place_pose = PoseStamped()
place_pose.header.frame_id = REFERENCE_FRAME
place_pose.pose.position.x = 0.18
place_pose.pose.position.y = -0.18
place_pose.pose.position.z = table_ground + table_size[2] + target_size[2] /
2.0
place_pose.pose.orientation.w = 1.0
```

Next we specify the location where we want the target object to be placed after it is grasped by the robot. The pose defined above is located on the table to the right of the right box (as viewed by the robot). By setting the orientation to the unit quaternion, we are also indicating that the object should be placed in the upright position.

```
# Initialize the grasp pose to the target pose
grasp_pose = target_pose
```

The MoveIt! pick operation requires that we provide at least one possible grasp pose for the gripper that might work for holding the target object. In fact, we will provide a range of different grasp poses but we start with a pose that matches the target pose.

```
# Shift the grasp pose by half the width of the target to center it
```

```
grasp_pose.pose.position.y -= target_size[1] / 2.0
```

Here we tweak the initial grasp pose a little to the right of the target's center. The only reason we are doing this is that Pi Robot's gripper moves only the right finger while the left finger is fixed. By shifting the grasp pose a little toward the fixed finger, the closing of the right finger against the target will result in a more centered grasp. If you are using a parallel gripper with either two servos (one for each finger) or a prismatic joint that moves both fingers evenly with a single servo, then this adjustment would not be necessary.

```
# Generate a list of grasps
grasps = self.make_grasps(grasp_pose, [table_id])
```

As we mentioned earlier, we will generate a range of grasp poses for MoveIt! to try during the pick operation. Here we use a utility function called `make_grasps()` that we will describe in detail later. The function takes two arguments: the initial grasp pose to try and a list of object IDs that the gripper is allowed to touch on its final approach to the target. In this case, we allow it to touch the table. The `make_grasps()` function returns a list of possible grasp poses for the gripper.

```
for grasp in grasps:
    self.gripper_pose_pub.publish(grasp.grasp_pose)
    rospy.sleep(0.2)
```

As explained earlier, it is sometimes helpful to visualize the grasp poses in `RViz`. Here we publish each pose in rapid succession so we can quickly view the poses if desired.

```
# Track success/failure and number of attempts for pick operation
result = None
n_attempts = 0

# Repeat until we succeed or run out of attempts
while result != MoveItErrorCodes.SUCCESS and n_attempts < max_pick_attempts:
    result = right_arm.pick(target_id, grasps)
    n_attempts += 1
    rospy.loginfo("Pick attempt: " + str(n_attempts))
    rospy.sleep(0.2)
```

We have finally reached the block where we attempt to pick up the target object using our list of grasp poses. First we set a flag to test the `result` that will allow us to abort the loop if we find a successful grasp pose. We also track the number of times (`n_attempts`) that we run through the collection of grasps. Note that each attempt involves a run through the entire grasp list. Because of random perturbations used in the underlying pick and IK algorithms, it often pays to cycle through the list more than once if earlier attempts fail.

The actual pick operation is the line highlighted in **bold** above. The `pick()` function takes two arguments: the id of the object to grasp and the list of grasps to try. If one of the grasps is found to pass all tests for reachability and avoiding collisions, the `pick()` operation returns `SUCCESS` and allows us to break out of the loop. A successful `pick()` will move the arm into position while opening the gripper, then close the gripper on the target object. If we exceed the maximum number of attempts we set at the top of the script, we also exit the loop.

If the object is picked up successfully, we turn to the place operation as shown next.

```
# If the pick was successful, attempt the place operation
if result == MoveItErrorCodes.SUCCESS:
    result = None
    n_attempts = 0

    # Generate valid place poses
    places = self.make_places(place_pose)
```

First we reset the `result` flag and the number of attempts. We then use the utility function `make_places()` defined later in the script to generate a list of candidate poses to place the object. We give `make_places()` the desired `place_pose` we defined earlier in the script as an argument. The function then returns a list of poses similar to the target pose but that might vary slightly in position and/or orientation to give the IK solver a better chance of succeeding.

```
while result != MoveItErrorCodes.SUCCESS and n_attempts < max_place_attempts:
    for place in places:
        result = right_arm.place(target_id, place)
        if result == MoveItErrorCodes.SUCCESS:
            break
    n_attempts += 1
    rospty.loginfo("Place attempt: " + str(n_attempts))
    rospty.sleep(0.2)
```

Here we execute a loop similar to the one we used for `pick()`. The actual `place()` operation occurs on the line highlighted in bold above. Unlike the `pick()` function, `place()` takes a single pose rather than a list of poses as an argument. So we manually cycle through our list of places and run the `place()` operation on each pose. If the `place()` operation determines that the object can be successfully moved to the given pose, the arm will place the object at that pose and the gripper will release the object.

That essentially completes the script. Next we look at the all important `make_grasps()` utility function.

```
1     def make_grasps(self, initial_pose_stamped, allowed_touch_objects):
2         # Initialize the grasp object
```

```

3      g = Grasp()
4
5      # Set the pre-grasp and grasp postures appropriately
6      g.pre_grasp_posture = self.make_gripper_posture(GRIPPER_OPEN)
7      g.grasp_posture = self.make_gripper_posture(GRIPPER_CLOSED)
8
9      # Set the approach and retreat parameters as desired
10     g.pre_grasp_approach = self.make_gripper_translation(0.01, 0.1, [1.0,
11     0.0, 0.0])
12     g.post_grasp_retreat = self.make_gripper_translation(0.1, 0.15, [0.0,
13     -1.0, 1.0])
14
15     # Set the first grasp pose to the input pose
16     g.grasp_pose = initial_pose_stamped
17
18     # Pitch angles to try
19     pitch_vals = [0, 0.1, -0.1, 0.2, -0.2, 0.3, -0.3]
20
21     # Yaw angles to try
22     yaw_vals = [0]
23
24     # A list to hold the grasps
25     grasps = []
26
27     # Generate a grasp for each pitch and yaw angle
28     for y in yaw_vals:
29         for p in pitch_vals:
30             # Create a quaternion from the Euler angles
31             q = quaternion_from_euler(0, p, y)
32
33             # Set the grasp pose orientation accordingly
34             g.grasp_pose.pose.orientation.x = q[0]
35             g.grasp_pose.pose.orientation.y = q[1]
36             g.grasp_pose.pose.orientation.z = q[2]
37             g.grasp_pose.pose.orientation.w = q[3]
38
39             # Set and id for this grasp (simply needs to be unique)
40             g.id = str(len(grasps))
41
42             # Set the allowed touch objects to the input list
43             g.allowed_touch_objects = allowed_touch_objects
44
45             # Don't restrict contact force
46             g.max_contact_force = 0
47
48             # Degrade grasp quality for increasing pitch angles
49             g.grasp_quality = 1.0 - abs(p)
50
51             # Append the grasp to the list
52             grasps.append(deepcopy(g))
53
54     # Return the list
55     return grasps

```

The `make_grasps()` function was borrowed from Michael Ferguson's [chessbox](#) package. A MoveIt! Grasp message requires a pre-grasp posture for the gripper (e.g. open), a grasp posture (e.g. closed), a grasp approach vector (the direction the gripper should take to approach the target), and a grasp retreat vector (the direction to move the gripper once the object is grasped).

With these requirements in mind, let's now break down the `make_grasps()` function line by line.

```
3     g = Grasp()
```

First we initialize the variable `g` as a `Grasp` object.

```
6     g.pre_grasp_posture = self.make_gripper_posture(GRIPPER_OPEN)
7     g.grasp_posture = self.make_gripper_posture(GRIPPER_CLOSED)
```

Next we set the `pre_grasp_posture` for the gripper to be the open position and the `grasp_posture` to be the closed position. Note that we are using another utility function called `make_gripper_posture()` that is also defined in the script and which we will describe later.

```
10    g.pre_grasp_approach = self.make_gripper_translation(0.01, 0.1, [1.0,
11        0.0, 0.0])
11    g.post_grasp_retreat = self.make_gripper_translation(0.1, 0.15, [0.0,
-1.0, 1.0])
```

Here we define the approach and retreat vectors using the `make_gripper_translation()` utility function that will be described below. That function takes three parameters: a minimum distance, a desired distance and a direction vector specified as a list with `x`, `y` and `z` components.

Referring to Line 10 above, we want the gripper to approach along the `x`-direction, `[1.0, 0.0, 0.0]`, which is aligned with the forward direction of the base since the robot is standing behind the target. We want the minimum approach distance to be 1cm and the desired approach distance to be 10cm.

In a similar manner, Line 11 defines a retreat direction upward and to the right, `[0.0, -1.0, 1.0]`, since we want to lift the object over one of the boxes and move it to the right of that box. Here we also set a minimum retreat distance of 10cm and a desired distance of 15cm.

```
14    g.grasp_pose = initial_pose_stamped
```

Next we set the `grasp_pose` of the grasp object to the initial pose that was passed to the `make_grasps()` function as an argument. Recall that the pose we passed in was the same as the target pose.

```
16      # Pitch angles to try
17      pitch_vals = [0, 0.1, -0.1, 0.2, -0.2, 0.4, -0.4]
18
19      # Yaw angles to try
20      yaw_vals = [0]
```

Next we will generate a number of alternate grasp poses with various values for pitch and (optionally) yaw angles. The `pitch_vals` and `yaw_vals` lists above indicate the angles we will try (in radians). Note that in this case, we are only going to vary the pitch angle. You could also add a roll parameter if you find it necessary.

```
25      # Generate a grasp for each pitch and yaw angle
26      for y in yaw_vals:
27          for p in pitch_vals:
28              # Create a quaternion from the Euler angles
29              q = quaternion_from_euler(0, p, y)
```

For each value of yaw and pitch, we create a quaternion from the two Euler components using the `quaternion_from_euler` function that we imported at the top of the script.

```
32      g.grasp_pose.pose.orientation.x = q[0]
33      g.grasp_pose.pose.orientation.y = q[1]
34      g.grasp_pose.pose.orientation.z = q[2]
35      g.grasp_pose.pose.orientation.w = q[3]
```

We then set the grasp pose orientation components to those of the computed quaternion.

```
38      g.id = str(len(grasps))
```

Every grasp pose requires a unique id so we simply set it to the length of the list of grasps which grows by one as we add each grasp.

```
41      g.allowed_touch_objects = allowed_touch_objects
```

A grasp can be given a list of objects it is allowed to touch. In our case, we will allow the gripper to touch the table if needed on its final approach to the target. The `allowed_touch_objects` list is passed as an argument to the `make_grasps()` function and recall that we passed the list `[table_id]` to allow the gripper to touch the table.

```
44
```

```
g.max_contact_force = 0
```

If we were controlling a real arm, we would set a reasonable value for the maximum contact force but since we are working only in simulation, we set a value of 0.

```
47
```

```
g.grasp_quality = 1.0 - abs(p)
```

The MoveIt! pick operation expects the grasp pose candidates to be ranked by quality. Here we assume that our initial guess (the target pose) has the highest quality (1.0) which corresponds to a pitch value of 0. For other pitch values, we degrade the quality by subtracting the absolute value of the pitch from 1.0 which reflects the fact that we don't expect high values of pitch to be necessary.

```
50
```

```
grasps.append(deepcopy(g))
```

Finally, we append the grasp to the list of grasps. We use the Python `deepcopy()` function to ensure that we add a truly distinct copy of the grasp object `g`. Otherwise, subcomponents of `g` might only contain references to a previous version of `g` as we cycle through our loop.

```
53
```

```
return grasps
```

When we have finished generating the list of grasp poses, the list is returned to the calling program.

The `make_grasps()` function makes use of two additional utility functions, `make_gripper_posture()` and `make_gripper_translation()`. Let's look at these briefly now.

```
1  def make_gripper_posture(self, joint_positions):
2      # Initialize the joint trajectory for the gripper joints
3      t = JointTrajectory()
4
5      # Set the joint names to the gripper joint names
6      t.joint_names = GRIPPER_JOINT_NAMES
7
8      # Initialize a joint trajectory point to represent the goal
9      tp = JointTrajectoryPoint()
10
11     # Assign the trajectory joint positions to the input positions
12     tp.positions = joint_positions
13
14     # Set the gripper effort
15     tp.effort = GRIPPER_EFFORT
16
```

```

17     # Append the goal point to the trajectory points
18     t.points.append(tp)
19
20     # Return the joint trajectory
21     return t

```

The `make_gripper_posture()` function is fairly self-explanatory and takes a goal posture for the gripper given as a list of joint positions and returns a joint trajectory that will move the fingers to the desired positions. In our case, we use the function to turn the `OPEN` and `CLOSED` postures for the gripper into opening and closing trajectories to use when approaching the target object and grasping it once it reached it.

```

1  def make_gripper_translation(self, min_dist, desired, vector):
2      # Initialize the gripper translation object
3      g = GripperTranslation()
4
5      # Set the direction vector components to the input
6      g.direction.vector.x = vector[0]
7      g.direction.vector.y = vector[1]
8      g.direction.vector.z = vector[2]
9
10     # The vector is relative to the gripper frame
11     g.direction.header.frame_id = GRIPPER_FRAME
12
13     # Assign the min and desired distances from the input
14     g.min_distance = min_dist
15     g.desired_distance = desired
16
17     return g

```

The `make_gripper_translation()` function is also fairly self-explanatory. Recall that we used this function to define the approach and retreat vectors for the gripper when reaching for the object and then moving it to its new location on the table.

Finally, we have the `make_places()` function that was used to generate a number of alternative poses to place the grasped object on the table.

```

1  def make_places(self, init_pose):
2      # Initialize the place location as a PoseStamped message
3      place = PoseStamped()
4
5      # Start with the input place pose
6      place = init_pose
7
8      # A list of x shifts (meters) to try
9      x_vals = [0, 0.005, 0.01, 0.015, -0.005, -0.01, -0.015]
10
11     # A list of y shifts (meters) to try
12     y_vals = [0, 0.005, 0.01, 0.015, -0.005, -0.01, -0.015]
13

```

```

14     # A list of pitch angles to try
15
16     pitch_vals = [0]
17
18     # A list of yaw angles to try
19     yaw_vals = [0]
20
21     # A list to hold the places
22     places = []
23
24     # Generate a place pose for each angle and translation
25     for y in yaw_vals:
26         for p in pitch_vals:
27             for y in y_vals:
28                 for x in x_vals:
29                     place.pose.position.x = init_pose.pose.position.x + x
30                     place.pose.position.y = init_pose.pose.position.y + y
31
32                     # Create a quaternion from the Euler angles
33                     q = quaternion_from_euler(0, p, y)
34
35                     # Set the place pose orientation accordingly
36                     place.pose.orientation.x = q[0]
37                     place.pose.orientation.y = q[1]
38                     place.pose.orientation.z = q[2]
39                     place.pose.orientation.w = q[3]
40
41                     # Append this place pose to the list
42                     places.append(deepcopy(place))
43
44     # Return the list
45     return places

```

As you can see, this function is very similar to the `make_grasps()` utility that was described in detail above. Instead of returning a list of grasp poses that vary in orientation, we are returning a list of place poses that vary in position around the desired pose.

11.27 Adding a Sensor Controller

So far we have been adding simulated objects to the MoveIt! planning scene to test collision avoidance. MoveIt! can also use point cloud data from a depth camera to add real obstacle information directly into the scene. Note that this is not the same as detecting specific objects for grasping which we will cover later in the chapter and again in the following chapter. MoveIt! simply builds an occupancy grid from depth data which allows motion planning to avoid hitting obstacles.

MoveIt! uses an [octree](#) representation of its 3D environment. As its name implies, each node in an octree has eight child nodes which can be used to represent the eight octants around a point in space and provides an efficient way to represent a spatial occupancy

map. MoveIt! currently knows how to use point clouds and depth maps to create an octree representation of the planning scene. The setup is quite easy as we now show for point clouds.

First we add a file called `sensors_rgbd.yaml` to the `config` subdirectory of our robot's MoveIt! configuration package that contains the following lines:

```
sensors:  
  - sensor_plugin: occupancy_map_monitor/PointCloudOctomapUpdater  
    point_cloud_topic: /camera/depth_registered/points  
    max_range: 5.0  
    frame_subsample: 1  
    point_subsample: 1  
    padding_offset: 0.1  
    padding_scale: 1.0  
    filtered_cloud_topic: filtered_cloud
```

This file defines our collection of sensors through the use of MoveIt! sensor plugins. In this case, we are defining a single plugin that uses the `PointCloudOctomapUpdater` to generate an octree occupancy map. The plugin expects to subscribe to the topic named by the `point_cloud_topic` parameter which in this case is `/camera/depth_registered/points`. We then specify parameters for the maximum depth range we want to process in meters, sub-sampling rates for the video stream (`frame_subsample`) and point cloud grid (`point_subsample`). The `padding_offset` (in centimeters) and `padding_scale` determine how much of a buffer around the robot itself we want to use when creating the self-filter. The resulting point cloud with the robot parts removed is published on the `filtered_cloud_topic`.

In addition to the `sensors_rgbd.yaml` configuration file, we also need a launch file to load the parameters. For Pi Robot, this file is called `pi_robot_moveit_sensor_manager.launch.xml` and is located in the `launch` subdirectory of Pi's MoveIt! package. The contents of the launch file are as follows:

```
<launch>  
  <param name="octomap_frame" type="string" value="odom" />  
  <param name="octomap_resolution" type="double" value="0.05" />  
  <param name="max_range" type="double" value="1.5" />  
  <rosparam command="load" file="$(find  
pi_robot_moveit_config)/config/sensors_rgbd.yaml" />  
</launch>
```

where the parameter `octomap_frame` specifies the coordinate frame in which this representation will be stored. If you are working with a mobile robot, this frame should be a fixed frame in the world. The `octomap_resolution` parameter specifies the resolution at which this representation is maintained (in meters).

To test things out, first fire up the OpenNI camera driver for your Kinect or Xtion Pro:

For the Microsoft Kinect:

```
$ rosrun freenect_launch freenect.launch
```

For the Asus Xtion, Xtion Pro, or Primesense 1.08/1.09 cameras:

```
$ rosrun openni2_launch openni2.launch
```

Next bring up Pi Robot in the ArbotiX simulator if it is not already running:

```
$ rosrun rbx2_bringup pi_robot_with_gripper.launch sim:=true
```

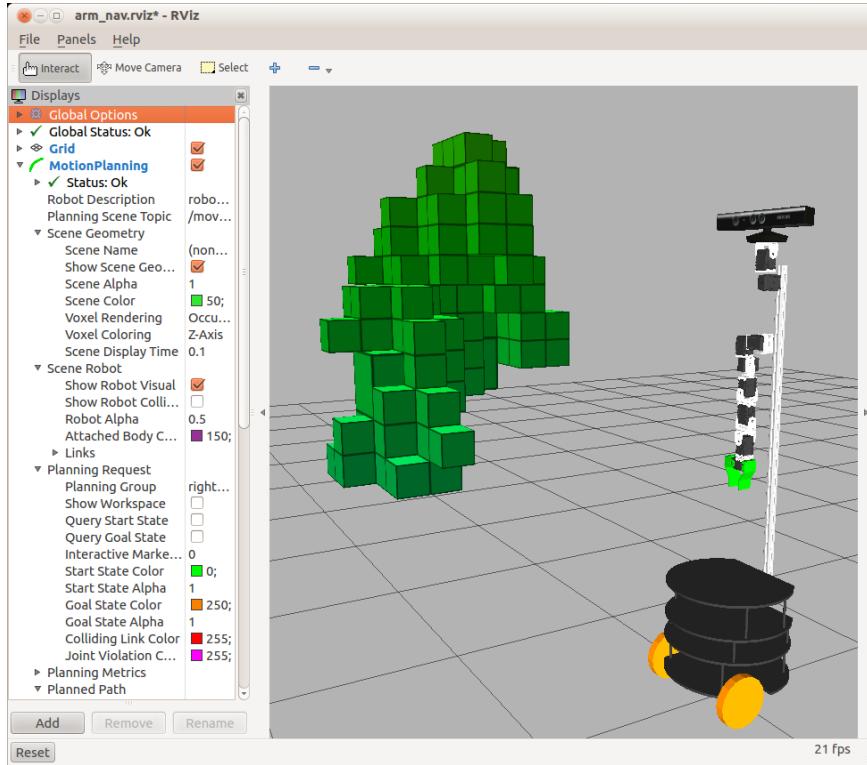
Now bring up the MoveIt! environment for Pi Robot which includes the sensor manager created above:

```
$ rosrun pi_robot_moveit_config move_group.launch
```

Finally, bring up RViz with the arm_nav.rviz config file:

```
$ rosrun rviz rviz -d `rospack find rbx2_arm_nav`/config/arm_nav.rviz
```

Assuming there are objects within view of the camera, the image in RViz should look something like the following:



If you don't see the boxes representing the octomap occupancy grid, check in **RViz** under the **Motion Planning** display, then open the **Scene Geometry** settings and ensure that the checkbox labeled **Show Scene Geometry** is checked. If you move objects in front of the camera, the octomap boxes should update accordingly.

If you were using a real robot, MoveIt! would incorporate this occupancy data when computing arm trajectories so as to avoid collisions between the arm and the detected objects.

NOTE: Unless you are explicitly using the octomap data with your robot, it is best to turn it off since it consumes a fair bit of CPU power. To disable the sensor processing, edit the sensor launch file we created above (for Pi Robot this is the `pi_robot_moveit_sensor_manager.launch.xml` file) and comment out all the lines like this:

```

<launch>
<!--
  <param name="octomap_frame" type="string" value="odom" />
  <param name="octomap_resolution" type="double" value="0.05" />
  <param name="max_range" type="double" value="1.5" />
  <rosparam command="load" file="$(find
pi_robot_moveit_config)/config/sensors_rgbd.yaml" />
-->
</launch>

```

Where the beginning and ending comment characters are shown in bold. Next, terminate your `move_group.launch` file if it is running, then delete any existing sensor parameters before re-running your `move_group.launch` file:

```

$ rosparam delete /move_group/sensors
$ roslaunch pi_robot_moveit_config move_group.launch

```

You only have to do this once since `move_group.launch` will now no longer set the sensor parameters thereby disabling octomap processing.

11.28 Running MoveIt! on a Real Arm

One of MoveIt!'s strengths is that we can run any of the scripts we have already developed on a real arm instead of the fake robot with essentially no modifications. Of course, this assumes that the real robot shares the same URDF model as the fake robot we have been using. If you have your own robot arm, the setup steps would be the same as we have done for Pi Robot:

1. Run the MoveIt! Setup Assistant for your URDF model to generate a MoveIt! configuration package for your robot.
2. Test your scripts using the MoveIt! demo mode and fake controllers.
3. For a more realistic test, run your scripts with a simulated version of your robot using the ArbotiX simulator.
4. Create or test the launch and configuration files for your servo controller. In this book we have shown how to use the [arbotix](#) package to control Dynamixel servos.
5. Create the required MoveIt! configuration files and launch files to match the trajectory controller used with your real servos,. We saw how to do this for the ArbotiX controllers in the section *Configuring MoveIt! Joint Controllers*.

Once your setup files are ready, the launch sequence is as follows:

1. Run the launch file(s) for your robot.
2. Run the `move_group.launch` file from your robot's MoveIt! package.
3. (Optional) Bring up `RViz` to monitor your robot's movements on the screen.
4. Run your own MoveIt! nodes or scripts to control the robot.

Before trying some of our demo scripts on the real Pi Robot, let's look at how you might modify these files for your own robot.

11.28.1 Creating your own launch files and scripts

Since the details of your robot's launch file(s) as well as the URDF model for your robot will not be the same as Pi Robot's, you will need to create and run your own files before you can test MoveIt! on a real robot. The best way to do this is to create your own package(s) for use with your robot. You can then copy over some of the files from the `rbx2` repository to use as templates if desired.

11.28.2 Running the robot's launch files

Before running the various test scripts in the following sections, we'll assume you have run the startup file(s) for your robot's controllers as well as the appropriate `move_group.launch` file from your robot's MoveIt! package. For Pi Robot, first we launch the startup file with the `sim` parameter set to `false`:

```
$ rosrun rbx2_bringup pi_robot_with_gripper.launch sim:=false
```

Then we run the `move_group.launch` file as before:

```
$ rosrun pi_robot_moveit_config move_group.launch
```

To protect the servos from overheating, we will also run the `monitor_dynamixels` node we created in Chapter 6:

```
$ rosrun rbx2_diagnostics monitor_dynamixels.launch
```

These launch files can stay running indefinitely as long as you are connected to your robot.

While not necessary when using the real robot, you can also bring up `RViz`:

```
$ rosrun rviz rviz -d `rospack find rbx2_arm_nav`/config/arm_nav.rviz
```

Or use your own choice of configuration file.

11.28.3 Forward kinematics on a real arm

Let's begin with the `moveit_fk_demo.py` we ran earlier. Note that you will probably have to change the target pose as defined by the `joint_positions` array unless your robot's arm is identical to Pi Robot's. You will probably also have to change the named poses (e.g. "resting" and "straight_forward") or comment out those lines.

As with any new script, it's best to test it first with a simulated version of your robot, either using the `MoveIt!_demo.launch` file for your robot or the ArbotiX simulator with a configuration file customized for your robot.

When you are ready, you can run the script the same way with the real robot. In Pi Robot's case we would run the `moveit_fk_demo.py` script exactly the same way we did with the simulated robot:

```
$ rosrun rbx2_arm_nav moveit_fk_demo.py
```

Hopefully your robot's arm will move in the desired way. When the script has finished, it is a good idea to relax all the servos to let them cool down. If you are using Dynamixel servos and the `arbotix` package, you can use the following command:

```
$ rosrun rbx2_dynamixels arbotix_relax_all_servos.py
```

11.28.4 Inverse kinematics on a real arm

Before running your copy of the `moveit_ik_demo.py` script, be sure to pick a target pose for the end effector that is reachable by your robot's gripper. Recall that in the original script, the target pose is defined by the block:

```
target_pose = PoseStamped()
target_pose.header.frame_id = reference_frame
target_pose.header.stamp = rospy.Time.now()
target_pose.pose.position.x = 0.20
target_pose.pose.position.y = -0.1
target_pose.pose.position.z = 0.85
target_pose.pose.orientation.x = 0.0
target_pose.pose.orientation.y = 0.0
target_pose.pose.orientation.z = 0.0
target_pose.pose.orientation.w = 1.0
```

This pose is 85cm above the `base_footprint`, 20cm ahead of center and 10cm to the right of center. These numbers will have to be adjusted to create a pose that is attainable by your robot's arm and gripper. One way to do this is as follows:

- make sure the arm's servos are in the relaxed state. If you are using the ArbotiX package, you can run the `arbotix_relax_all_servos.py` script.
- position the gripper manually at the desired target position and orientation
- run the `get_arm_pose.py` script we introduced earlier

```
$ rosrun rbt2_arm_nav
```

You can now release the arm and read off the desired position and orientation components from the screen. Copy and paste these values into your script for the target pose.

As in the previous section, it's best to test things out with a simulated version of your robot, either using the MoveIt! `demo.launch` file for your robot or the ArbotiX simulator with a configuration file customized for your robot.

When you are ready, you can run the script the same way with the real robot. In Pi Robot's case we would run:

```
$ rosrun rbt2_arm_nav moveit_ik_demo.py
```

When the arm has gone through its paces, relax the servos:

```
$ rosrun rbt2_dynamixels arbotix_relax_all_servos.py
```

11.28.5 Cartesian paths on a real arm

The `moveit_cartesian_demo.py` script will also require some tweaking to match the particulars of your robot and MoveIt! configuration file. In particular, the included script references the two saved poses "straight_forward" and "resting" which you might not have defined in your MoveIt! configuration. You might also have to modify the waypoint values so that the target poses are within reach of your robot's arm.

In the case of Pi Robot, we can run exactly the same script for the real robot as we did for the fake robot:

```
$ rosrun rbt2_arm_nav moveit_cartesian_demo.py
```

11.28.6 Pick-and-place on a real arm

In the simulated `moveit_pick_and_place.py` demo, we know the exact pose of the fake target object since we placed it in the scene programmatically. Similarly, we know the pose of the virtual table and obstacles. In a real pick-and-place task, the robot would

use its 3D vision to identify the shape and pose of the target in the real world. It would also identify the table and other objects as obstacles to be avoided when moving the arm.

The segmentation of the visual scene into discrete objects and support surfaces is often called "object detection" or "the perception pipeline". The term "object *recognition*" is reserved for cases where we actually match particular objects to stored models such as a mug, a bowl or a stapler. The programming details underlying these perception processes are outside the scope of this volume. However, we can still use ROS packages created by others to give our robot the basic perceptual abilities required to execute a real pick-and-place task.

We will run through a full pick-and-place example in the next chapter on Gazebo after learning how to use the object detection pipeline created for the [UBR-1 robot](#). As we will see, the UBR-1 perception pipeline can be run as a standalone process so that we can use it with our own robot to detect the pose of a target object sitting on a table top. The target pose will then be used more-or-less directly with our existing pick-and-place code to control the robot's arm to grasp the object and move it to another location on the table.

11.28.7 Pointing at or reaching for a visual target

We can also try the `arm_tracking.py` script we introduced earlier in the chapter. Instead of the fake target that we used in the simulation, we will use the `nearest_cloud.py` node introduced earlier so that the arm will tend to point at or reach for the object nearest to the camera.

Assuming you still have your launch files and MoveIt! nodes running for your robot, bring up the OpenNI node for the depth camera:

For the Microsoft Kinect:

```
$ rosrun freenect freenect.launch
```

For the Asus Xtion, Xtion Pro, or Primesense 1.08/1.09 cameras:

```
$ rosrun openni2 openni2.launch
```

Next run the `nearest_cloud.py` node:

```
$ rosrun rbx2_vision nearest_cloud.launch
```

Before starting the arm tracker, we need to run the head tracker node so that the robot will keep the depth camera pointed at the nearest object. Notice how we set the `sim` argument to `false` in the command below:

```
$ roslaunch rbx2_dynamixels head_tracker.launch sim:=false
```

Finally, start up the `arm_tracker.py` node:

```
$ rosrun rbx2_arm_nav arm_tracker.py
```

If all goes well, the robot should periodically update the position of his arm so that the gripper more or less reaches toward the nearest object positioned in front of the camera (but at least 0.5 meters away).

For extra bonus points, it is left to the reader to improve the `arm_tracker.py` script by setting the gripper orientation so that it actually points directly at the target rather than always been set to a horizontal orientation as it is now. For example, if the target is above the level of the shoulder, not only will the arm reach upward as it does now, but the gripper will also angle upward toward the target.

12. GAZEBO: SIMULATING WORLDS AND ROBOTS

Up until now we have been using the ArbotiX fake simulator to test our ROS nodes before trying things out on a real robot. The fake simulator is fast and easy to use but it has its limitations. For one thing, it does not simulate the actual physics (e.g. inertia, force, damping, friction, etc), nor does it provide a way to simulate objects in the world. Sometimes we would also like to simulate the data returned from sensors such as a laser scanner or depth camera as the robot moves around a simulated environment.

Enter [Gazebo](#), the sophisticated open source 3D simulator originally developed by Nate Koenig and Andrew Howard in 2002 as part of the [Player](#) project founded by Brian Gerkey, Richard Vaughan and Andrew Howard at the University of Southern California at Los Angeles. In 2011, Gazebo became an independent project under Willow Garage where it played an integral role in the development of ROS and the PR2 robot. Gazebo is now actively developed by the [Open Source Robotics Foundation](#) (OSRF) where Gerkey and Koenig serve as CEO and CTO respectively.

Gazebo not only provides a state-of-the-art physics engine, it also enables the creation of complex 3-dimensional virtual environments (called "worlds") for a simulated robot to play around in. If someone has created a fully configured Gazebo model of the robot we are interested in, then we can test a number of real-world properties such as the robot's momentum and inertia when trying to stop or accelerate, or the friction between a gripper and an object being grasped. We can also test bump sensors when running into things, cliff sensors when nearing an edge, or even perform a full blown SLAM experiment with a simulated laser to create a map of the robot's simulated environment.

Gazebo can also be used to prototype a new robot before it is even built, or learn how to operate a virtual copy of an expensive robot that only few can use in person such as the Willow Garage PR2, [Baxter from Rethink Robotics](#), the [Husky from Clearpath Robotics](#), the [NASA-GM Robonaut 2](#), or the [UBR-1](#).

In this Volume, we will be learning only the basics of how to use Gazebo as an end-user rather than a developer. Therefore, before we can use Gazebo with a particular robot, someone needs to do the work of creating a [Gazebo-compatible URDF](#) file for the robot as well as the harder task of configuring or programming [Gazebo control plugins](#) for the various sensors and actuators used by the robot. Although this process is not covered in this Volume, you can learn about the details online using the newly written [Gazebo Tutorials](#). Fortunately for us, we can use two ready made Gazebo-ized robots: the [Kobuki](#) (a.k.a TurtleBot 2) from [Yujin Robot](#) and the [UBR-1](#). After walking through the

installation of the Gazebo simulator as well as the models for the Kobuki and UBR-1, we will try out a few simulations.

12.1 Installing Gazebo

As of this writing, the latest version of Gazebo is 5.0 but we will be using the version that is installed with ROS Indigo which is 2.2. If you installed ROS Indigo using the `ros-indigo-desktop-full` meta-package, then you should already have Gazebo 2.2. However, to be certain, run the following commands to install `gazebo2`:

```
$ sudo apt-get install gazebo2
```

For more installation information, including source installs, see the [online tutorial](#).

After completing the installation, test the basic functionality by starting the Gazebo server:

```
$ gzserver
```

You should see some startup messages that look something like this:

```
Gazebo multi-robot simulator, version 2.2.3
Copyright (C) 2012-2014 Open Source Robotics Foundation.
Released under the Apache 2 License.
http://gazebosim.org
```

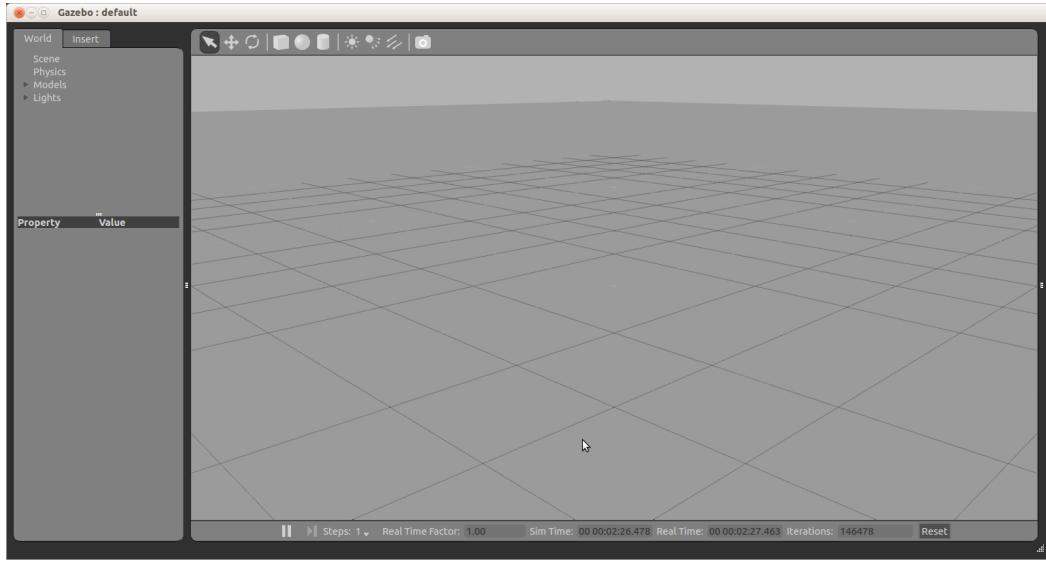
```
Msg Waiting for master
Msg Connected to gazebo master @ http://127.0.0.1:11345
Msg Publicized address: 10.0.0.3
```

The server will then continue to run in the foreground.

Open another terminal window and run the Gazebo client to bring up the GUI:

```
$ gzclient
```

If all goes well, the Gazebo GUI should appear which looks like the following:



As you can see above, the default world is empty. Once we install some robot models, we can try out a few simulations.

NOTE: If the GUI crashes and the window above does not appear, see the next section on the use of hardware acceleration.

12.2 Hardware Graphics Acceleration

Gazebo requires a fair bit of horsepower to run the physics and graphics engines but it is designed to make use of GPU hardware acceleration (if it is available on your computer) to reduce the load on the CPU. However, on some machines, hardware acceleration can cause the Gazebo GUI to crash, similar to what we sometimes find with `RViz`.

If you turned off hardware acceleration earlier in the book because of problems with `RViz` crashing, it is a good idea to try turning it on again when running Gazebo just in case it works. To make sure hardware acceleration is on, run the following command:

```
$ unset LIBGL_ALWAYS_SOFTWARE
```

Terminate any `gzserver` or `gzclient` processes you might have already running, then bring up the server and client together with the command;

```
$ gazebo
```

If the Gazebo GUI appears without trouble, then you are good to go. If it starts to come up and then crashes, try running the `gazebo` command again two or three times. (For some reason, the GUI will sometimes fail to come up on the first attempt but then run fine on a subsequent launch.) If all such attempts fail, terminate any running ROS processes including `roscore`. Then restart `roscore` and try the `gazebo` command a few more times.

If all attempts fail, try turning off hardware acceleration:

```
$ export LIBGL_ALWAYS_SOFTWARE=1
```

This will almost always solve the problem although at the cost of consuming more CPU cycles. If this works for you, place the above command at the end of your `~/.bashrc` file (if you haven't already) so it will be run each time you open a new terminal.

12.3 Installing the ROS Gazebo Packages

Gazebo is a standalone simulator that can be used independently of ROS but it also plays very nicely with ROS topics and services. To make the two work together, we only need to install a few Gazebo-related ROS packages. However, first make sure you remove any older Gazebo packages from previous ROS versions you might have on your machine:

```
$ sudo apt-get remove ros-fuerte-gazebo*
$ sudo apt-get remove ros-groovy-gazebo*
$ sudo apt-get remove ros-hydro-gazebo*
```

Now install the new packages for ROS Indigo:

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

To test your Gazebo ROS installation, run the `empty_world.launch` file from the `gazebo_ros` package:

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

If successful, you should eventually see the Gazebo GUI with an empty world. If the launch process crashes on the first attempt, type `Ctrl-C` and try again two or three times. If it keeps on crashing, see the previous section on turning off hardware acceleration.

For more information on installing ROS support for Gazebo including source installs, see the [online tutorial](#).

12.4 Installing the Kobuki ROS Packages

Even if you do not own a Kobuki, you can still install the Kobuki ROS packages and test the robot in Gazebo. The following command will install everything we need for the Kobuki robot under ROS Indigo:

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

NOTE: The Kobuki Gazebo package provides a model of the Kobuki's base only without a Kinect depth camera.

12.5 Installing the UBR-1 Files

Both a Gazebo model and MoveIt! configuration files are available for the UBR-1. To install everything we need you'll need to install two supporting Debian packages and then clone the `ubr1_preview` repository and build the source. Here are the required steps:

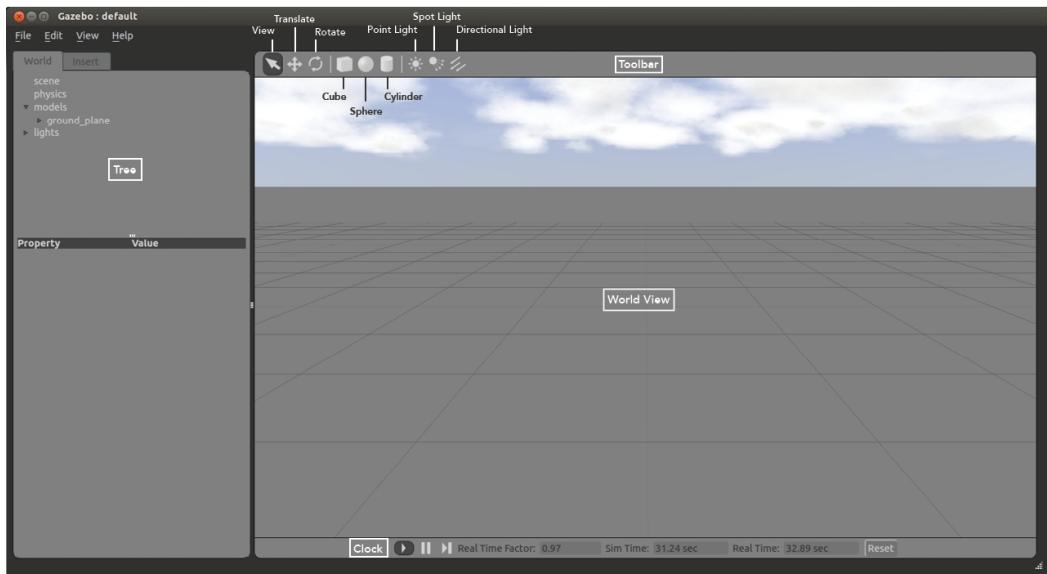
```
$ sudo apt-get install ros-indigo-grasping-msgs ros-indigo-moveit-python
$ cd ~/catkin_ws/src
$ git clone https://github.com/pirobot/ubr1_preview
$ cd ~/catkin_ws
$ catkin_make
$ rospack profile
```

If all goes well, the last build message you should see on the screen is:

```
[100%] Built target ubr1_gazebo_controllers
```

12.6 Using the Gazebo GUI

NOTE: The online Gazebo User Guide referred to in the Hydro version of this book seems to have disappeared from the Gazebo web site. Instead, the Gazebo website provides a [list of tutorials](#) for building robots and worlds, writing plugins and so on. Unfortunately, the screen shots illustrating the basic GUI controls seems to have gone missing so we will have to be content with the basic summary shown below.



One of the more frequently used menu items is the **Reset Model Poses** command found under the **Edit** menu. This will return the robot and any objects to the poses they had when the simulation was first started. The **Reset World** menu item does the same thing but also resets the clock.

NOTE: When using RViz, we use the **left** mouse button to move the camera in orbit mode. However, Gazebo uses the **middle** mouse button for this same function. On many mice, the middle mouse button is also a scroll wheel so it can sometimes be a little tricky to move the camera in this way. For a laptop with only two buttons, press both buttons together to emulate the middle mouse button.

Another important feature is the ability to move objects around within the world. The Gazebo web site used to include a handy chart listing mouse and key combinations for translating and rotating objects along different axes. We have included a copy of that chart below. For example, to lift an object upward (along the z-axis), first click on the translate control on the toolbar, then hold down the z key on your keyboard while using the left mouse button to move the object up or down.

View Mode

Select the single arrow in the toolbar. This Mode is used to move the camera to view the environment or models from different angles. When in this mode, you cannot move the models.

Translate	Left-press + drag
Orbit	Middle-press + drag
Zoom	Scroll wheel
Accelerated Zoom	Alt + Scroll wheel
Jump to object	Double-click object
Select object	Left-click object

Translate Mode

Select the four-headed arrow in the toolbar. To translate, click and drag a model to the desired location. For more precise translation, hold down the x, y, or z key while dragging to move only along that axis.

Translate	Left-press + drag
Translate (x-axis)	Left-press + X + drag
Translate (y-axis)	Left-press + Y + drag
Translate (z-axis)	Left-press + Z + drag

(Orbit & Zoom work in this mode, as well)

Rotate Mode

Select the two circular arrows in the toolbar. Rotate Mode allows you to rotate about each axis. Similar to Translate Mode, the x, y, and z keys are used to rotate about each axis.

Rotate (spin) object	Left-press + drag
Rotate (x-axis)	Left-press + X + drag
Rotate (y-axis)	Left-press + Y + drag
Rotate (z-axis)	Left-press + Z + drag

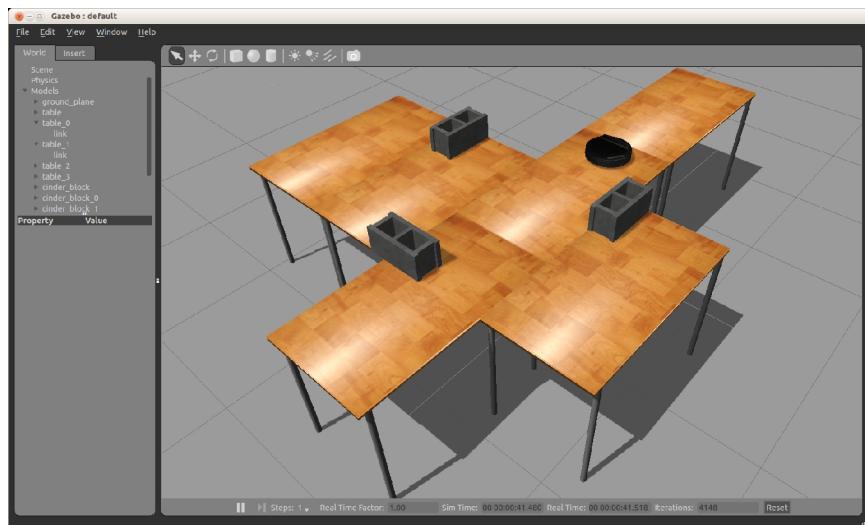
(Orbit & Zoom work in this mode, as well)

12.7 Testing the Kobuki Robot in Gazebo

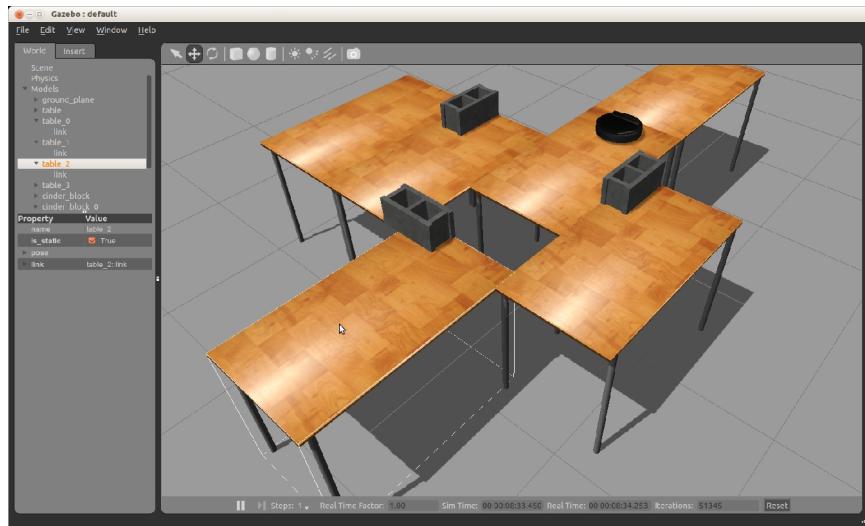
The Kobuki Gazebo package comes with a world that includes a collection of tables with the robot and a number of cinder blocks sitting on top. To bring up Gazebo with the Kobuki and this world terminate any running Gazebo instances and run the following launch file:

```
$ roslaunch kobuki_gazebo kobuki_playground.launch
```

When the robot and world have been loaded, the Gazebo screen should look like this:



As you can see, the world consists of a number of tables with the Kobuki and three cinder blocks sitting on top of them. Click on the translate or rotate control on the toolbar and then use your mouse (left click) to move a table, block or the robot. Use the middle mouse button to rotate the camera view. The image below shows the scene after one of the tables has been moved to create a gap:



We can now try to control the robot using standard ROS commands. But first we need to know the topics and services used by the simulated robot. Get the list of current topics by running the command:

```
$ rostopic list
```

which should produce the following output:

```
/clock  
/gazebo/link_states  
/gazebo/model_states  
/gazebo/parameter_descriptions  
/gazebo/parameter_updates  
/gazebo/set_link_state  
/gazebo/set_model_state  
/joint_states  
/mobile_base/commands/motor_power  
/mobile_base/commands/reset_odometry  
/mobile_base/commands/velocity  
/mobile_base/events/bumper  
/mobile_base/events/cliff  
/mobile_base/sensors/imu_data  
/odom  
/rosout  
/rosout_agg  
/tf
```

Most of the Kobuki-related topics live under the namespace `/mobile_base`. The topic `/mobile_base/commands/velocity` accepts `Twist` messages to control the base while the `/mobile_base/events/cliff` and `/mobile_base/events/bumper` topics register events from the cliff and bumper sensors. The standard `/odom` topic is used to publish the robot's odometry data.

Since the Kobuki expects `Twist` commands on the topic

`/mobile_base/command/velocity` instead of `/cmd_vel`, try moving the robot forward at 0.2 meters per second by running the following command:

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

You should see the robot move across the table and fall off the edge where we created the gap. To keep the poor robot from bouncing around on the floor (or driving out of the scene if it landed on its wheels), type `Ctrl-C` to kill the `rostopic` command and then click on the **Edit** menu in the Gazebo GUI and select **Reset Model Poses**.

Since a real Kobuki robot is equipped with cliff sensors, why did the simulated robot drive off the edge of the table? The Kobuki's Gazebo model includes plugins for the cliff sensors and they do indeed detect the drop-off at the edge of the simulated table; however, in this example we controlled the robot through the topic `/mobile_base/commands/velocity` that sends `Twist` commands directly to the base controller and ignores any data coming from the sensors. In the next section, we will see how to examine the data returned by the simulated sensors and how to drive the robot more safely.

12.7.1 Accessing simulated sensor data

A real Kobuki robot has three cliff sensors (left, right and center) and three bump sensors (left, right and center), and the Kobuki's Gazebo model includes plugins for each of these sensors. This means that the simulated Kobuki can detect when it is about to drive off an edge or when it has run into something like a cinder block. The simulated sensor data are published on two ROS topics: `/mobile_base/events/cliff` and `/mobile_base/events/bumper`.

You can verify that the simulated cliff sensors are working by monitoring the `/mobile_base/events/cliff` topic using the command:

```
$ rostopic echo /mobile_base/events/cliff
```

Initially you will see the following output:

```
WARNING: no messages received and simulated time is active.  
Is /clock being published?
```

This warning is expected since we are running a simulation, not a real robot, and the cliff sensors only publish a message when at least one of them detects a drop-off.

Now repeat the experiment described in the previous section and set the robot heading for a table edge while keeping your eye on the terminal window monitoring the cliff messages. It might help to slow down the speed of the robot so this time change the linear speed to 0.1 meters per second:

```
$ rostopic pub -r 10 /mobile_base/commands/velocity \  
geometry_msgs/Twist '{linear: {x: 0.1}}'
```

As the robot passes over the table edge, you should see the following three messages on the `/mobile_base/events/cliff` topic:

```

sensor: 1
state: 1
bottom: 42647
---
sensor: 0
state: 1
bottom: 50660
---
sensor: 2
state: 1
bottom: 42647
---

```

These messages indicate that all three cliff sensors (labeled 0, 1 and 2 in the `sensor` field above) have entered `state 1` which means "cliff detected". The values displayed in the `bottom` field represent the distance to the floor although it is not clear from the Kobuki documentation what the units are. Also, keep in mind that the IR sensors used for cliff detection have a range of only 2-15 cm and the table tops are much higher than this off the floor.

To see the definition of the Kobuki cliff message, run the command:

```
$ rosmsg show -r kobuki_msgs/CliffEvent
```

The `-r` option stands for "raw" and includes the comments written in the message definition file. The key part of the output is:

```

# cliff sensor
uint8 LEFT    = 0
uint8 CENTER = 1
uint8 RIGHT   = 2

# cliff sensor state
uint8 FLOOR = 0
uint8 CLIFF = 1

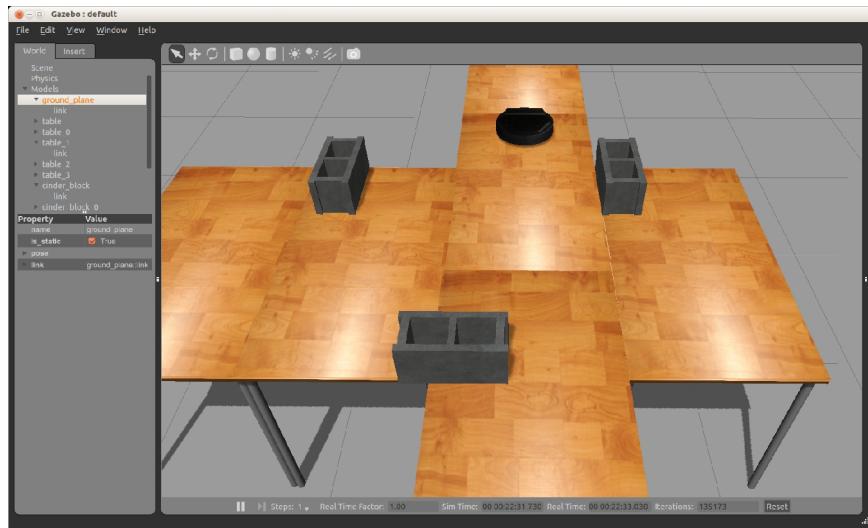
uint8 sensor
uint8 state

# distance to floor when cliff was detected
uint16 bottom

```

The message definition indicates that each message has three fields, `sensor`, `state` and `bottom` and that the `sensor` field can values 0, 1 and 2 to indicate the left, center and right IR sensors, while the `state` field takes the value 0 if a floor is detected or 1 if a cliff is detected.

To test the simulated bump sensors, first stop the robot by terminating the velocity publishing command if you still have it running. Then click on the **Edit** menu in the Gazebo GUI and select **Reset Model Poses**. You will notice that the robot is now lined up to run into one of the cinder blocks.



To monitor the messages on the bump sensors topic, run the command:

```
$ rostopic echo /mobile_base/events/bumper
```

Set the robot running toward the block:

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

Then flip back to the other terminal window and keep an eye out for bumper messages while watching the robot in Gazebo. The moment the robot strikes the block, you should see a series of messages on the bumper topic that look like this:

```
bumper: 1
state: 1
---
bumper: 1
state: 0
---
bumper: 1
state: 1
---
```

As with the cliff sensors, the bumpers are numbered 0, 1 and 2 for left, center and right. So the message above indicates that the center bumper has struck an object, bounced back enough to release the bumper, then struck again as the robot continue to move forward. (Depending on the processing speed of your computer, the robot might not bounce off the block so that you will only see the first message above.) To see the `BumperEvent` message definition, run the command:

```
$ rosmsg show -r kobuki_msgs/BumperEvent
```

```
# bumper
uint8 LEFT      = 0
uint8 CENTER   = 1
uint8 RIGHT    = 2

# state
uint8 RELEASED = 0
uint8 PRESSED  = 1

uint8 bumper
uint8 state
```

which shows us that there are three sensors that can have a state value of either 0 (released) or 1 (pressed).

With the velocity publisher still running, use your mouse to move the robot back away from the block but this time aim it so that it strikes the block on either the left or right side of the robot. This time you should see the bumper message indicate that either bumper 0 (left) or 2 (right) was depressed.

Note that running into an object does not cause the robot to stop*, just like the cliff sensors did not automatically stop the robot from plunging off the table. In the next section we'll see how to control the robot more safely.

(* When the simulated Kobuki runs head on against an object, it does appear to stop. However, this is simply because the forward motion is being impeded by the object, not

because the drive motors have stopped. As we saw above, the robot can actually bounce off the cinder block and continued to push against it.)

12.7.2 Adding safety control to the Kobuki

The `/mobile_base/command/velocity` topic provides direct control over the robot's motion while ignoring any data coming from the cliff or bump sensors. For example, you might want the robot to push a box across the floor in which case we don't want a bumper press to stop the robot. But the ROS programmers at Yujin Robot have created another control mechanism that will automatically do its best to keep the robot from harming itself or others.

The [`kobuki_safety_controller`](#) monitors events from the Kobuki's cliff, bumper and wheel-drop sensors and publishes an appropriate velocity command if any of these sensors indicate trouble. In the case of the cliff and bump sensors, the controller causes the robot to stop and back up whereas a wheel-drop event simply causes the robot to stop.

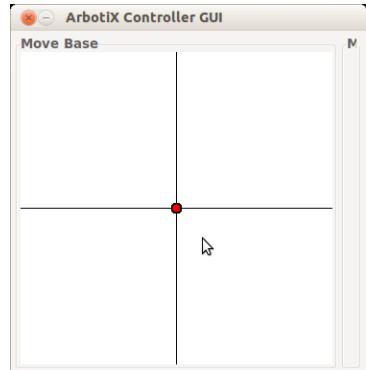
The safety controller works with the [`voes_cmd_vel_mux`](#) node that we explored in Chapter 8. This node can be easily configured to give the safety controller priority over all other velocity commands from other sources such as the navigation stack (`move_base`), a teleop node, or the command line.

To see the safety controller in action, first terminate any velocity publishing commands you might still have running from an earlier section. Then reset the test setup in Gazebo by clicking on the **Edit** menu and selecting **Reset Model Poses**.

This time, instead of publishing velocity commands directly from the command line, let's use the ArbotiX GUI which publishes `Twist` messages on the standard `/cmd_vel` topic:

```
$ arbotix_gui
```

You should see the trackpad control appear as shown on the right. Try moving the red control disc with your mouse and you will find that Kobuki does not move. This is because the `arbotix_gui` publishes `Twist` messages on the `/cmd_vel` topic while the simulated Kobuki is listening on the `/mobile_base/commands/velocity` topic.



Leave the ArbotiX Controller GUI running, then open another terminal and launch the Kobuki safety controller and `CmdVelMuxNodelet` using the following launch file:

```
$ rosrun roslaunch rbx2_gazebo kobuki_yocs_safety_controller.launch
```

This launch file loads the config file `yocs_safety_mux.yaml` from the directory `rbx2_gazebo/config`. The file has the same basic layout as the configuration files we used in Chapter 8 and is shown below:

```
subscribers:
  - name: "Safe reactive controller"
    topic: "safety_controller"
    timeout: 1.0
    priority: 2

  - name: "Joystick control"
    topic: "/joystick_cmd_vel"
    timeout: 1.0
    priority: 1

  - name: "Any node that publishes directly to /cmd_vel"
    topic: "/cmd_vel"
    timeout: 1.0
    priority: 0

publisher: "output/cmd_vel"
```

Note how we have given the safety controller top priority, followed by a joystick node that publishes on the `/joystick_cmd_vel` topic like we used in Chapter 8, then any other node that publishes on the standard `/cmd_vel` topic. Since the `arbotix_gui` node is publishing on `/cmd_vel`, you should now be able to control the robot using the trackpad; however, you will not be able to drive the robot off the table since the cliff sensors will alert the safety controller which in turn will override the input from the trackpad. Give it a try! (Just don't drive the robot backward over an edge since the robot only has cliff sensors in front.) You can also try driving into the cinder blocks and you should see that the safety controller causes the robot to "bounce" by backing it up a bit after impact.

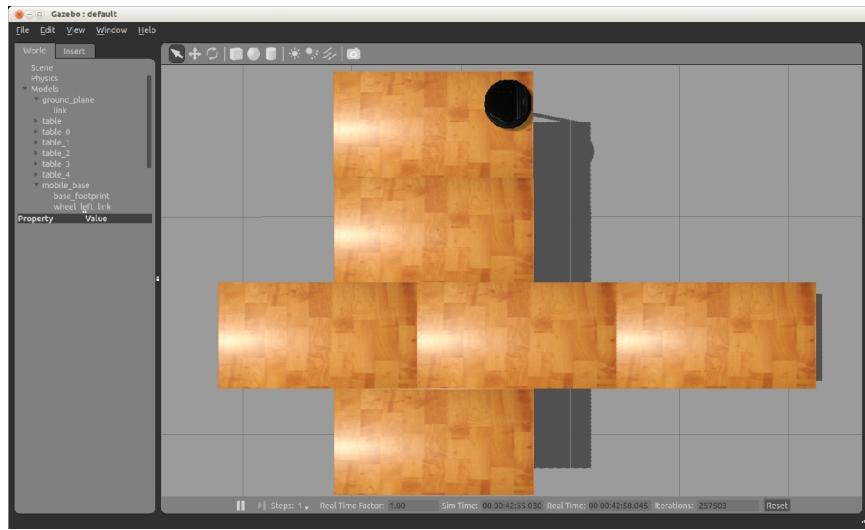
If you have a joystick, you can also add it to the mix by running the launch file:

```
$ rosrun roslaunch rbx2_nav mux_joystick_teleop.launch
```

Now drive the robot around with the joystick and observe how the safety controller keeps you out of trouble.

12.7.3 Running the `nav_square.py` script from Volume 1

As one final example, let us try one of our navigation scripts from *Volume 1* but this time using Gazebo instead of the ArbotiX simulator. Terminate any nodes you might have running from earlier sessions including the `arbotix_gui`. Assuming you have Gazebo running with the Kobuki playground world, either delete the cinder blocks or move them off the table, then position the robot as shown below:



The tables in this world are about 1 meter long and half a meter wide so two tables together makes a 1 meter square that should nicely accommodate our `nav_square.py` script from the `rbx1_nav` package. Notice how we have positioned the robot so that it is pointing to the left parallel to the table edge and a couple of inches away from the edge behind it. When you are finished positioning the robot, make sure it is no longer selected by clicking on the arrow tool on the toolbar and selecting anything other than the robot. (If you don't do this, the robot will not move when we send motion command below.)

To ensure the robot does not fall off the edge of the table, bring up the safety controller if it is not already running:

```
$ roslaunch rbx2_gazebo kobuki_yocs_safety_controller.launch
```

Now run the `nav_square.py` script:

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

The robot should do one loop around the perimeter of the table. If it gets too close to an edge, one of the cliff detectors will fire and cause the safety controller to kick in. The robot should then continue on its path.

NOTE: If you find that the robot simply spins in place when trying to make the first turn around the table, chances are the simulator needs to be restarted. (The author noticed on one trial that the simulated odometry had stopped registering rotations on the /odom topic.) Exit the Gazebo GUI then type `Ctrl-C` in the terminal used to launch the Kobuki playground world. Launch the world again and repeat the experiment described above.

From this demonstration we see how Gazebo can be used to test a robot in a realistic environment without even having access to the real robot or even the test setup (e.g. tables and cinder blocks in this case.) Our local robotics club (HBRC) holds a "TableBot Challenge" every year where the goal is to program a robot to locate an object on the table and move it from one location to another, all while keeping the robot from falling off. Using Gazebo would allow testing different robots and control strategies in a variety of environments before risking the robot in what might be a potentially expensive process of trial and error learning!

12.8 Loading Other Worlds and Objects

Fortunately for us, other Gazebo users have created a variety of worlds we can use to test our robot. If you are interested in robot soccer, you can load the official RoboCup field. Or you can test your robot in a simulated office, a kitchen, on top of an asphalt surface, or even inside a section of the International Space Station!

To load the Kobuki robot with the RoboCup soccer field, first terminate any running Gazebo simulations, then bring up Gazebo and the Kobuki in an empty world:

```
$ roslaunch kobuki_gazebo kobuki_empty_world.launch
```

When the Gazebo GUI is up, click on the **Insert** tab. Here you should see a list of models in your local cache as well as those in the database on the gazebosim.org site. (It might take a minute or so for the remote database to be loaded.) Open the remote database and select the **RoboCup 2009 SPL Field**. It will take a few moments to load, then place it in the scene with your mouse and click the left mouse button to drop it in place with the robot somewhere near the center of the field. Next add a **RoboCup 3D Simulation Ball** using the same technique.

To test the physics of this world, first make sure you do not have the Kobuki safety controller running from an earlier session, then bring up the `arbotix_gui` while remapping the default `cmd_vel` topic to the Kobuki's `/mobile_base/commands/velocity` topic:

```
$ arbotix_gui cmd_vel:=/mobile_base/commands/velocity
```

Now test your soccer skills by driving the robot using the simulated track pad and attempt to knock the ball into the goal. If you miss and the ball starts rolling out of the field, simply select **Reset Model Poses** from the **Edit** menu.

Try inserting other objects listed under the **Insert** tab. If you end up creating a world you like, you can save it using the **Save World As** option under the **File** menu. (By tradition, Gazebo world files are saved with either a `.world` or `.sdf` extension.) Be sure to save the world inside one of your ROS package directories so you can load it using a standard launch file later if desired. For an example of how this is done, take a look at the `kobuki_playground.launch` file in the `kobuki_gazebo` package:

```
$ roscd kobuki_gazebo/launch  
$ cat kobuki_playground.launch
```

You can also view it online [here](#).

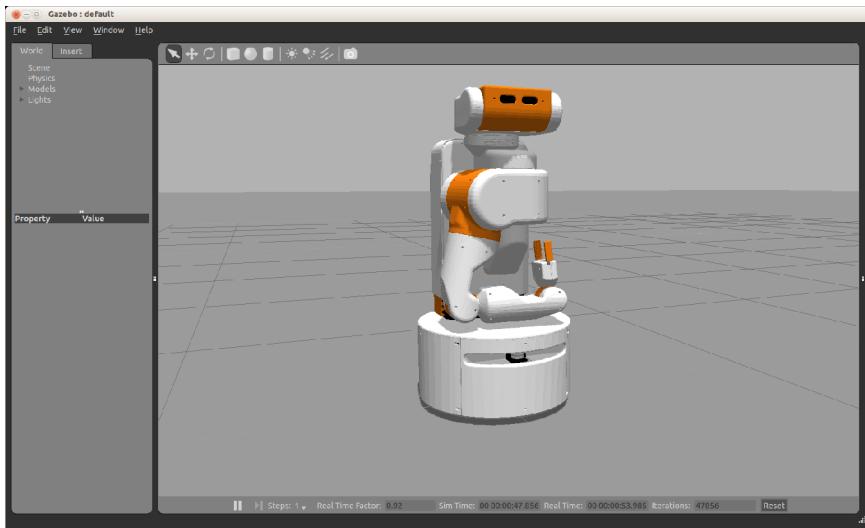
12.9 Testing the UBR-1 Robot in Gazebo

The [UBR-1](#) robot is similar to a smaller version of the PR2. It sports a 7-DOF arm with gripper, a telescoping torso joint, laser scanner, and a pan-and-tilt head with 3D camera. The official documentation for the UBR-1 for ROS Indigo can be found here:

http://unboundedrobotics.github.io/ubr1_preview

To bring up the UBR-1 in Gazebo using an empty world, terminate any running simulations, then run the command:

```
$ roslaunch ubr1_gazebo simulation.launch
```



If you get this far, we are ready to test some arm navigation.

12.9.1 UBR-1 joint trajectories

To test the basic functionality of the robot's arm and head joints, we can use a slightly modified version of the `trajectory_demo.py` script we wrote for Pi Robot in Chapter 11. All we need to do is change the name of the arm joints and set the goal positions accordingly. We also include some motion of the torso lift joint which makes the UBR-1 especially versatile when it comes to reaching objects with its arm and gripper. The modified script can be found in the file `ubr1_trajectory_demo.py` in the `rbx2_gazebo/scripts` directory. Try it first with the `reset` parameter set to `false`:

```
$ rosrun rbx2_gazebo ubr1_trajectory_demo.py _reset:=false
```

The UBR-1's arm and head should move upward and to the right while the torso joint moves upward. To lower the torso, re-center the head, and return the arm to the tucked position, run the same script with the `reset` parameter set to `true`:

```
$ rosrun rbx2_gazebo ubr1_trajectory_demo.py _reset:=true
```

The fact that we can use essentially the same script to control both the ultra-sophisticated UBR-1 and the Dynamixel-powered Pi Robot once again underscores the power of ROS. Since both robots and their underlying controllers are designed to use the same ROS `FollowJointTrajectoryAction` interface, our code can be programmed at a more abstract level without having to worry about the underlying hardware.

Now that we know that the basic simulation is working, let's try some arm navigation using MoveIt!.

12.9.2 The UBR-1 and MoveIt!

We can use the MoveIt! configuration package for the UBR-1 to try some of our earlier arm navigation scripts. When working with Pi Robot, we used RViz to visualize the scene. This time we will use Gazebo.

If you don't already have the UBR-1 simulation running, bring it up now:

```
$ roslaunch ubr1_gazebo simulation.launch
```

Next, fire up the MoveIt! nodes for the UBR-1:

```
$ roslaunch ubr1_moveit move_group.launch
```

Now let's try an inverse kinematics demo. The script called `ubr1_ik_demo.py` (located in the `rbx2_gazebo/scripts` directory) is similar to our earlier `moveit_ik_demo.py` script. The new script does not refer to the named poses (like "resting") that were defined in Pi Robot's MoveIt! configuration since these poses are not defined in the UBR-1 MoveIt! config. Also, the script uses the UBR-1's `arm_with_torso` planning group so that the torso joint can be included in the IK solution. The target pose we set for the gripper in the script would be too high for the robot to reach if it did not have the telescoping torso so we will see how useful that joint can be—and, how easily MoveIt! incorporates the joint into finding a solution.

To try out the script, run the following command while keeping an eye on the robot in Gazebo:

```
$ rosrun rbt_gazebo ubr1_ik_demo.py
```

You should see the arm and torso move upward and while the gripper assumes a horizontal pose pointing forward.

To tuck the arm back into its resting position, use the UBR-1 `tuck_arm.py` script included in the `ubr1_grasping` package:

```
$ rosrun ubr1_grasping tuck_arm.py
```

Next let's try a Cartesian path using the script `ubr1_cartesian_demo.py`.

```
$ rosrun rbx2_gazebo ubr1_cartesian_demo.py _cartesian:=true
```

The arm should first move along a non-Cartesian path to a starting configuration defined in the script. The next trajectory should move the gripper along a Cartesian triangle ending up back at the starting position.

Finally, let's try a pick-and-place demo. This demonstration uses simulated perception through the 3D head camera to pick a small cube off a tabletop and place it back down at a new location. (If you want to dig into the code to see how they did the perception part, take a look at the C++ files in the [ubr1_grasping/src](#) directory. For example, the file `shape_extraction.cpp` computes the shape of the cube sitting on the table using the Point Cloud Library).

Before getting started, terminate the current Gazebo session by selecting **Quit** from the **File** menu, then typing `Ctrl-C` in the terminal used to launch the simulation. Next, type `Ctrl-C` in the terminal used to run the `move_group.launch` file.

Now begin a new simulation using the `simulation_grasping.launch` file:

```
$ roslaunch ubr1_gazebo simulation_grasping.launch
```

This time you should see the UBR-1 together with a table and a cube on the table as shown below:



Next, bring up the MoveIt! nodes using the `grasping.launch` file which will also fire up the perception code that enables the simulated 3D camera to visually locate the cube on the table's surface:

```
$ roslaunch ubr1_grasping grasping.launch
```

Finally, run the pick-and-place demo:

```
$ rosrun ubr1_grasping pick_and_place.py --once
```

If all goes well, the UBR-1 should pick up the cube with its gripper and place it back on the table to the right of the robot. Note that the operation is not guaranteed to succeed but typically it will get it on the first try.

12.10 Real Pick-and-Place using the UBR-1 Perception Pipeline

The one aspect of real pick-and-place that we haven't yet covered is segmenting the visual scene so that the robot knows what to grasp and how to position and orient its gripper relative to the target object. A typical approach is to use the [Point Cloud Library](#) (PCL) as is done in Michael Ferguson's [chessbox](#) package for controlling a chess playing robot, as well as the [UBR-1 grasping package](#). Both packages use the C++ API for PCL but one could also try the [Python-PCL](#) bindings.

Although the programming details of the perception process are outside the scope of this volume, the basic procedure is as follows:

- fit a plane to the depth camera's point cloud to detect the support surface (usually a table top)
- remove the points from the cloud that belong to that surface (called "inliers") so that the remaining points must correspond to objects sitting on the surface
- group the remaining points into clusters based on the Euclidean distance between points
- fit either boxes or cylinders to the clusters and compute their poses
- once we have the pose of a target object, compute a set of suitable grasp poses for the gripper like we did earlier with the virtual pick-and-place demo

Note that this technique does not use the RGB image from the camera at all so in theory, the method would even work in the dark.

Although we will use the UBR-1 perception code for our demonstration, another useful resource is the [handle_detector](#) package that uses 3D point cloud data to locate grasping locations on different shaped objects. There is also the [moveit_simple_grasps](#) package for generating gripper poses for grasping different primitive shapes such as boxes and cylinders and the [GraspIt!](#) package that handles a greater variety of shapes. For object *recognition*, one can try the [object_recognition_kitchen](#) based on [ecto](#).

An alternative to PCL-based object detection or recognition is to use the AR tags we learned about in Chapter 10. In this approach, one or more AR tags are placed on the object to be grasped, then the [ar_track_alvar](#) package can be used to return the pose of the target. This pose could then be used more-or-less directly in our earlier [moveit_pick_and_place_demo.py](#) script as the starting point to generate possible grasp poses for the gripper.

12.10.1 Limitations of depth cameras

When using a Kinect or Xtion Pro for a depth camera for real pick and place tasks, keep in mind that the target needs to be at least 0.5 meters away from the camera since both cameras cannot measure depth inside this distance. In the meantime, the length of the arm and its position relative to the camera has to enable it to reach outside that 0.5 meter window where objects will be detected. Placing the camera at least a couple of feet above the arm's shoulder joint tends to work well to address both concerns.

12.10.2 Running the demo

The perception pipeline included in the [ubr1_grasping](#) package can be run as a standalone ROS action server to detect regularly shaped objects like boxes and cylinders sitting on a flat surface such as a table top. The grasping perception action server returns the poses and shapes of any detected objects thereby enabling us to use the results for a pick-and-place task with our own robot. Of course, our robot will need to have an arm and gripper as well as a depth camera.

To see how it works, you'll need a target object such as a small box or cylinder sitting on a table top within reach of your robot's arm and gripper. A travel-sized toothpaste box or pill bottle works well for small grippers. A soda can be used if your robot's gripper opens wide enough.

First terminate any running Gazebo simulations as well as any UBR-1 specific nodes or MoveIt! launch files.

To begin, launch the startup file for your robot. In the author's case, a slightly modified version of Pi Robot was used with the arm lowered by about 0.3 meters on the torso. This enables Pi's relatively short arm to reach target objects placed on a low table situated outside the 0.5 meter minimum range of the Kinect. The relevant launch file for this robot model is:

```
$ rosrun rbt2_bringup grasping_pi_robot_no_base.launch sim:=false
```

If your robot uses Dynamixel servos and the ArbotiX ROS driver like Pi Robot, you might want to keep the servos from overheating by running the `monitor_dynamixels.py` node we created in the chapter on ROS Diagnostics:

```
$ rosrun rbt2_diagnostics monitor_dynamixels.launch
```

Next, bring up the OpenNI node for your depth camera:

For the Microsoft Kinect:

```
$ rosrun freenect_launch freenect.launch
```

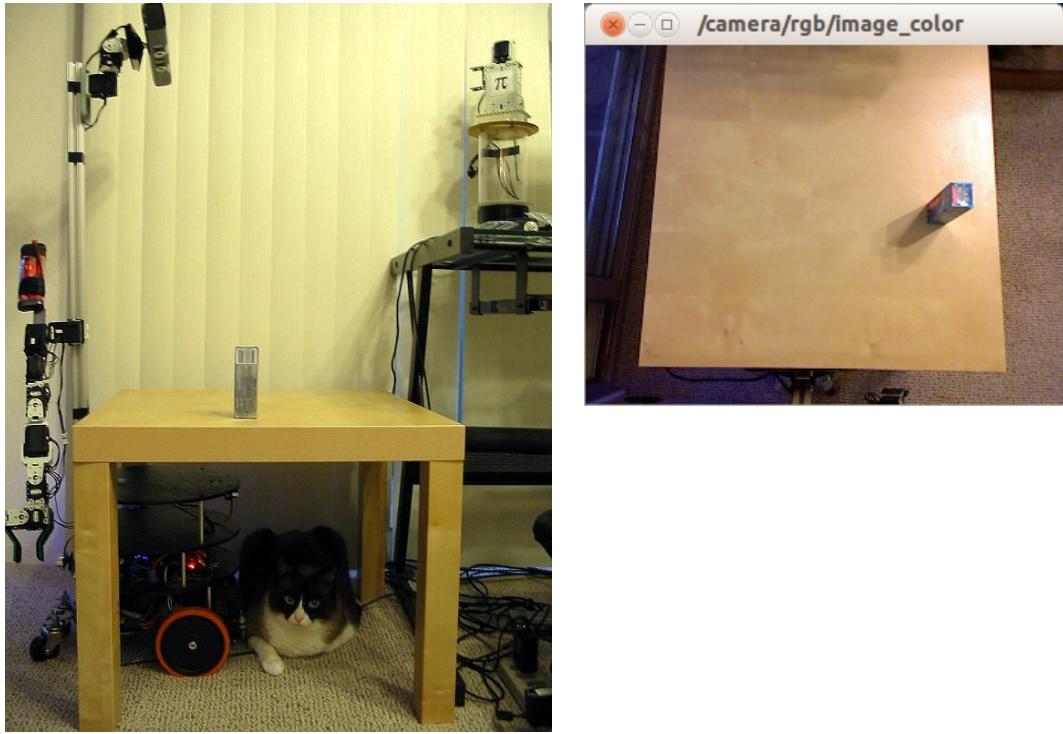
For the Asus Xtion, Xtion Pro, or Primesense 1.08/1.09 cameras:

```
$ rosrun openni2_launch openni2.launch
```

Fire up an `image_view` node so you can see what the camera sees:

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

Make sure the camera is at least a couple of feet or so above the table and angle it downward so that it is looking at the target object. The author used a small toothpaste box as the target object and the view through the camera looked like the image below on the right along with a side shot of the setup on the left:



Next, if you haven't done so already, disable MoveIt!'s octomap processing as we described at the end of section [11.27](#). Generating and displaying the occupancy map from the camera's depth cloud consumes a fair bit of CPU power and since we don't need it for this test, we'll get better performance by turning it off.

Now bring up the MoveIt! nodes for your robot. For Pi Robot we would run:

```
$ rosrun grasping_pi_robot_moveit_config move_group.launch
```

Next, fire up RViz with the `real_pick_and_place.rviz` config file:

```
$ rosrun rviz rviz -d `rospack find \
rbx2_arm_nav`/config/real_pick_and_place.rviz
```

To run the standalone UBR-1 perception pipeline, use the `ubr1_perception.launch` file in the `rbx2_gazebo` package:

```
$ roslaunch rbx2_gazebo ubr1_perception.launch
```

You should see output messages similar to:

```
process[basic_grasping_perception-1]: started with pid [440]
[ INFO] [1405723663.137069807]: basic_grasping_perception initialized
```

The `ubr1_perception.launch` file is nearly the same as the [grasping.launch](#) file in the `ubr1_grasping` package but includes a remapping of the `/head_camera` topic used by the UBR-1 to the `/camera` topic used by the `openni2` and `freenect` launch files.

The UBR-1 perception pipeline creates a ROS action server in the namespace `basic_grasping_perception/find_objects` that can be called to trigger a segmentation of the visual scene into a support surface (or surfaces) and the target object (or objects). The result is a set of object primitives (boxes and cylinders) and their poses that can be inserted into the MoveIt! planning scene. We can then use these poses to guide the robot's arm to grasp and move the target while avoiding collisions with the table top or other objects. Additional details will be given after we run the demo.

We are now ready to run our real pick-and-place script called [`real_pick_and_place.py`](#) found in the `rbx2_arm_nav/scripts` directory. This script begins by calling the UBR-1 perception action server to get the shapes and poses of the table top and supported objects. It then uses these poses with code from our earlier [`moveit_pick_and_place_demo.py`](#) script to generate appropriate grasp poses before executing the `pick()` and `place()` operations. (The place pose used for the demonstration was chosen so that the robot would move the object to a location just to the left of the torso and at the same distance in front of the robot as the original target location.)

To test the perception pipeline but without trying to move the arm, run the script with the `--objects` argument.

```
$ rosrun rbx2_arm_nav real_pick_and_place.py --objects
```

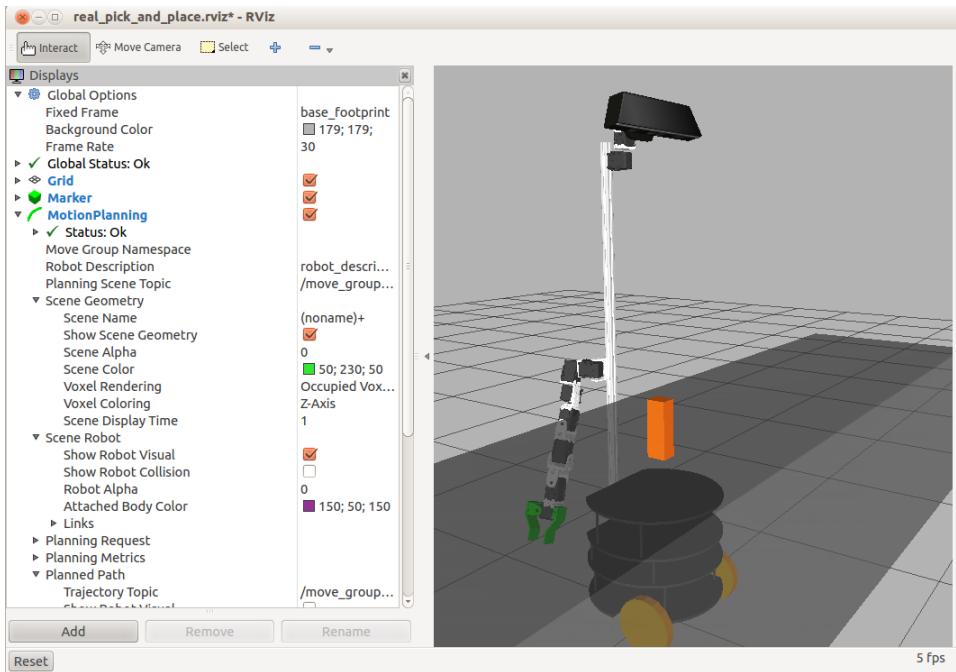
Initially, you might see a number of messages like the following in the terminal window:

```
[INFO] [WallTime: 1405913685.062874] Found 0 objects
```

Simply adjust the tilt angle of the camera either up or down until ideally only one object is detected:

```
[INFO] [WallTime: 1405913640.773310] Found 1 object
```

Back in RViz, the view should look something like the following:



Here we see that the UBR-1 perception pipeline has successfully identified the table (shown in transparent gray) and the shape and location of the box sitting on top (shown in orange). Note that the box might sometimes appear as a cylinder rather than rectangular (and vice versa for cylindrical targets) since the fitting process is not perfect. For the most part, this will not affect grasping as long as the width of the virtual object is roughly the same in both cases.

Assuming your target object can be detected, `Ctrl-C` out of the `real_pick_and_place.py` script, make sure your robot is ready for the real test, and run the script again with the `--once` argument. This will attempt to execute one pick-and-place cycle to grasp and move the object and then exit:

```
$ rosrun rbx2_arm_nav real_pick_and_place.py --once
```

If everything goes well, the robot should begin by lifting its arm around the end of the table taking care not touch the table with any part of the arm. The robot will then reach toward the target object and open its gripper in preparation for grasping. Once the gripper is moved into position around the target, the fingers should close so as to grasp the object. This completes the `pick()` operation. The script then executes the `place()` function which attempts to find an IK solution that will move the grasped object to the place pose defined in the script. If successful, the arm will move the object to the new location, lower it to the table and open the gripper to release it. The arm will then pull back and up to retreat from the object without knocking it over before returning to the resting position, once again avoiding any contact with the table along its trajectory.

Here is a [video](#) of Pi Robot executing a successful pick and place:



It is important to note that once the pick operation begins, there is no further visual feedback to guide the arm toward the target. In other words, once the table and target object are located by the perception pipeline, the arm begins to move along a preplanned trajectory all the way to the target. This means that the gripper will sometimes bump the target or fail to grasp it altogether. Similarly, the place operation assumes that the table is in the same position as it was originally detected. If we were to remove the table part way through the arm movement, the robot would simply drop the object on the floor at the place location. Finally, both the pick and place operations can fail due to a failure by the IK solver to find a solution. Creating an IKFast solver for your robot's arm can usually improve the reliability of finding solutions.

12.10.3 Understanding the `real_pick_and_place.py` script

As we have seen, executing pick-and-place using a real robot is very similar to running the same process with virtual objects. The main difference is that now we have to segment a real visual scene into a support surface and the object we want the robot to grasp. Fortunately, we were able to use the UBR-1 perception pipeline to do the visual processing for us. Once we have the poses of the relevant objects in the scene, we can run essentially the same code we used in the previous chapter where we did fake pick-and-place using the ArbotiX simulator.

The `real_pick_and_place.py` script combines the output from the UBR-1 perception pipeline with our earlier pick-and-place code found in the `moveit_pick_and_place_demo.py` script. Since we have already examined that script in detail, we will focus on the lines that involve getting the object information from the UBR-1 perception node.

Link to source: [real_pick_and_place.py](#)

```
import actionlib
from moveit_python import PlanningSceneInterface
from grasping_msgs.msg import *
```

Near the top of the `real_pick_and_place.py` script we import a few additional libraries and messages that we need to work with the UBR-1 node. The `actionlib` library is needed so we can connect to the UBR-1 `basic_grasping_perception` action server to get the object poses. From the `moveit_python` library created by Michael Ferguson we need the `PlanningSceneInterface` that includes Python wrappers for a number of MoveIt! utilities for adding and removing scene objects. The `grasping_msgs` package is used with the perception action server to trigger a visual segmentation and get back the results as we'll see next.

```
find_objects =
actionlib.SimpleActionClient("basic_grasping_perception/find_objects",
FindGraspableObjectsAction)
```

Here we assign a `SimpleActionClient` to the variable `find_objects` that connects to the UBR-1 `basic_grasping_perception/find_objects` action server and uses the message type `FindGraspableObjectsAction` found in the `grasping_msgs` package. We will use this action client to trigger a segmentation of the visual image which will return the poses of the table and target object.

```
goal = FindGraspableObjectsGoal()

# We don't use the UBR-1 grasp planner as it does not work with our gripper
goal.plan_grasps = False
```

```

# Send the goal request to the find_objects action server which will trigger
# the perception pipeline
find_objects.send_goal(goal)

# Wait for a result
find_objects.wait_for_result(rospy.Duration(5.0))

# The result will contain support surface(s) and objects(s) if any are detected
find_result = find_objects.get_result()

```

To trigger the perception pipeline, we first create an empty `goal` object using the `FindGraspableObjectsGoal` message type. Then we turn off the UBR-1's grasp planner which is specific to the UBR-1's parallel gripper. (We will use our own grasp generator as we did earlier with fake pick-and-place.) Next we send the empty goal to the action server using our `find_objects` action client which triggers the visual segmentation process on the action server. We then wait for the results for up to 5 seconds and store whatever is returned in the variable `find_results`. These results will be in the form of a `FindGraspableObjectsResult` message which contains a wealth of information about the shape and pose of any detected support surfaces and objects. You can see the full definition of the message by running the command:

```
$ rosmsg show grasping_msgs/FindGraspableObjectsResult
```

Now that we have the perception results we can get the poses of the support surface and target object.

```

for obj in find_result.objects:
    count += 1
    scene.addSolidPrimitive("object%d"%count,
                           obj.object.primitives[0],
                           obj.object.primitive_poses[0],
                           wait = False)

```

This loop cycles through all detected objects in the results and adds their shapes and poses to the MoveIt! planning scene.

```

# Choose the object nearest to the robot
dx = obj.object.primitive_poses[0].position.x - args.x
dy = obj.object.primitive_poses[0].position.y
d = math.sqrt((dx * dx) + (dy * dy))
if d < the_object_dist:
    the_object_dist = d
    the_object = count

# Get the size of the target
target_size = obj.object.primitives[0].dimensions

# Set the target pose

```

```
target_pose.pose = obj.object.primitive_poses[0]
```

As we loop through the objects, we keep track of the one that is closest to the robot. This will become our target. We get the size of the target from the dimensions of the object's primitive (e.g. box, cylinder, etc.) and the pose from the object's primitive_poses array.

```
target_pose.pose.orientation.x = 0.0
target_pose.pose.orientation.y = 0.0
target_pose.pose.orientation.z = 0.0
target_pose.pose.orientation.w = 1.0
```

Since we generally want the arm to approach the target with the gripper oriented horizontally, we set the target pose accordingly. Note however that you could also try using the object's pose directly for the target pose.

A similar loop in the script (which we won't repeat here) cycles through any detected support surfaces and inserts them into the planning scene as we did for objects. In our simple setup there is only one support surface so we can assume it is the table top in front of the robot.

```
place_pose = PoseStamped()
place_pose.header.frame_id = REFERENCE_FRAME
place_pose.pose.position.x = target_pose.pose.position.x
place_pose.pose.position.y = 0.03
place_pose.pose.position.z = table_size[2] + target_size[2] / 2.0 + 0.015
place_pose.pose.orientation.w = 1.0
```

Here we define the place_pose which is the position and orientation we want the object to be moved to after it is grasped. In this case we want the object to end up the same distance in front of the robot (position.x), then 3 cm to the left of the torso (position.y) and on the table surface (position.z) which we compute from the table height, half the target height and a calibration factor of 0.015 meters upward just to be sure we do not try to embed the object inside the table.

```
# Initialize the grasp pose to the target pose
grasp_pose = target_pose

# Shift the grasp pose half the size of the target to center it in the gripper
try:
    grasp_pose.pose.position.x += target_size[0] / 2.0
    grasp_pose.pose.position.y -= 0.01
    grasp_pose.pose.position.z += target_size[2] / 2.0
except:
    rospy.loginfo("Invalid object size so skipping")
    continue
```

```
# Generate a list of grasps
grasps = self.make_grasps(grasp_pose, [target_id])
```

The last step before starting the pick operation is to set the initial grasp pose and generate a collection of alternative grasps to try with the pick planner. Here we initialize the grasp pose to the target pose. We then shift the pose by half the target size in the x and z dimensions to better center the grasp on the object. The lateral shift in the y dimension of -1.0 cm was determined empirically to center the object between the gripper fingers.

The rest of the script is essentially the same as the `moveit_pick_and_place_demo.py` script that we discussed in detail in section [11.26](#).

12.11 Running Gazebo Headless + RViz

Gazebo can be run in *headless* mode which means it does not run the graphics engine or bring up the GUI. The robot can still be viewed in `RViz` however. This can be useful if you want to use Gazebo's physics engine but the display capabilities of `RViz`.

The easiest way to run a robot's Gazebo simulation in headless mode is for the launch file to be configured to accept a "headless" argument that is passed through to the `gzserver` process. While the Kobuki launch file we used earlier does not include this argument, the UBR-1 launch files do. In particular, let's take a look at the `UBR-1 simulation.launch` file we ran earlier:

Link to source: [simulation.launch](#)

```
1 <launch>
2
3   <!-- roslaunch arguments -->
4   <arg name="debug" default="false"/>
5   <arg name="gui" default="true"/>
6
7   <!-- We resume the logic in empty_world.launch, changing only the name of
the
8 world to be launched -->
9   <include file="$(find gazebo_ros)/launch/empty_world.launch">
10    <arg name="debug" value="$(arg debug)" />
11    <arg name="gui" value="$(arg gui)" />
12    <arg name="paused" value="false"/>
13    <arg name="use_sim_time" value="true"/>
14    <arg name="headless" value="false"/>
15  </include>
16
17  <!-- Add the robot, set head forward -->
```

```
18   <include file="$(find ubr1_gazebo)/launch/include/simulation.ubr1.xml" />
19   <node name="prepare_head" pkg="ubr_teleop" type="set_head_pose.py" args="0
0"/
20 >
21
22 </launch>
```

Note that Line 9 above includes the standard [empty_world.launch](#) file from the `gazebo_ros` package. Lines 10-14 pass a number of arguments to that launch file; in particular, a value of `false` is set for the `headless` parameter and a value of `true` is set for the `gui` parameter. Taken together, these cause the `empty_world.launch` file to run the `gzserver` process in headless mode and without the `gzclient` GUI.

To try it out, terminate any running Gazebo simulations and launch the UBR-1 grasping simulation in headless mode as follows:

```
$ roslaunch ubr1_gazebo simulation_grasping.launch headless:=true
gui:=false
```

You should see various startup messages related to the UBR-1 controllers, but the Gazebo GUI will not appear.

Wait for the simulation launch file to finish coming up, then open another terminal and run the `ubr1_grasping` launch file which runs the UBR-1 MoveIt! nodes as well as the perception pipeline:

```
$ roslaunch ubr1_grasping grasping.launch
```

Now fire up `RViz` with the `ubr1.rviz` config file found in the `rbx2_gazebo/config` directory:

```
$ rosrun rviz rviz -d `rospack find rbx2_gazebo`/config/ubr1.rviz
```

After a few seconds delay, you should see the model of the UBR-1 in `RViz`. Note that we do not yet see the table and cube as we did in the Gazebo GUI. This is because those objects are being represented in the simulation in a manner that cannot be displayed directly in `RViz`. They will become visible shortly.

Finally, run the UBR-1 pick-and-place demo and keep an eye on `RViz`:

```
$ rosrun ubr1_grasping pick_and_place.py --once
```

The first part of the pick and place script makes a call to the UBR-1's PCL-based perception pipeline that can detect planar surfaces and any objects on top of them. Although we do not see the table and cube initially in `RViz`, they are being simulated in Gazebo as is the UBR-1's depth camera.

Once the table and cube are detected, they are published as primitive shapes on the `/move_group/monitored_planning_scene` topic to which we are subscribing in `RViz` under **Planning Scene** in the **Motion Planning** display. You will therefore see the table and cube suddenly appear after which the UBR-1 should perform the pick-and-place operation as we saw earlier in Gazebo.

NOTE: You will probably notice that the movement of the grasped cube seems to lag the movement of the gripper. This is an artifact of some as yet unidentified bug in the MoveIt! `RViz` plugin. You can see smooth motion of the cube at the expense of jerky movement of the arm by changing the following settings in `RViz`:

- under **Scene Robot**, change the **Alpha** value from 0 to 1
- un-check the **RobotModel** display

Now run the pick-and-place demo again. This time the arm motion will not be as smooth but the grasped object will move in sync with the gripper.

13. ROSBRIDGE: BUILDING A WEB GUI FOR YOUR ROBOT

In a world of touch screens and point-and-click interfaces, running ROS nodes and launch files from the command line using multiple terminal windows might seem a little tedious. There are a number of ways to build a GUI for your robot so that it can be operated and monitored using a more intuitive interface. Some of these approaches require graphical toolkits such as [wxWidgets](#) or [Qt](#) and run natively under the OS you are using. For example, `RViz` is built on the Qt framework. However, there is no guarantee that a GUI built on these toolkits will run on your particular device or OS.

An alternative approach is based on web sockets, HTML and Javascript and will run on almost any device with a web browser. The key ingredient is [rosbridge_suite](#), a set of ROS packages created at Brown University for interacting with ROS topics and services using the [JSON](#) protocol over [websockets](#).

Using `rosbridge` will enable us to monitor and control our robot using nothing more than a web browser on just about any OS and platform, including tablets and smart phones. Furthermore, although we will use JavaScript for our examples, you can develop your client GUI using nearly any language that supports websockets such as Python, Java, Android, C# / .NET and C++ / Boost.

13.1 Installing the `rosbridge` Packages

To install `rosbridge_suite` and supporting packages, run the following commands:

```
$ sudo apt-get update  
$ sudo apt-get install ros-indigo-rosbridge-suite \  
  ros-indigo-robot-pose-publisher ros-indigo-tf2-web-republisher  
$ rospack profile
```

We will also need a number of JavaScript packages that are not typically distributed as Debian packages. These packages are already included in the `rbx2_gui/js` directory but if you need to reinstall them for some reason, you can run the `install-js-packages.sh` script as follows:

```
$ rosdep install rbx2_gui/scripts  
$ sh install-js-packages.sh
```

The script downloads the following Javascript packages:

```
http://cdn.robotwebtools.org/roslibjs/r5/roslib.min.js  
http://cdn.robotwebtools.org/mjpegcanvasjs/current/mjpegcanvas.min.js  
http://cdn.robotwebtools.org/EventEmitter2/0.4.11/eventemitter2.min.js  
http://cdn.robotwebtools.org/EaselJS/0.6.0/easeljs.min.js  
http://cdn.robotwebtools.org/ros2djs/r2/ros2d.min.js  
http://cdn.robotwebtools.org/ros3djs/current/ros3d.min.js  
http://threejs.org/build/three.min.js  
http://code.jquery.com/jquery-1.10.2.min.js  
http://d3lp1msu2r81bx.cloudfront.net/kjs/js/lib/kinetic-v5.0.1.min.js
```

Note that one additional Javascript package found at
<http://cdn.robotwebtools.org/nav2djs/current/nav2d.js> had to be
tweaked to fix an offset bug when using a map for navigation. The version included in
rbx2_gui/js has been adjusted to work with the test map found at
rbx2_nav/maps/test_map.yaml.

13.2 Installing the web_video_server Package

We will use the [web_video_server](#) package to stream live video from ROS topics over HTTP to our browser GUI. Install it now with the following command:

```
$ sudo apt-get install ros-indigo-web-video-server
```

To test the server, first connect to a camera. For a Kinect or Asus Xtion Pro fire up the OpenNI node:

For the Microsoft Kinect:

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

For the Asus Xtion, Xtion Pro, or Primesense 1.08/1.09 cameras:

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

If you are using a webcam, you can use the [usb_cam](#) driver that we installed in *Volume One*:

```
$ roslaunch rbx2_vision usb_cam.launch
```

Now open another terminal and run the web_video_server node:

```
$ rosrun web_video_server web_video_server
```

You should see the following startup message:

```
[ INFO] [1430486172.059059772]: Waiting For connections on 0.0.0.0:8080
```

Note that `web_video_server` uses port 8080 by default. If you are already using that port for something else, you can run `web_video_server` with a different port parameter. The following example uses port 8181:

```
$ rosrun web_video_server web_video_server _port:=8181
```

NOTE: When you run `web_video_server` with a different port as shown above, the port number is stored on the ROS parameter server. So if you run `web_video_server` in a later session without the port parameter, it will pick up the port number from the parameter server instead of the default port 8080. To override the value on the parameter server, you can specify the default port on the command line just as we did above with the alternate port. You can also use the included `web_video_server` launch file which sets the port parameter to the default 8080:

```
$ roslaunch rbx2_gui web_video_server.launch
```

If you need to run `web_video_server` on a different port, you can edit this file or make a copy of it and set the port accordingly.

With both the camera node and `web_video_server` running, you should now be able to view the camera image at the following URL:

```
http://localhost:PORT/stream_viewer?topic=/IMAGE_TOPIC
```

where `PORT` is the port we are running `web_video_server` on and `IMAGE_TOPIC` is the camera image topic we want to view. Using the default port 8080 and the image topic `/camera/rgb/image_raw` used by both the depth camera launch files and the `usb_cam.launch` file, we would use the URL:

```
http://localhost:8080/stream_viewer?topic=/camera/rgb/image_raw
```

Assuming you can see the live streaming image and your camera driver supports dynamic reconfigure, you can even change the image resolution on the fly using `rqt_reconfigure`:

```
$ rosrun rqt_reconfigure rqt_reconfigure
```

For example, when using an Xtion Pro camera, select the camera driver in the reconfigure window and change the image mode from QVGA_30Hz (5) to VGA_30Hz (2). Looking back at your web browser, you should see the image immediately change size from 320x240 pixels to 640x480.

Note that if you are using a depth camera, you can also view the depth image using web_video_viewer. For example, if you are using a Kinect or Xtion Pro camera, the URL to view the depth image is:

```
http://localhost:8080/stream_viewer?topic=/camera/depth/image
```

Your web browser should display a grayscale video stream where darker pixels represent points closer to the camera and lighter pixels represent points further away.

Four parameters often used to modify the streaming image are:

- `width` (integer, default: original width) The image stream will be resized to a new width and height. This parameter has to be used in conjunction with the `height` parameter.
- `height` (integer, default: original height) The image stream will be resized to a new width and height. This parameter has to be used in conjunction with the `height` parameter.
- `quality` (integer, default: 90) The jpeg image quality (1-100). This parameter can be used to reduce the size of the resulting mjpeg stream.
- `invert` (none, default:) Rotates the image by 180 degrees before streaming

So to view the color image at a size of 1280x960 (regardless of the original resolution) and a jpeg quality of 50, use the URL:

```
http://localhost:8080/stream_viewer?  
topic=/camera/rgb/image_raw&width=1280&height=960&quality=50
```

Using a lower quality setting can be useful for reducing the bandwidth needed for the video stream. To learn about other parameters applicable to the `web_video_server` node, see the [online Wiki page](#).

13.3 Installing a Simple Web Server (`mini-httdp`)

While `web_video_server` provides its own HTTP server for viewing video streams, `rosbridge` requires a separate web server for processing websocket requests.

Since not everyone runs a web server like Apache on their robot's computer, we will use a simple web server called `mini-httdp` to serve up our robot GUI. Run the following command to install the `mini-httdp` package:

```
$ sudo apt-get install mini-httdp
```

At the end of the installation, you will see the message:

```
You have to edit /etc/mini-httdp.conf and  
/etc/default/mini-httdp before running mini-httdp!
```

Ignore this for now as we will use our own configuration file located in the `rbx2_gui` directory.

Included in the `rbx2_gui/scripts` directory is a simple shell script called `mini-httdp.sh` that will launch the webserver on port 8181 and set the document directory to the `rbx2_gui` directory. Run the script now as follows:

```
$ rosrun rbox2_gui mini-httdp.sh
```

You will likely see the message:

```
bind: Address already in use
```

which can be ignored as there seems to be a bug in `mini-httdp` that causes this message to be displayed even when the port is free. If the port really is in use by another process, change the port in the `mini-httdp.conf` file to something else, like 8282 and try running it again. (But don't use ports 8080 or 9090 as these will be used by other processes needed by `rosbridge`.) Remember the port number you end up using as we will need it below.

13.4 Starting `mini-httdp`, `rosbridge` and `web_video_server`

The launch file `rosbridge.launch` in the `rbx2_gui/launch` directory can be used to fire up the `mini-httdp` webserver, `rosbridge` websocket server and `web_video_server`. Let's take a look at it now:

```
<launch>  
  
  <rosparam ns="/robot_gui">  
    maxLinearSpeed: 0.5  
    maxAngularSpeed: 2.0  
    videoTopic: /camera/rgb/image_color  
  </rosparam>
```

```

<node name="mini_httpd" pkg="rbx2_gui" type="mini-httpd.sh" output="screen" />

<node name="web_video_server" pkg="web_video_server" type="web_video_server"
output="screen">
    <param name="port" value="8080" />

    <node name="robot_pose_publisher" pkg="robot_pose_publisher"
type="robot_pose_publisher" output="screen" />

    <include file="$(find rosbridge_server)/launch/rosbridge_websocket.launch" />

    <include file="$(find rbx2_bringup)/launch/laptop_battery.launch" />

</launch>
```

First we set a few parameters under the namespace `/robot_gui`. These will be used later on in our Javascript files. Next we launch the `mini-httpd` server, followed by the `web_video_server` node and a `robot_pose_publisher` node which is only required if your `rosbridge` application using 2D navigation. We then launch the laptop battery node in case we are on a laptop and want to monitor its battery level. Finally, we run the `rosbridge` websocket server.

Before running the launch file, terminate any `web_video_server` node you might still have running from an earlier session. Then run the command:

```
$ roslaunch rbx2_gui rosbridge.launch
```

The output on your screen should look something like this:

```

process[mini_httpd-1]: started with pid [7605]
process[web_video_server-2]: started with pid [7608]
process[robot_pose_publisher-3]: started with pid [7619]
[ INFO] [1405216068.487706174]: Starting mjpeg server
[ INFO] [1405216068.487971928]: Bind(8080) succeeded
[ INFO] [1405216068.488020494]: waiting for clients to connect
process[rosbridge_websocket-4]: started with pid [7641]
process[rosapi-5]: started with pid [7648]
registered capabilities (classes):
 - rosbridge_library.capabilities.call_service.CallService
 - rosbridge_library.capabilities.advertise.Advertise
 - rosbridge_library.capabilities.publish.Publish
 - rosbridge_library.capabilities.subscribe.Subscribe
 - <class 'rosbridge_library.capabilities.defragmentation.Defragment'>
 - rosbridge_library.capabilities.advertise_service.AdvertiseService
 - rosbridge_library.capabilities.service_response.ServiceResponse
 - rosbridge_library.capabilities.stop_service.StopService
Launching mini-httpp...
process[laptop_battery-6]: started with pid [7654]
[INFO] [WallTime: 1405216069.059353] Rosbridge WebSocket server started on port
9090
[mini_httpd-1] process has finished cleanly
```

If you are on a desktop, you will see a couple of harmless warnings that the battery cannot be detected by the laptop battery node.

We are now ready to bring up our test robot GUI.

13.5 A Simple rosbridge HTML/Javascript GUI

The `rbx2_gui` package includes a simple HTML/Javascript GUI defined by the file `rbx2_gui/simple_gui.html`. To test the GUI on your machine, you will need an HTML5-compatible browser such as Google Chrome.

First make sure you have your video driver running. For a Kinect or Xtion Pro you can use:

For the Microsoft Kinect:

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

For the Asus Xtion, Xtion Pro, or Primesense 1.08/1.09 cameras:

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

If you are using a webcam, you can use the [usb_cam](#) driver that we installed in *Volume One*:

```
$ roslaunch rbx2_vision usb_cam.launch
```

Next, if you haven't already run the `rosbridge.launch` file, bring it up now:

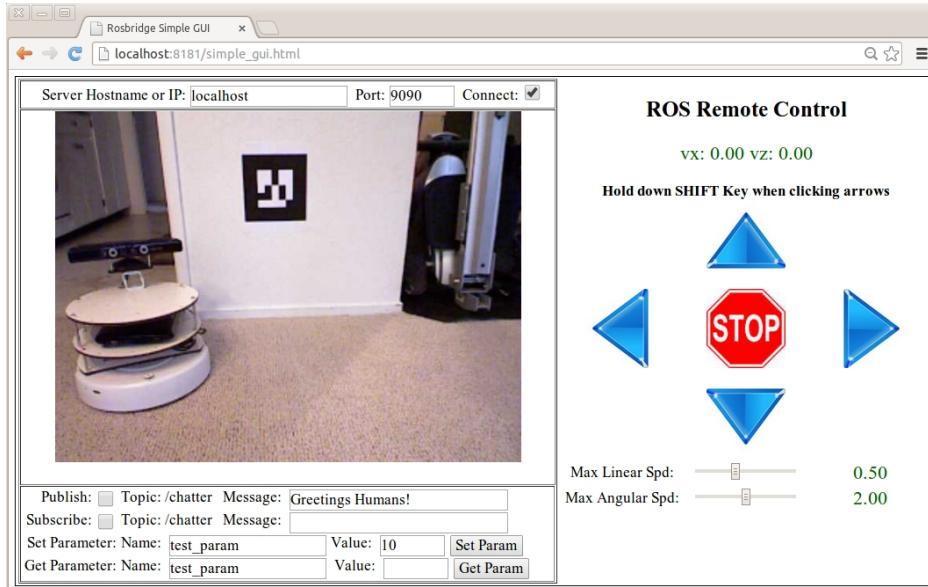
```
$ roslaunch rbx2_gui rosbridge.launch
```

Now point your browser to the following URL:

http://localhost:8181/simple_gui.html

(Change the port number if you were not able to use port 8181 for the `mini-httplibd` webserver.)

If everything goes well, your web browser should look something like the following:



If you don't see the video stream after a few seconds, try reloading the web page.

As we saw earlier, the `rosbridge.launch` file sets the video topic to `/camera/rgb/image_raw` which is the default topic when using either the `openni2_node.launch` file or the `usb_cam.launch` file. If your camera is publishing the video stream on a different topic, change the `videoTopic` parameter in `rosbridge.launch` and restart the launch file.

Next, let's focus on the areas above and below the video display. The row along the top includes a text box for the `rosbridge` host and port that have been given default values of `localhost` and `9090`. The checkbox is initially checked and causes the GUI to connect to the `rosbridge` server when it is first loaded. Try un-checking the box and the video stream should either disappear or freeze. Check the box again and the video should become live again. If the video doesn't reappear for some reason, simply reload the page. We'll see how to map HTML controls into actions when we look at the underlying Javascript.

Now take a look at the area below the video display. Notice that the **Publish** and **Subscribe** check boxes are currently un-checked. Open a terminal window and issue the command:

```
$ rostopic echo /chatter
```

You will probably see the following message:

```
WARNING: topic [/chatter] does not appear to be published yet
```

Keeping the terminal window visible, return to your web browser and check the **Publish** checkbox. You should see a series of messages in the terminal like the following:

```
data: Greetings Humans!
---
data: Greetings Humans!
---
data: Greetings Humans!
---
```

So checking the **Publish** checkbox results in publishing the message entered in the **Message** text box beside it. Now check the **Subscribe** checkbox. You should see the message "Greetings Humans!" appear in the **Message** text box to the right. Keeping both check boxes checked, enter some new text in the **Publish** text box. As you type, the message in the **Subscribe Message** box should mirror the new message. Returning to your terminal window, you should also see the new message displayed there as well.

You can test the parameter buttons and text boxes in a similar fashion. Entering a name and value for a parameter and clicking the **Set** button will store that parameter value on the ROS parameter server. Entering a parameter name and clicking the **Get** button will read the parameter value into the corresponding text box.

We will explore the `simple_gui.html` file and its associated Javascript below. But first let's try out the navigation controls using the fake TurtleBot.

13.6 Testing the GUI with a Fake TurtleBot

Before using the simple GUI to control a real robot, it is a good idea to test the navigation controls in simulation. First bring up the fake TurtleBot from the `rbx2_tasks` package:

```
$ roslaunch rbx2_tasks fake_turtlebot.launch
```

Next, bring up `RViz` with an appropriate configuration file:

```
$ rosrun rviz rviz -d `rospack find rbx2_nav`/config/sim.rviz
```

If you don't already have the `rosbridge` launch file running, fire it up now as well:

```
$ roslaunch rbx2_gui rosbridge.launch
```

We won't need a camera for this test but you can leave it running if you already have it up.

Point your browser to http://localhost:8181/simple_gui.html and arrange your windows so that both `RViz` and the browser window are visible at the same time. You should then be able to drive the robot around `RViz` by holding down the `Shift` key and clicking the large arrow keys on the GUI. (You can also use the arrows keys on your keyboard while holding down the `Shift` key.) The `Shift` key acts as a "dead man's switch" so that when you release it, the robot will stop. With the `Shift` key held down, each click of a given arrow will increment the robot's speed in that direction. You can also use the slider controls beneath the arrows to increase or decrease the maximum linear and angular speeds of the robot.

13.7 Testing the GUI with a Real Robot

If you have a real robot that responds to `Twist` messages published on the `/cmd_vel` topic, then you can use the simple web GUI right away to control your robot. Simply launch your robot's startup files as well as the `rosbridge.launch` file used in the previous section, then use the `Shift` key and the your mouse to move the robot around by clicking on the arrow keys.

You can also use the slider controls beneath the arrows to increase or decrease the maximum linear and angular speeds of the robot. If you want to change the default max speeds, change the appropriate parameter values in the `rosbridge.launch` file.

13.8 Viewing the Web GUI on another Device on your Network

If you have tablet or other device connected to the same router as the machine running `rosbridge`, then you should be able to open a web browser on that device and point it to the following URL:

`http://a.b.c.d:8181/simple_gui.html`

where `a.b.c.d` is the IP address of your `rosbridge` host. For example, if the machine running `rosbridge` has local IP address `10.0.0.3`, then enter the following URL on your tablet or other device:

`http://10.0.0.3:8181/simple_gui.html`

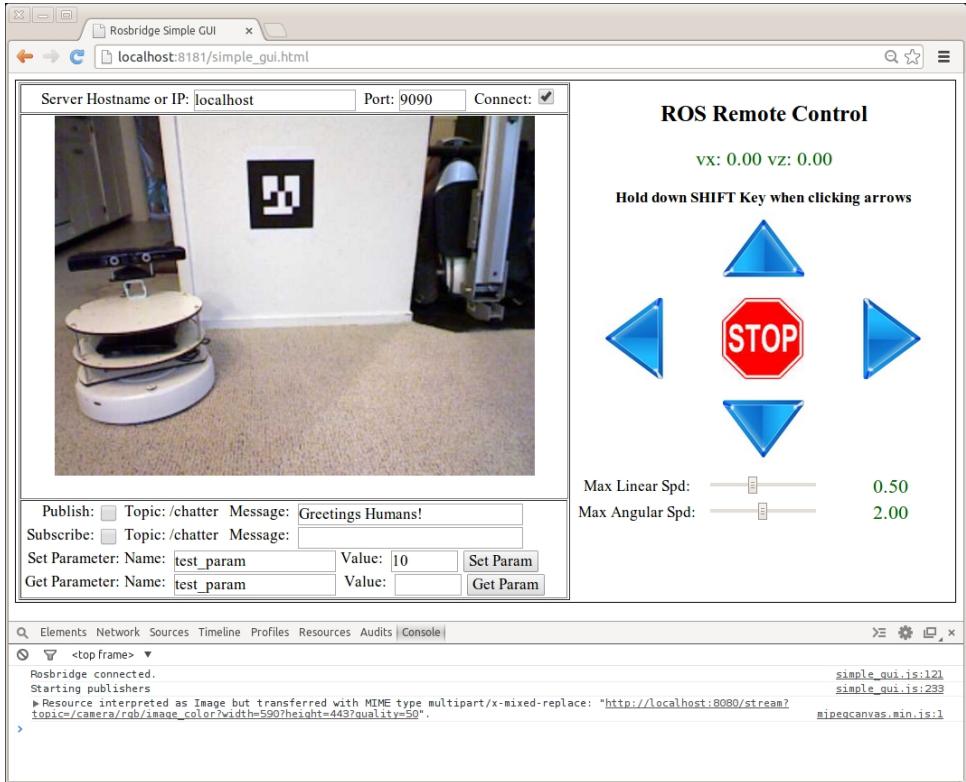
(If the two machines are not connected through the same router then you will need to configure any firewalls between the two device to allow access on ports 8080, 9090 and 8181.)

If you are using a tablet, then you can use the touch screen to control your robot by tapping on the arrow keys, although on some tablets this will magnify the screen as well. Since there is no `Shift` key on the tablet, the "dead man's switch" is implemented with a timer so that robot will stop moving if you stop tapping for 1 second. You will probably find the max speed sliders impossible to actually slide using touch; instead, tap to the left or right of the slider to make it move. In short, this simple GUI does not work very well with a touch interface so we will look at a more appropriate set of touch controls later in the chapter.

NOTE: If you have the web GUI up in more than one browser at a time, the robot's behavior will become erratic. In particular, it will likely move with a "stop-and-go" motion as one browser publishes the velocity commands you control and the other browser publishes "stop" commands since there is no mouse or touch input on that device. So if you are using a touch device, be sure to close your desktop web browser window and vice versa.

13.9 Using the Browser Debug Console

When developing your own `rosbridge` GUI, you will find your browser's debugging console invaluable for tracking down problems. If you are using Google Chrome or Firefox, hold down the keys `CTRL-SHIFT-j` together to either show or hide the console. With the console open in Google Chrome, your browser window should look something like the following:



The lower panel displays the output of the Console. You can display messages here in your Javascript code using the `console.log()` command. In the screen shot above, our script displayed the two messages "Rosbridge connected" and "Starting publishers". Beside each message is the script name and line number that generated the output. You can click on the line number to go directly to the location in your script.

Error messages will be displayed in red along with a line number and will generally be accompanied by a malfunctioning of your GUI. Fix the error in your script using your favorite editor, then reload the web page to see if the error goes away.

13.10 Understanding the Simple GUI

The simple GUI example is defined by two files in the `rbx2_gui` package:

`simple_gui.html` (in the top level directory) defines the layout of the GUI while `simple_gui.js` (located in the `js` subdirectory) handles the `rosbridge` code and any dynamic aspects of the interface.

13.10.1 The HTML layout: simple_gui.html

Let's take a look at the GUI layout as defined by the file simple_gui.html

Link to source: [simple_gui.html](#)

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML5//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html>
  <head>
    <meta content="text/html; charset=ISO-8859-1" http-equiv="content-type">
    <meta name="viewport" content="width=device-width, height=device-height,
initial-scale=0.65, user-scalable=yes">

    <title>Robot GUI</title>

    <script type="text/javascript" src="js/eventemitter2.min.js"></script>
    <script type="text/javascript" src="js/mjpegcanvas.min.js"></script>
    <script type="text/javascript" src="js/roslib.min.js"></script>
    <script type="text/javascript" src="js/simple_gui.js"></script>

    <script type="text/javascript">
      function init() {
        var readyStateCheckInterval = setInterval(function() {
          if (document.readyState === "complete") {
            init();
            clearInterval(readyStateCheckInterval);
          }
        }, 100);
      }
    </script>
```

We start by importing the various Javascript libraries we need including:

- eventemitter2.js – a event module required by mjpegcanvas.js
- mjpegcanvas.js – the Javascript client for connecting to the web_video_server node
- roslib.js – the main rosbridge library
- simple_gui.js – our own Javascript code that will be described below

We then define an `init()` function that simply waits for the page to be loaded completely before continuing. This ensures that we won't try to connect to the rosbridge and mjpeg servers until the various HTML elements are all in place.

```
<body onload="init();init_ros();">
```

In the main body tag, we run the `init()` function described above followed by the `init_ros()` function. This latter function is defined in our `simple_gui.js` file and takes care of connecting to `rosbridge` and `web_video_server` as we shall see.

```
<form name="simple_gui" method="get" action="./">

<table width="100%" border="0" cellpadding="2" cellspacing="2">
  <tbody>

    <!-- ===== Left Panel ===== -->
    <tr>
      <td>
        <table style="text-align: left; width: 100%;" border="1" cellpadding="2"
cellspacing="2">
          <tbody>
            <!-- *** Table Row 1: Connect/Disconnect to the rosbridge Server -->
            <tr>
              <td colspan="2">
                <div style="white-space: nowrap; vertical-align: middle; text-align: middle; margin-left: 10px;">
                  Server Hostname or IP: <input type="text" id="rosbridgeHost"
value="">&nbsp;
                  Port: <input type="text" id="rosbridgePort" size="5"
value="">&nbsp;<nobr />
                  Connect:&nbsp;<input id="connectROS" type="checkbox"
class="checkbox" checked onClick="connectDisconnect();">
                </div>
              </td>
            </tr>
          </tbody>
        </table>
      </td>
    </tr>
  </tbody>
</table>
```

Here we begin the layout of our GUI. We use a number of nested tables to split the screen into four main panels: upper left for the `rosbridge` connection and video image, lower left for a number of publish/subscribe boxes and buttons; upper right for the page title and status area; and lower right for the motion controls. (You'll probably find the source easier to read using your own editor or by viewing the [code on Github](#).) The first few lines shown above create a row just above the video display containing a pair of text boxes for the `rosbridge` host and port and a checkbox that toggles the connection off or on. The `connectDisconnect()` function assigned to the checkbox is defined in our `simple_gui.js` file.

```
<!-- *** Table Row 2: Display the video image -->
<tr>
  <td colspan="2" width="100%"><div id="videoCanvas" style="display:
block;"></div><br/></td>
</tr>
```

Next we insert a `<div>` tag named `videoCanvas` where the `mjpegcanvas` client will display the video image.

```
<!-- *** Table Row 3: Publish a message on the /chatter topic -->
```

```

<tr>
  <td>
    <table id="pubSubBlock">
      <tr>
        <td style="vertical-align: top; text-align: right;">Publish: <input id="chatterToggle" type="checkbox" class="checkbox" onClick="toggleChatter();">
        </td>
        <td style="vertical-align: top; text-align: left;">&nbsp;Topic:&nbsp;/chatter
          &nbsp;&nbsp;Message:&nbsp; <input type="text" size="30" id="chatterMessage" value="Greetings Humans!" onInput="updateChatterMsg(this.value);">
        </td>
      </tr>
    </table>
  </td>
</tr>

```

These lines add a checkbox to publish the message entered in the accompanying text box on the /chatter topic. The toggleChatter() function is defined in simple_gui.js. Note how we also assign the function updateChatterMsg() to the onInput event handler. The updateChatterMsg() function is also defined in simple_gui.js.

```

<!-- *** Table Row 4: Subscribe to the message on the /chatter topic -->
<tr>
  <td style="vertical-align: top; text-align: right;">
    Subscribe: <input id="chatterSub" type="checkbox" class="checkbox" onClick="subChatter();">
  </td>
  <td style="vertical-align: top; text-align: left;">&nbsp;
    Topic:&nbsp;/chatter&nbsp;&nbsp;
    Message:&nbsp; <input readonly="readonly" type="text" size="30" id="chatterData">
  </td>
</tr>

```

Here we have a checkbox and read-only text box used to subscribe to the /chatter topic and display the current message.

```

<!-- *** Table Row 5: Set a parameter value -->
<tr>
  <td style="vertical-align: top; text-align: right;">Set Parameter:</td>
  <td style="vertical-align: top; text-align: left;">&nbsp;Name:&nbsp; <input type="text" id="setParamName" value="test_param">&nbsp;Value:&nbsp; <input type="text" id="setParamValue" value="10" size="5"> <input id="setParameter" type="button" value="Set Param" onClick="setROSParam();">
  </td>
</tr>

<!-- *** Table Row 6: Get a parameter value -->
<tr>
  <td style="vertical-align: top; text-align: right;">Get Parameter:</td>

```

```

<td style="vertical-align: top; text-align: left;">>&nbsp;Name:&nbsp; <input  

type="text"  

    id="getParamName" value="test_param">&nbsp; Value:&nbsp; <input  

type="text"  

    readonly="readonly" id="getParamValue" size="5"> <input id="getParameter"  

type="button"  

    value="Get Param" onClick="getROSParam();">  

</td>  

</tr>

```

In these two blocks, we create buttons and text boxes to set or get a ROS parameter from the parameter server.

```

<!-- ===== Right Panel ===== -->
<td valign="top">
    <h1>ROS Remote Control</h1>
    <table width="100%" border="0" cellpadding="2" cellspacing="2">
        <tbody>
            <tr valign="top">
                <td colspan="3"><label id="cmdVelStatusMessage"></label><br>
                    <h3><label id="navInstructions"></label></h3>
                </td>
            </tr>
            <tr>
                <td colspan="3">
                    </td>
                </tr>
                <tr>
                    <td width="33%">
                    </td>
                    <td width="33%">
                    <td width="33%">
                    </td>
                </tr>
                <tr>
                    <td colspan="3">
                    </td>
                </tr>
        </tbody>
    </table>
</td>

```

Here we begin the right hand panel where we place four arrow icons for controlling the robot's motion. In the lines above, we first insert two labels for displaying the current linear and angular speeds as well as instructions on how to use the navigation arrows.

Next, we arrange the four navigation arrow icons in a square with a stop icon in the middle. First we place an upward pointing arrow icon stored in the `images` subdirectory and whose `onClick()` and `onTouchStart()` functions are set to `setSpeed('forward')`. The `setSpeed()` function is defined in `simple_gui.js` and publishes a `cmd_vel` message to move the robot. We also set the `onTouchEnd()` event to call the function `timedStopRobot()` that will stop the robot if the user's finger is lifted for more than one second from the screen when using a touch device. The `timedStopRobot()` function itself is defined in `simple_gui.js` as we shall see later on.

The next group of lines simply repeat the above pattern with the three other arrow icons for moving the robot back, left and right, as well as a stop icon in the middle that will stop the robot when clicked.

This brings us near the end of the file that looks like this:

```
<tr>
  <td align="right"><span style="font-size: 14pt;">Max Linear Spd:</span></td>
    <td><input type="range" id="maxLinearSpeed" min="0.01" max="0.5" step="0.01"
value="0.5"
      onChange="writeStatusMessage('maxLinearSpeedDisplay', this.value);"
      onMouseUp="setMaxLinearSpeed(this.value);"
onInput="maxLinearSpeedDisplay.value=this.value;"></td>
    <td><span style="font-weight: bold;"><output id="maxAngularSpeedDisplay"
size="4"></output></span></td>
  </tr>

<tr>
  <td align="right"><span style="font-size: 14pt;">Max Angular Spd:</span></td>
    <td><input type="range" id="maxAngularSpeed" min="0.01" max="2.0" step="0.01"
value="2.0"
      onChange="writeStatusMessage('maxAngularSpeedDisplay', this.value);"
      onMouseUp="setMaxAngularSpeed(this.value);"
onInput="maxAngularSpeedDisplay.value=this.value;"></td>
    <td><span style="font-weight: bold;"><output id="maxAngularSpeedDisplay"
size="4"></output></span></td>
  </tr>
```

These lines create a pair of slider controls for changing the max linear and angular speed of the robot. We set callbacks on the `onChange()`, `onMouseUp()` and `onInput()` events so that we can display the new settings and update the new values in our underlying Javascript variables.

13.10.2 The JavaScript code: `simple_gui.js`

Next, let's examine the `simple_gui.js` file located in the `js` subdirectory.

Link to source: [simple_gui.js](#)

We'll take the script in sections, starting at the top:

```
// Set the rosbridge and web_video_server port
var rosbridgePort = "9090";
var mjpegPort = "8080";

// Get the current hostname
thisHostName = document.location.hostname;

// Set the rosbridge and mjpeg server hostname accordingly
var rosbridgeHost = thisHostName;
var mjpegHost = thisHostName;

// Build the websocket URL to the rosbridge server
var serverURL = "ws://" + rosbridgeHost + ":" + rosbridgePort;

// The mjpeg client object that will be defined later
var mjpegViewer;

// The default video topic (can be set in the rosbridge launch file)
var videoTopic = "/camera/rgb/image_raw";

// The mjpeg video quality (percent)
var videoQuality = 50;
```

We begin by setting the port numbers for the `rosbridge` and `mjpeg` servers. Here we use the standard ports but you can change them if you are running the server processes on alternate ports. We also pick up the hostname from the server and then construct the websocket URL for connecting to the `rosbridge` server. The `mjpeg` host and port will be used later in the script for connecting to the video stream.

Next we set a default ROS topic name for the video stream that can be overridden in the `rosbridge.launch` file and we set the video quality to 50%. This setting generally produces an acceptable image and significantly reduces the load on the CPU from the `web_video_server` process. However, feel free to adjust to your liking.

```
// A variable to hold the chatter topic and publisher
var chatterTopic;

// A variable to hold the chatter message
var chatterMsg;
```

Here we define two variables to hold the chatter topic name and message, both of which will be defined later in the script.

```
// The ROS namespace containing parameters for this script
var param_ns = '/robot_gui';
```

Parameters set in the `rosbridge.launch` file are tucked underneath a namespace also defined in the launch file. By default, we use the namespace `"/robot_gui"`. If for some reason you want to use a different namespace, make sure you use the same name in both the launch file and the `simple_gui.js` file.

```
// Are we on a touch device?  
var isTouchDevice = 'ontouchstart' in document.documentElement;  
  
// A flag to indicate when the Shift key is being depressed  
// The Shift key is used as a "dead man's switch" when using the mouse  
// to control the motion of the robot.  
var shiftKey = false;
```

Here we test to see if we are on a touch device such as a tablet and if so, the variable `isTouchDevice` is set to `true`; otherwise it will be `false`. We can then use this variable to customize the display depending on the type of device we are on.

We use the variable `shiftKey` to indicate when the user is holding down the `Shift` key when using a conventional display and mouse. This will allow us to use the `Shift` key as a dead man's switch so that we can stop the robot if the key is released.

```
// The topic on which to publish Twist messages  
var cmdVelTopic = "/cmd_vel";  
  
// Default linear and angular speeds  
var defaultLinearSpeed = 0.5;  
var defaultAngularSpeed = 2.0;  
  
// Current maximum linear and angular speed (can be overridden in  
rosbridge.launch)  
var maxLinearSpeed = defaultLinearSpeed;  
var maxAngularSpeed = defaultAngularSpeed;  
  
// Minimum linear and angular speed  
var minLinearSpeed = 0.05;  
var minAngularSpeed = 0.1;  
  
// How much to increment speeds with each key press  
var vx_click_increment = 0.05;  
var vz_click_increment = 0.1;  
  
// Current desired linear and angular speed  
var vx = 0.0;  
var vz = 0.0;
```

The next set of variables relate to moving the robot base. Most ROS robots will subscribe to the `/cmd_vel` topic for instructions on how to move so we set that topic as the default here. We then set defaults for the maximum linear and angular speeds and how much to increment each speed when clicking on the navigation arrows.

```

// A handle for the publisher timer
var pubHandle = null;

// A handle for the stop timer
var stopHandle = null;

// The rate in Hz for the main ROS publisher loop
var rate = 5;

```

We will use a Javascript timer for publishing Twist messages on the /cmd_vel topic as well as the text message on the /chatter topic (if selected). We will also use a stop timer for automatically stopping the robot when using a touch device when the user lifts their finger off the controls. Here we define two variables to store handles to these timers and will set the timers themselves later in the code. The `rate` variable determines how often we publish messages.

```

// Get the current window width and height
var windowHeight = this.window.innerHeight;
var windowWidth = this.window.innerWidth;

// Set the video width to 1/2 of the window width and scale the height
// appropriately.
var videoWidth = Math.round(windowWidth / 2.0);
var videoHeight = Math.round(videoWidth * 240 / 320);

```

To ensure a good fit of the video stream regardless of the size of the user's display, we set the video width and height variables to be proportional to the current window width and height.

```

// The main ROS object
var ros = new ROSLIB.Ros();

// Connect to rosbridge
function init_ros() {
    ros.connect(serverURL);

    // Set the rosbridge host and port values in the form
    document.getElementById("rosbridgeHost").value = rosbridgeHost;
    document.getElementById("rosbridgePort").value = rosbridgePort;
}

```

Here we create the main `ROSLIB.Ros()` object and store it in the variable named `ros`. This object is defined in the `roslib.min.js` library that we imported at the top of the `simple_gui.html` file. We will use this object shortly to connect to `rosbridge`. We also initialize the `hostname` and `port` text fields in our GUI from the values defined earlier in the script.

```

// If there is an error on the back end, an 'error' emit will be emitted.
ros.on('error', function(error) {

```

```
        console.log("Rosbridge Error: " + error);
    });
}
```

The `ROSLIB` object defines a number of event callbacks. In these lines we set the "on error" callback to simply display the error in the Javascript console.

```
// Wait until a connection is made before continuing
ros.on('connection', function() {
    console.log('Rosbridge connected.');

    // Create a Param object for the video topic
    var videoTopicParam = new ROSLIB.Param({
        ros : ros,
        name : param_ns + '/videoTopic'
    });

    videoTopicParam.get(function(value) {
        if (value != null) {
            videoTopic = value;
        }
    });
}

// Create the video viewer
if (!mpegViewer) {
    mjpegViewer = new MJPEGCANVAS.Viewer({
        divID : 'videoCanvas',
        host : mjpegHost,
        port : mjpegPort,
        width : videoWidth,
        height : videoHeight,
        quality : videoQuality,
        topic : videoTopic
    });
}
}
```

Now we get into the heart of the script. Note how we place the key ROS-related functions inside the "on connection" callback for our main `ros` object. This ensures that we don't try to read or set ROS parameters or topics until we have a connection to the `rosbridge` server.

In the lines above, we first use the `ROSLIB.Param()` object to read in the video topic as defined in our `rosbridge.launch` file. We test for a null value in which case we will use the default topic set earlier in the script.

Next we define the `mjpegViewer` variable as an instance of the `MJPEGCANVAS.Viewer()` object. This object type is defined in the `mjpegcanvas.min.js` library that we imported in the `simple_gui.html` file. The `mjpeg` viewer object requires the ID of the `<div>` used in the HTML file to display the video stream. Recall that we used `<div id="videoCanvas">` in the `simple_gui.html` file. We also have to set the host, port, width, height, video quality

and video topic, all of which we have assigned to the appropriate variables earlier in the script.

```
// Create a Param object for the max linear speed
var maxLinearSpeedParam = new ROSLIB.Param({
    ros : ros,
    name : param_ns + '/maxLinearSpeed'
});

// Get the value of the max linear speed parameter
maxLinearSpeedParam.get(function(value) {
    if (value != null) {
        maxLinearSpeed = value;

        // Update the value on the GUI
        var element = document.getElementById('maxLinearSpeed');
        element.setAttribute("max", maxLinearSpeed);
        element.setAttribute("value", maxLinearSpeed / 2.0);

        writeStatusMessage('maxLinearSpeedDisplay',
maxLinearSpeed.toFixed(1));
    }
});
```

This next block attempts to read in a parameter value for `maxLinearSpeed` that might have been set in the `rosbridge.launch` file. If a non-null value is obtained, then it overrides the global `maxLinearSpeed` variable set near the top of the script. We then update the slider control on the GUI with the new values. Note that we have to do this inside the parameter callback since `roslib` functions run asynchronously to prevent locking up the overall script. We therefore want to be sure to update the form element just after the parameter value is returned.

The next block in the script is nearly identical to the one above (so we won't display it here) but sets the `maxAngularSpeed` in the same way. Next we turn to the chatter message.

```
// Create the chatter topic and publisher
chatterTopic = new ROSLIB.Topic({
    ros : ros,
    name : '/chatter',
    messageType : 'std_msgs/String',
});

// Create the chatter message
var message = document.getElementById('chatterMessage');
chatterMsg = new ROSLIB.Message({
    data : message.value
});
```

Here we set the `chatterTopic` variable to an instantiation of the `ROSLIB.Topic` object. The `ROSLIB.Topic` object takes parameters for the parent `ros` object, the topic

name and the message type. It also defines functions for advertising, publishing and subscribing as we will see later on. We also create a `ROSLIB.Message` object to store the chatter message and we initialize the message data with the text found in the `chatterMessage` text box on the GUI.

```
document.addEventListener('keydown', function(e) {
    if (e.shiftKey)
        shiftKey = true;
    else
        shiftKey = false;
    setSpeed(e.keyCode);
}, true);

document.addEventListener('keyup', function(e) {
    if (!e.shiftKey) {
        shiftKey = false;
        stopRobot();
    }
}, true);
```

These next two blocks assign callback functions to the `keydown` and `keyup` events. In particular, we look for the user to depress the Shift key and set a flag to indicate whether or not the Shift key is currently been pressed. Recall that we will use this flag to implement a dead man's switch when using the mouse to control the motion of the robot. If the Shift key is not depressed, we call the function `stopRobot()` which is defined later in the script. Otherwise we pass along any detected keycode (like the press of an arrow key) to the `setSpeed()` function, also defined later in the script.

```
// Display a line of instructions on how to move the robot
if (isTouchDevice) {
    // Set the Nav instructions appropriately for touch
    var navLabel = document.getElementById("navInstructions");
    navLabel.innerHTML = "Tap an arrow to move the robot";

    // Hide the publish/subscribe rows on touch devices
    var pubSubBlock = document.getElementById("pubSubBlock");
    pubSubBlock.style.visibility = 'hidden';
}
else {
    // Set the Nav instructions appropriately for mousing
    var navLabel = document.getElementById("navInstructions");
    navLabel.innerHTML = "Hold down SHIFT Key when clicking arrows";
}
```

Here we display a line of instructions for moving the robot depending on whether we are on a touch screen or a normal display with a mouse. When using a mouse, we remind the user to hold down the Shift key. Otherwise, the user is instructed to tap the arrow keys. For touch devices (which typically have smaller screens), we also hide the publish/subscribe section below the video display and show basically just the navigation controls.

```
// Start the publisher loop
console.log("Starting publishers");
pubHandle = setInterval(refreshPublishers, 1000 / rate);
```

Finally, we start the publisher loop. We use the Javascript `setInterval()` function that can be used like a timer to execute a function given as the first argument every `t` milliseconds as specified by the second argument. In our case, we are running the function called `refreshPublishers` (described below) every 200 milliseconds which gives us our desired rate of 5 times per second.

That concludes the "on connection" part of the script. We now turn to the remaining functions that can be called after the connection is up and running.

```
function toggleChatter() {
    var pubChatterOn = document.getElementById('chatterToggle').checked;
    if (pubChatterOn) chatterTopic.advertise();
    else chatterTopic.unadvertise();
}

function updateChatterMsg(msg) {
    chatterMsg.data = msg;
}
```

Recall that the `toggleChatter()` function is bound to the Publish checkbox defined in the `simple_gui.html` file. If the checkbox is checked, we call the `advertise()` function; otherwise we call `unadvertise()`. Note that advertising a topic does not automatically cause the corresponding message to be published. Publishing the message is taken care of by the next function.

```
function refreshPublishers() {
    // Keep the /cmd_vel messages alive
    pubCmdVel();

    if (chatterTopic.isAdvertised)
        chatterTopic.publish(chatterMsg);
}
```

Recall that the `refreshPublishers()` function fires on the timer we set earlier at a frequency of `rate` times per second. First we run the `pubCmdVel()` function (described below) to publish the current `Twist` message for controlling the robot's base. We then check to see if the `chatterTopic` topic is currently advertising and if so, we publish the current chatter message.

```

var cmdVelPub = new ROSLIB.Topic({
    ros : ros,
    name : cmdVelTopic,
    messageType : 'geometry_msgs/Twist'
});

function pubCmdVel() {
    vx = Math.min(Math.abs(vx), maxLinearSpeed) * sign(vx);
    vz = Math.min(Math.abs(vz), maxAngularSpeed) * sign(vz);

    if (isNaN(vx) || isNaN(vz)) {
        vx = 0;
        vz = 0;
    }

    var cmdVelMsg = new ROSLIB.Message({
        linear : {
            x : vx,
            y : 0.0,
            z : 0.0
        },
        angular : {
            x : 0.0,
            y : 0.0,
            z : vz
        }
    });

    cmdVelPub.publish(cmdVelMsg);
}

```

Publishing a `Twist` message on the `/cmd_vel` topic introduces a couple of new `ROSLIB` objects. First we create an instance of the `ROSLIB.Topic` object that takes the main `roslib` object, the topic name, and the message type as arguments. We then define the function `pubCmdVel()` that uses the current global variables `vx` and `vz` representing the desired linear and angular velocities of the robot and creates an instance of a `ROSLIB.Message` object configured with the `Twist` message type and `vx` and `vz` in the appropriate slots. At the end of the function, we publish the actual message on the `/cmd_vel` topic using the `publish()` function that is defined for all `ROSLIB.Topic` objects.

```

// Speed control using the arrow keys or icons
function setSpeed(code) {
    // Stop if the deadman key (Shift) is not depressed
    if (!shiftKey && !isTouchDevice) {

```

```

        stopRobot();
        return;
    }

    // Use space bar as an emergency stop
    if (code == 32) {
        vx = 0;
        vz = 0;
    }
    // Left arrow
    else if (code == "left" || code == 37) {
        vz += vz_click_increment;
    }
    // Up arrow
    else if (code == 'forward' || code == 38) {
        vx += vx_click_increment;
    }
    // Right arrow
    else if (code == 'right' || code == 39) {
        vz -= vz_click_increment;
    }
    // Down arrow
    else if (code == 'backward' || code == 40) {
        vx -= vx_click_increment;
    }

    var statusMessage = "vx: " + vx.toFixed(2) + " vz: " + vz.toFixed(2);
    writeStatusMessage('cmdVelStatusMessage', statusMessage);
}

```

Recall that in the `simple_gui.html` file that defines the layout of our GUI, we assigned the `setSpeed()` function to the `onClick` and `onTouchStart` events for the navigation arrow icons. Here at last we define that function. The speed code can either be a string such as 'forward' or 'left' or a keycode that will be available when a key is pressed such as an arrow key. Depending on the keycode detected, we increase or decrease the linear and/or angular velocities `vx` and `vz` appropriately. Since the `pubLoop()` is running continually on the timer we set earlier, these new values will be published on the `/cmd_vel` topic.

```

function stopRobot() {
    vx = vz = 0;
    var statusMessage = "vx: " + vx.toFixed(2) + " vz: " + vz.toFixed(2);
    writeStatusMessage('cmdVelStatusMessage', statusMessage);
    pubCmdVel();
}

function timedStopRobot() {
    stopHandle = setTimeout(function() { stopRobot() }, 1000);
}

function clearTimedStop() {
    clearTimeout(stopHandle);
}

```

The `stopRobot()` function is fairly self-explanatory. While the explicit call to `pubCmdVel()` is not strictly necessary because of our repeating `pubLoop()`, it doesn't hurt either and it is best to be safe when trying to stop a moving robot.

The `timedStopRobot()` function is used when using a touch device and calls `stopRobot()` after a delay of one second (1000 milliseconds). This function is assigned to the `onTouchEnd` event in our GUI so that if the user stops touching the arrow icons for one second or more, the robot will stop. The `clearTimedStop()` function is then used to cancel the stop timer.

```
function subChatter() {
    var subscribe = document.getElementById('chatterSub').checked;
    var chatterData = document.getElementById('chatterData');
    var listener = chatterTopic;

    if (subscribe) {
        console.log('Subscribed to ' + listener.name);
        listener.subscribe(function(msg) {
            chatterData.value = msg.data;
        });
    } else {
        listener.unsubscribe();
        console.log('Unsubscribed from ' + listener.name);
    }
}
```

Recall that the `subChatter()` function is bound to the `Subscribe` checkbox on the GUI. First we determine if the checkbox is checked and assign the result to the `subscribe` variable. We also get a pointer to the `chatterData` text box on the GUI so we know where to display the results.

Next we assign the `chatterTopic` object to the local `listener` variable. While not strictly necessary, the assignment reminds us that we are now running in "listen" mode rather than publishing. The `ROSLIB.Topic` object includes a `subscribe` function that takes another function as a callback. The callback function takes a single argument which is the message received on the topic subscribed to. In the code above, if the `subscribe` checkbox is checked, we create a callback that displays the received message in the `chatterData` textbox on the GUI. Otherwise, we call the `unsubscribe()` function to stop listening.

The remaining few functions in the script should now be fairly self-explanatory given the discussion above. We turn next to a more complete GUI using more modern Javascript libraries especially suitable to touch devices.

13.11 A More Advanced GUI using jQuery, jqWidgets and KineticJS

We can build a fancier GUI for our robot using some modern HTML5 libraries including [jQuery](#), [jqWidgets](#) and [KineticJS](#). These libraries provide functions for creating tabbed layouts, multi-layered panels, and user controls more suitable for touch screens. Although we won't provide a detailed break down of this more advanced GUI, the code is relatively straightforward once you know how to use these toolkits. For tutorials and documentation for the toolkits themselves, see the links above.

The new GUI is defined by the HTML file `robot_gui.html` in the `rbx2_gui` package and the Javascript file `robot_gui.js` in the `rbx2_gui/js` directory. There are also a small number of styles set in the file `styles/robot_gui.css`. The `robot_gui.html` file defines a number of tabs (multiple overlapping screens) which are defined by separate files in the `rbx2_gui/tabs` directory. In summary, our GUI is made up of the following directories and files:

- /rbx2_gui
 - robot_gui.hjhtml
 - /tabs
 - main.html
 - navigation.html
 - diagnostics.html
 - parameters.html
 - misc.html
 - /js
 - robot_gui.js
 - styles
 - robot_gui.css

For the Microsoft Kinect:

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

For the Asus Xtion, Xtion Pro, or Primesense 1.08/1.09 cameras:

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

If you are using a webcam, you can use the [usb_cam](#) driver that we installed in *Volume One*:

```
$ roslaunch rbt2_vision usb_cam.launch
```

Then launch the `rosbridge.launch` file if it is not already running:

```
$ rosrun rox2_gui rosbridge.launch
```

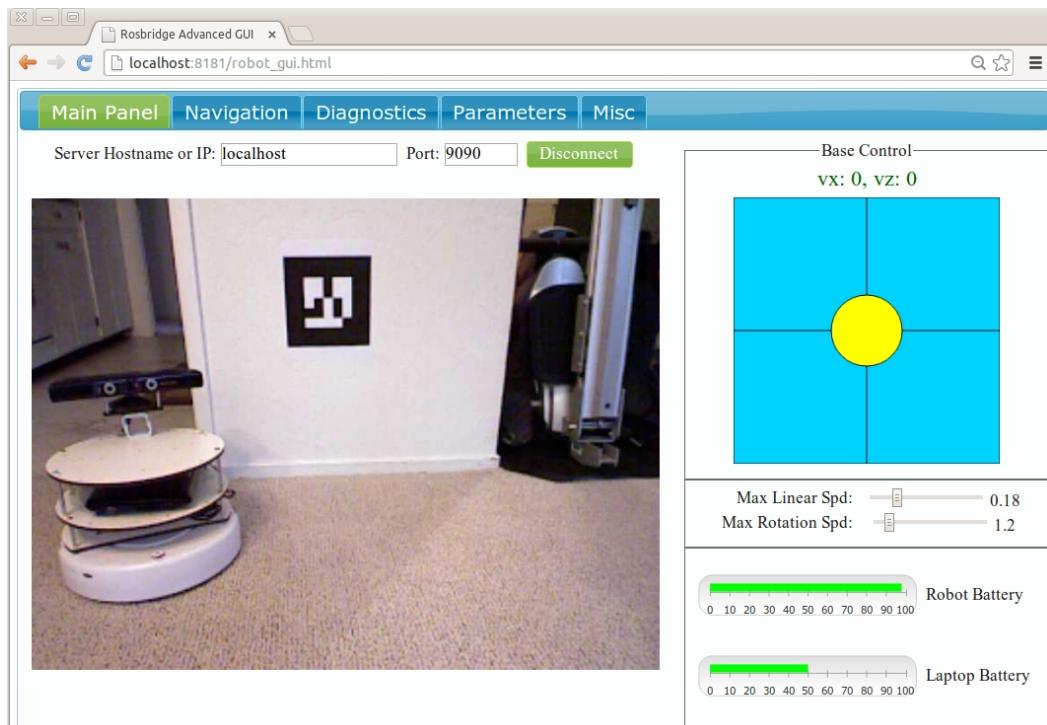
Notice how we are using the same `rosbridge` launch file as before but we will now point our browser to a different web page. This allows us to have different GUIs served up by the same `rosbridge` server.

Now point your browser to the following URL:

http://localhost:8181/robot_gui.html

(Change the port number if you were not able to use port 8181 for the `mini-httppd` webserver.)

Your browser window should look something like the following:



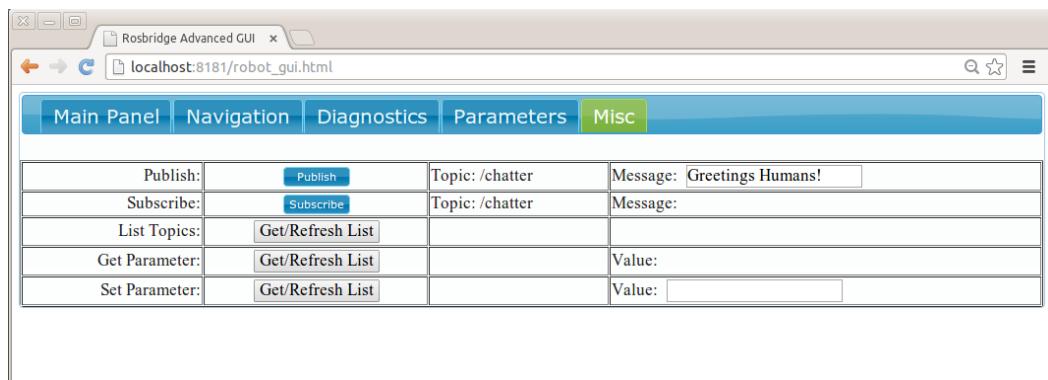
The tabs are arranged across the top—click or tap a tab label to switch to the corresponding screen. Only the **Main**, **Navigation** and **Misc** tabs currently have any function—the **Diagnostics** and **Parameters** tabs are just place holders to serve as additional examples. These tabs are created using the [jqWidgets](#) toolkit and the code can be found in the `robot_gui.html` file.

On the **Main** tab, the base control arrows have been replaced with a simulated touch pad (colored blue) similar to the base control provided by the ArbotiX Gui we have used earlier in the book. The touch pad is created using the [KineticJS](#) toolkit and the code is located in the `robot_gui.js` file. Drag the yellow disc either with the mouse or your finger (on a touch screen) and move it in the direction you want the robot to move. The further away from center you move the disc, the faster the robot will move. Release the disc and the robot will stop. The control pad is positioned so that if you are holding a tablet, the yellow disc should fall nicely under your thumb for controlling the base.

The maximum linear and angular speeds can be adjusted by the slider controls beneath the touch pad.

The charge level of the robot's battery and the laptop battery (if running on a laptop) are displayed using a pair of [jqWidgets fuel gauges](#) defined in the file `tabs/main.html`. Subscribing to the appropriate battery level topics is taken care of in the `robot_gui.js` file and will need to be customized for your robot. We will use the fake battery simulator to test them out below.

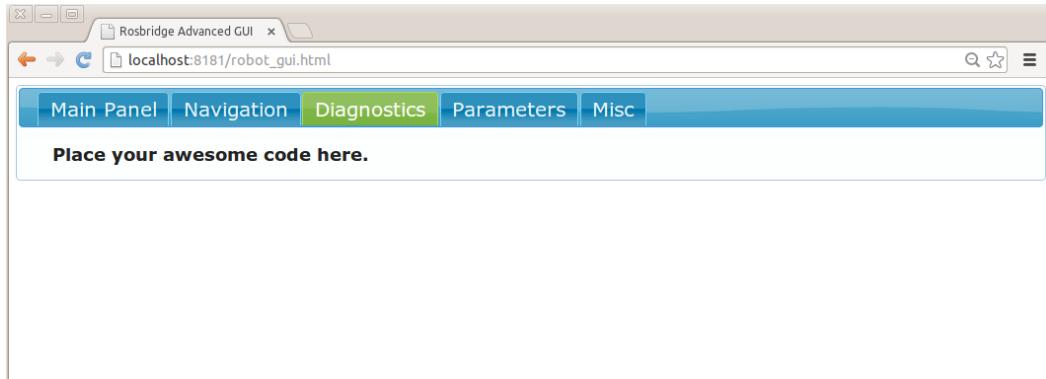
The **Misc** tab contains the same code we used with the simple GUI and is shown below:



TODO: Fix Set Param on the Misc screen

The **Diagnostics** and **Parameters** tabs are currently just place holders defined by the files `tabs/diagnostics.html` and `tabs/parameters.html`. Edit these files to

include your own code or remove the tabs altogether by editing the file `robot_gui.html`. For now, the pages simply display the following message:



Before describing the **Navigation** tab, let's fire up the fake TurtleBot using the test map found in `rbx2_nav/maps`. (This is the same map we used for the simulated navigation test in *Volume 1*). We will also set the fake battery runtime to 15 minutes (900 seconds). Run the `fake_turtlebot.launch` file from the `rbx2_tasks` package using the `battery_runtime` argument and `map` argument as follows:

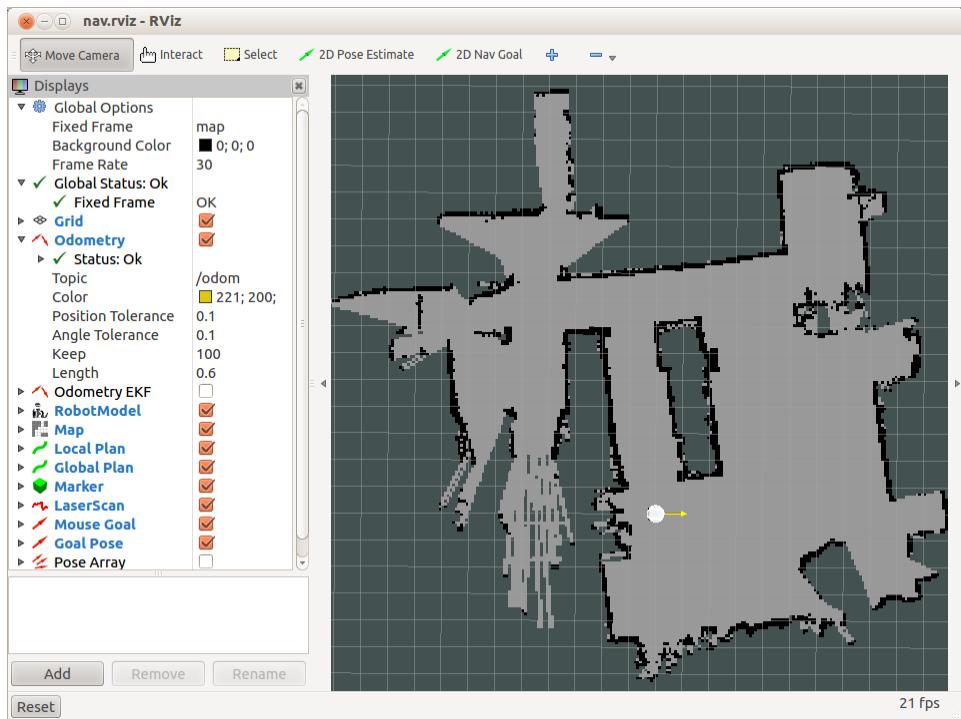
```
$ rosrun rbtasks fake_turtlebot.launch battery_runtime:=900 \
map:=test_map.yaml
```

With the fake battery node running, the Robot Battery fuel gauge on the rosbridge GUI should now start falling from 100 to 0 over the course of 15 minutes.

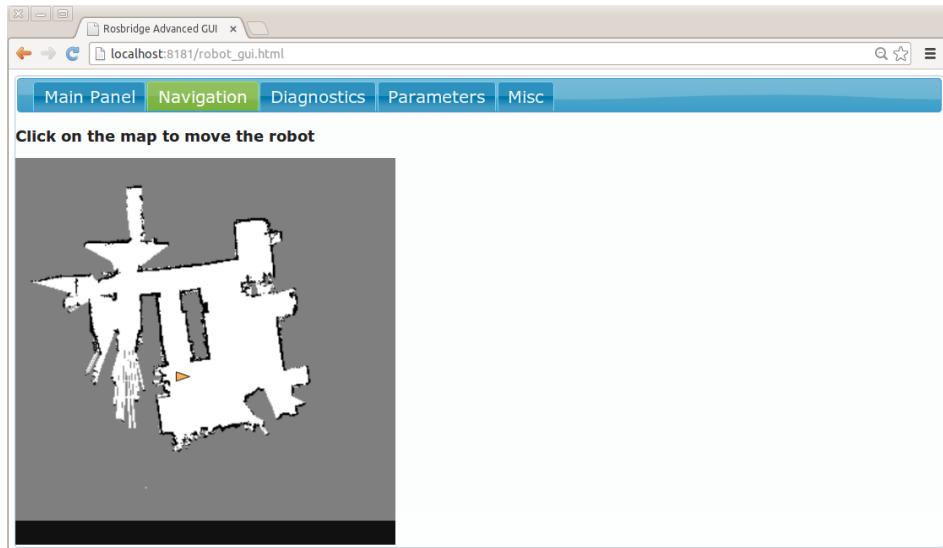
Next bring up `RViz` with the `nav.rviz` configuration file so that we can confirm that map has been loaded and the fake TurtleBot can be seen on the map.

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

The view in `RViz` should look something like this:



Finally, return to the rosbridge GUI in your web browser and click on the **Navigation** tab. You should see the same map appearing like this:



If you don't see the map, refresh or reload the web page, then click on the Navigation tab again.

The little orange triangle represents the position and orientation of the robot. To send the robot to a new location, click or tap on the goal location on the map. If you are using a mouse (not a touch screen), you can also click and hold then set the orientation as can be done in RViz. (On a touch screen, only a target position can currently be set.) After setting the new goal location, the robot marker should move across the map in your browser window while the fake TurtleBot follows the same path in RViz.

NOTE: The navigation function used here depends on the [nav2djs](#) package. The current version of this package appears to have a bug that causes the map to shift in the viewer by an amount equal to the offset of the map's origin. The version of the nav2d.js file found in the rbt2_gui/js directory has been tweaked to fix this bug.

13.12 Rosbridge Summary

These two examples should give you a good idea of how to design your own web-based controller for your robot. In summary, the following components are generally required to run on the computer attached to the robot:

- the rosbridge sever
- a camera driver
- the web_video_server node
- a webserver such as mini-httdp

In addition, the layout and functions of your GUI will be defined by one or more HTML files and Javascript files located in the document root of the web server. Any HTML5 web client should then be able to connect to your robot by pointing to the URL:

http://localhost:8181/your_gui.html

or, if you access your robot over a network:

http://x.y.z.w:8181/your_gui.html

where x.y.z.w is the IP address of the machine running the web server.

APPENDIX: PLUG AND PLAY USB DEVICES FOR ROS: CREATING UDEV RULES

Many robots use USB serial devices such as micro controllers, servo controllers, laser scanners or other peripherals. When these devices are plugged into a computer running Ubuntu, they are assigned a file name of the form `/dev/ttyUSBn`, `/dev/ttyACMn` or `/dev/ttySn` where `n` is an integer beginning with 0 and the choice of USB, ACM, or S in the filename depends on the hardware details of the device (e.g. UART). If more than one device in the same class is plugged in, the devices are numbered sequentially such as `/dev/ttyUSB0` and `/dev/ttyUSB1`.

The trouble is that Ubuntu assigns these device names on a first-come first-serve basis. So if we plug in device A followed by device B, the first device might be found on `/dev/ttyUSB0` and the second device on `/dev/ttyUSB1`. However, if we plug in device B first, the assignment will probably be the reverse.

This means that it is not a good idea to hard code these device names into configuration files or launch files since they will likely change if devices are unplugged then plugged back in again in a different order.

Fortunately, Ubuntu provides a way around the problem by allowing us to map unique hardware serial numbers to file names of our choosing. The process is fairly straightforward and takes only a few steps.

13.13 Adding yourself to the `dialout` Group

Before anything else, it is essential to add yourself to the Linux `dialout` group. This will ensure that you have the needed read/write access on all serial ports. Simply run the following command:

```
$ sudo adduser your_login_name dialout
```

For example, if your login name is `robomeister`, then use the command:

```
$ sudo adduser robomeister dialout
```

Then log out of your window session completely, and log back in again. Alternatively, simply reboot your computer.

13.14 Determining the Serial Number of a Device

Suppose we have an Arduino that we want to plug into one of our robot's USB ports. Before plugging in the device, run the following command in any open terminal:

```
$ tail -f /var/log/syslog | grep tty
```

Initially, this command should produce no output.

Now plug in your device (the Arduino in our example) and monitor the output in the terminal window. In my case, after a short delay I see the output:

```
Oct  9 18:50:21 pi-robot-z935 kernel: [75067.120457] cdc_acm 2-1.3:1.0:  
ttyACM0: USB ACM device
```

The appearance of `ttyACM0` in the output tells me that the Arduino was assigned device name `/dev/ttyACM0`. You can verify this with the command:

```
$ ls -l /dev/ttyACM0
```

which should produce an output similar to this:

```
crw-rw---- 1 root dialout 166, 0 Oct  9 18:50 /dev/ttyACM0
```

(Note that the timestamp should be close to the time when you plugged in the device.)

Now that we know that the device name for the Arduino is `/dev/ttyACM0`, we can use the `udevadm` utility to get its serial number:

```
$ udevadm info -q all -n /dev/ttyACM0 | grep -w ID_SERIAL_SHORT
```

In my case, the output is:

```
E: ID_SERIAL_SHORT=74133353437351200150
```

The number we want is the part after the equals sign above:

```
74133353437351200150
```

Now that we have the Arduino's serial number, let's map it to a device filename.

13.15 UDEV Rules

A mapping between serial numbers and device filenames can be created using a set of [UDEV rules](#) that we store in a file in the directory `/etc/udev/rules.d`. The first thing we need to do is create a file in this directory to hold our rules.

Begin by moving into the directory and list the files that are already there:

```
$ cd /etc/udev/rules.d  
$ ls -l
```

The output should look something like this:

```
-rw-r--r-- 1 root root 862 Nov 18 2013 70-persistent-cd.rules  
-rw-r--r-- 1 root root 716 Nov 17 2013 70-persistent-net.rules  
-rwxr-xr-x 1 root root 66 Jul 31 2014 99-android.rules  
-rw-r--r-- 1 root root 1157 Apr 5 2012 README
```

Create a file with a unique name like `40-my-robot.rules` using your favorite editor where the leading number is unique. Note that since we are creating a file in a system directory, we need to use `sudo`:

```
$ sudo gedit 40-my-robot.rules
```

Now add a line like the following:

```
SUBSYSTEM=="tty", ENV{ID_SERIAL_SHORT}=="74133353437351200150",  
MODE="0666", OWNER="robomeister", GROUP="robomeister",  
SYMLINK+="arduino"
```

where we have set the `ID_SERIAL_SHORT` variable to the value we found earlier for the Arduino—change this to reflect your device's ID. The `MODE` can always be set to `0666` and the `OWNER` and `GROUP` should be set to your Ubuntu username.

The `SYMLINK` variable defines the name we want our device to have: in this case, it will be `/dev/arduino` since `"/dev"` will be automatically prepended to the name we choose.

Save your changes and exit the editor. Then run the following command to reload the UDEV rules:

```
$ sudo service udev restart
```

That's all there is to it. We're now ready to test our new rule.

13.16 Testing a UDEV Rule

If your USB device is still plugged in, unplug it now. Wait a few seconds, then plug it back in. Now see if you can list the device's new filename:

```
$ ls -l /dev/arduino
```

In my case this returns the result:

```
lrwxrwxrwx 1 root root 7 Oct 10 18:21 /dev/arduino -> ttyACM0
```

As you can see, a symbolic link has been created between `/dev/arduino` and the real device file `/dev/ttyACM0`. If there had been another ACM device plugged in before we plugged in the Arduino, then `/dev/ttyACM0` would have been taken and the link would have been made to `/dev/ttyACM1` instead. But in either case, we can just use `/dev/arduino` in our ROS configuration files and not worry about the underlying physical device name anymore

13.17 Using a UDEV Device Name in a ROS Configuration File

Earlier in the book, we ran a number of examples using the ArbotiX Dynamixel driver and a USB2Dynamixel controller. In those cases we set the port name in the configuration files to be `/dev/ttyUSB0`. If we create a UDEV rule so that our USB2Dynamixel's serial number maps into the symlink `/dev/usb2dynamixel`, then we can use this port name instead in the configuration file. For example, the `pi_robot_head_only.yaml` file found in the `rbx2_dynamixels/config/arbotix` directory could be written as follows:

```
port: /dev/usb2dynamixel
baud: 1000000
rate: 100
sync_write: True
sync_read: False
read_rate: 10
write_rate: 10

joints: {
    head_pan_joint: {id: 1, neutral: 512, min_angle: -145, max_angle: 145},
    head_tilt_joint: {id: 2, neutral: 512, min_angle: -90, max_angle: 90}
}

controllers: {
```

```
    head_controller: {onboard: False, action_name:  
head_controller/follow_joint_trajectory, type: follow_controller, joints:  
[head_pan_joint, head_tilt_joint]}  
}
```

Note that we have simply replaced `/dev/ttyUSB0` with `/dev/usb2dynamixel` for the `port` value. Now it does not matter if the USB2Dynamixel controller is assigned a different USB number by the operating system some time later—our configuration file will still work without modification.

