Wyatt S. Newman

A Systematic Approach to Learning Robot Programming with ROS

Contents

List of Figures

Preface

ROS (Robot Operating System) is becoming the de facto programming approach to modern robotics. The ROS wiki (`http://www.ros.org/history/`) claims:

> The ROS ecosystem now consists of tens of thousands of users worldwide, working in domains ranging from tabletop hobby projects to large industrial automation systems.

Why ROS? In 1956, Joseph Engelberger founded Unimation, Inc., the worlds first robotics company[?]. Over the last half century, however, advances in robotics have been disappointing. Robotics research around the world has largely produced demonstrations and curiosities that died in the lab. New researchers in the field typically started from scratch, building new robots from the ground up, tackling the problems of actuator and sensor interfacing, building low-level servo controls, and typically running out of resources before achieving higher levels of robot competence. These custom machines and custom software were seldom re-used for follow-on work.

It was recognized that this pattern of building towers of Babel was unproductive, and that the task of building more intelligent robots would require a sustained, collaborative effort that could build on foundations to reach higher levels of competence. In 1984, Vincent Hayward and Richard Paul introduced RCCL (Robot Control C Library) [?] as an approach to address this long-standing problem. Unfortunately, RCCL did not gain sufficient acceptance among robotics researchers. Both National Instruments and Microsoft introduced products to try to attempt to bring standardization to robot programming [?],[?]. However, researchers found these approaches lackingand expensive.

Origins of the "Robot Operating System" were initiated at the Stanford Artificial Intelligence Lab in 2007 [?]. This attempt to unify fragmented approaches to robotics was adopted by Google and supported via Willow Garage from 2008-2013 [?], and subsequently by Googles Open Source Robotics Foundation [?] (OSRF) from 2013 to present. The ROS approach followed the modern trend of open-source software and distributed collaboration. Further, it bridged to and leveraged parallel open-source efforts, including OpenCV [?], PointCloudLibrary [?], Open Dynamics Engine [?], Gazbeo [?], and Eigen [?]. While ROS may be similar to RCCL in its openness and accessibility to researchers, Googles 7 years of ongoing support may be credited with ROS surviving a crucial incubation period.

ROS is now used throughout the world in academia, industry and research institutions. Developers have contributed thousands of packages, including solutions from some of the worlds leading experts in targeted areas. New robot companies are offering ROS interfaces with their products, and established industrial robot companies are introducing ROS interfaces as well. With widespread adoption of ROS as the de facto standard approach to robot programming, there is new hope for accelerating the capabilities of robots. In the recent DARPA Robotics Challenge, most of the teams who qualified used ROS. Developers of new self-driving cars are exploiting ROS. New robot companies are springing up with leaps in capabilityin part driven by ROS resources. Given the momentum and track record of ROS, it is clear that today's robotics engineer must be skilled in ROS programming.

What is ROS? The name "Robot Operating System" is arguably a misnomer. Defining ROS succinctly is difficult, since it is comprised of myriad aspects, including style of programming (notably, relying on loosely-coupled, distributed "nodes"); interface definitions and paradigms for communications among nodes; interface definitions for incorporation of libraries and packages; a collection of tools for visualization, debugging, data logging and system diagnostics; a repository of shared source code; and bridges to multiple useful, independent open-source libraries. ROS is thus more of a way of life for robot programmers than simply an operating system. Definitions of ROS are drawn from the following sources.

Some definitions of ROS are:

From the ROS wiki [**?**]:

> ROS is an opensource, metaoperating system for your robot. It provides the services you would expect from an operating system, including hardware abstraction, lowlevel device control, implementation of commonlyused functionality, messagepassing between processes, and package management. It also provides tools and libraries for obtaining, building, writing, and running code across multiple computers.
>
> The primary goal of ROS is to support code reuse in robotics research and development. ROS is a distributed framework of processes (aka nodes) that enables executables to be individually designed and loosely coupled at runtime. These processes can be grouped into Packages, which can be easily shared and distributed. ROS also supports a federated system of code Repositories that enable collaboration to be distributed as well. This design, from the filesystem level to the community level, enables independent decisions about development and implementation, but all can be brought together with ROS infrastructure tools.

Online comment by Brian Gerkey [**?**]:

> I usually explain ROS in the following way:
>
> 1. plumbing: ROS provides publish-subscribe messaging infrastructure designed to support the quick and easy construction of distributed computing systems.
> 2. tools: ROS provides an extensive set of tools for configuring, starting, introspecting, debugging, visualizing, logging, testing, and stopping distributed computing systems.
> 3. capabilities: ROS provides a broad collection of libraries that implement useful robot functionality, with a focus on mobility, manipulation, and perception.
> 4. ecosystem: ROS is supported and improved by a large community, with a strong focus on integration and documentation. ros.org is a one-stop-shop for finding and learning about the thousands of ROS packages that are available from developers around the world.

From the text ROS by Example:

> The primary goal of ROS (pronounced "Ross") is to provide a unified and open source programming framework for controlling robots in a variety of real world and simulated environments.

Regarding ROS plumbing (from "ROS by Example" [**?**]):

> The core entity in ROS is called a node. A node is generally a small program written in Python or C++ that executes some relatively simple task or process. Nodes can be started and stopped independently of one another and they communicate by passing messages. A node can publish messages on certain topics or provide services to other nodes.
>
> For example, a publisher node might report data from sensors attached to your robot's microcontroller. A message on the `/head_sonar` topic with a value of 0.5 would mean that the sensor is currently detecting an object 0.5 meters away. Any node that wants to know the reading from this sensor need only subscribe to the `/head_sonar` topic. To make use of these values, the subscriber node defines a callback function that gets executed whenever a new message arrives on the subscribed topic. How often this happens depends on the rate at which the publisher node updates its messages.
>
> A node can also define one or more services. A ROS service produces some behavior or sends back a reply when sent a request from another node. A simple example would be a service that turns an LED on or off. A more complex example would be a service that returns a navigation plan for a mobile robot when given a goal location and the starting pose of the robot.

Approach to this Presentation: ROS has many features, tools, style expectations and quirks. ROS has a steep learning curve, since there is much detail to master before one can be productive. The ROS Wiki has links to documentation and a sequence of tutorials. However, these can be hard to follow for one new to ROS, as the definitions are scattered and the level of detail presented varies broadly from unexplained examples to explanations oriented towards sophisticated users. The intent of this document is to introduce the reader to essential components of ROS with detailed explanations of simple code examples along with the corresponding theory of operation. This introduction only scratches the surface, but it should get the reader started on building useful ROS nodes, as well as bring the reader to a state where the tutorials become more readable.

The ROS Wiki has links to documentation and a sequence of tutorials. However, these can be hard to follow for one new to ROS, as the definitions are scattered and the level of detail presented varies broadly from unexplained examples to explanations oriented towards sophisticated users. The intent of this document is to introduce the reader to essential components of ROS with detailed explanations of simple code examples along with the corresponding theory of operation.

ROS code can be written in either C++ or Python. This text uses C++ exclusively. For Python, the reader is referred to "Programming Robots with ROS: A Practical Introduction to the Robot Operating System" [**?**].

For the present examples, I am using the following versions: Linux Ubuntu 14.04 and ROS Indigo. These are the recommended versions for the "Baxter" robot simulator used as part of this presentation. To try out the examples presented herein, install ROS as described here: `http://wiki.ros.org/indigo/Installation/Ubuntu`. It may be helpful to refer to the book "A Gentle Introduction to ROS" [**?**] to further illuminate and assist with ROS installation. This book ([**?**]) is also recommended as supplementary reading for those desiring greater detail and behind-the-scenes explanations of ROS organization and communications. Additional books on ROS are listed at: `http://wiki.ros.org/Books`.

When installing ROS, one has the option to name a ROS "workspace." In this text, it will be assumed that this directory resides under the user's home directory and is named `ros_ws`. If you choose another name for your ROS workspace, then substitute that name for `ros_ws` in all places referred to herein by `ros_ws`.

Code examples referred to herein may be found at: `https://github.com/wsnewman/learning_ros`. Some additional packages used with this code are contained in `https://github.com/wsnewman/learning_ros_external_packages`. Both repositories should be cloned into the user's ROS workspace in the subdirectory `~/ros_ws/src` to be able to compile the examples. Specifically, after setting up your ROS environment, `cd` to `~/ros_ws/src` and enter:

```
git clone https://github.com/wsnewman/learning_ros.git
```

and

```
git clone https://github.com/wsnewman/learning_ros_external_packages.git
```

All of the example code referred to herein will then be present within these two subdirectories.

This text is not intended to be exhaustive. The interested student, researcher, automation engineer or robot enthusiast will find thousands of ROS packages to explore. Further, on-line tutorials describe details and extensions of the basics presented here. Rather, the intent of this text is to provide an introduction that will enable one to understand the organization of ROS, how to understand ROS packages, how to use ROS tools, how to incorporate existing ROS packages into new applications, and how to develop new packages for robotics and automation. It is also the intent to facilitate continuing education by preparing the reader to better understand the existing on-line documentation.

This text is organized in five parts:

1. ROS Foundations

2. Simulation and Visualization with ROS

3. Sensors and Perceptual Processing in ROS

4. ROS and Mobile Robots

5. Robot Arm Kinematics, Dynamics and Controls

Each of these topics lies within broad fields with considerable specialization and evolving research developments. This text cannot attempt to teach the reader these fields. However, a robot system requires integration of elements spanning hardware integration, human/machine interfacing, control theory, kinematics and dynamics, manipulation planning, motion planning, image processing, scene interpretation, and an array of topics in artificial intelligence. The robot engineer must be a generalist and thus needs an understanding of at least rudimentary practice in each of these areas. One hope of the ROS ecosystem is that one can import existing packages that contribute in each of these areas and integrate them into a custom systemwithout having to be an expert in every field. Thus understanding ROS interfacing and ROS approaches to each of these areas offers value to the system integrator, who can ideally leverage the expertise contributed by robotics researchers, computer scientists, and software engineers around the world.

I

ROS Foundations

CHAPTER 1

Introduction to ROS: ROS tools and nodes

CONTENTS

INTRODUCTION
This introductory chapter will focus on the concepts "nodes" in ROS, starting with minimal examples. A few ROS tools will be introduced as well to help illuminate the behavior of ROS nodes. The simplest means of ROS communications–publish and subscribe–will be used here. Alternative communications means ("services" and "action servers") will be deferred until Chapter 2.

1.1 SOME ROS CONCEPTS

Communications among nodes is at the heart of ROS. A "node" is a ROS program that uses ROSs middleware for communications. A node can be "launched" independent of other nodes and in any order among launches of other nodes. Many nodes can run on the same computer, or nodes may be distributed across a network of computers. A node is only useful if it can communicate with other nodesand ultimately with sensors and actuators.

Communication among nodes uses the concepts of “messages”, “topics”, “roscore”, “publishers” and “subscribers”. (“Services” are also useful, and these are closely related to publishers and subscribers). All communications among nodes is serialized network communications. A “publisher” publishes a “message”, which is a packet of data that is interpretable using an associated key. Since messages are received as a stream of bits, it is necessary to consult the key (the message type description) to know how to parse the bits and populate the corresponding data structure. A simple example of a message is Float64, which is defined in the “std˙msgs” that comes with ROS. The message type helps the publisher pack a floating-point number into the defined stream of 64 bits, and it also helps the subscriber interpret how to unpack the bitstream as a representation of a floating-point number.

A more complex example of a message is a “twist”, which consists of multiple fields describing 3-D translational and rotational velocities. Some messages also accommodate optional extras, such as time stamps and message identifier codes.

When data gets published by a publisher node, it is made available to any interested subscriber nodes. Subscriber nodes must be able to make connections to the published data. Often, the published data originates from different nodeswhether these publishers have changed due to software evolution or whether some publisher nodes are relevant in some contexts and other nodes in other contexts. For example, a publisher node responsible for commanding joint velocities might sometimes be a stiff position controller but in other scenarios a compliant-motion controller may be needed. This hand-off can occur by changing which node is publishing the velocity commands. This presents the problem that the subscriber does not know who is publishing its input. In fact, the need to know what node is publishing complicates construction of large systems. This problem is addressed by the concept of a “topic.”

A topic may be introduced and various publishers may take turns publishing to that topic. Thus a subscriber only needs to know the name of a topic and does not need to know what node (or nodes) publish(es) to that topic. For example, the topic for commanding velocities may be “vel˙cmd”, and the robots low-level controller should subscribe to this named topic to receive velocity commands. Different publishers might be responsible for publishing velocity-command messages on this topic, whether these are nodes under experimentation or trusted nodes that are swapped in to address specific task needs.

Although creating the abstraction of a “topic” helps some, a publisher and a subscriber both need to know how to communicate via a topic. This is accomplished through communications middleware in ROS via the executable “roscore.” Roscore is responsible for coordinating communications, like an operator. Although there can be many ROS nodes distributed across multiple networked computers, there can be only one instance of “roscore” running, and the machine on which roscore runs establishes the “master” computer of the system.

A publisher node initiates a topic by informing roscore of the topic (and the corresponding message type). This is called “advertising” the topic. To accomplish this, the publisher instantiates an object of the class “ros::Publisher”. This class definition is part of the ROS distribution, and using publisher objects allows the designer to avoid having to write communications code. After instantiating a publisher object, the user code invokes the member function “advertise” and specifies the message type and declares the desired topic name. At this point, the user code can start sending out messages to the named topic using the publisher member function “publish”, which takes as an argument the message to be published.

Since a publisher node communicates with roscore, roscore must be running before the any ROS node is launched. To run roscore, open a terminal in Linux and enter “roscore.” The response to this command will be a confirmation “started core services.” It will also print out the ROS˙MASTER˙URI, which is useful for informing nodes running on nonmaster computers how to reach roscore. The terminal running roscore will be dedicated to roscore, and it will be unavailable for other tasks. Roscore should continue running as long as the robot is actively controlled (or as long as it is desired to access the robots sensors).

After roscore has been launched, a publisher node may be launched. The publisher node will advertise its topic and may start sending out messages (at any rate convenient to the node, though at a frequency limited by system capacity). Publishing messages at 1kHz rate is normal for low-level controls and sensor data.

Introducing a sensor to a ROS system requires specialized code (and possibly specialized hardware) that can communicate with the sensor. For example, a LIDAR sensor may require RS488 communications, accept commands in a specific format, and start streaming data in a predefined format. A dedicated microcontroller (or a node within the main computer) must communicate with the LIDAR, receive the data, then publish the data with a ROS message type on a ROS topic. Such specialized nodes convert the specifics of individual sensors into the generic communications format of ROS.

When a publisher begins publishing, it is not necessary that any nodes are listening to the messages. Alternatively, there may be many subscribers to the same topic. The publisher does not need to be aware of whether it has any subscribers nor how many subscribers there may be. This is handled by the ROS middleware. A subscriber may be receiving messages from a publisher, and the publisher node may be halted and possibly replaced with an alternative publisher of the same topic, and the subscriber will resume receiving messages with no need to restart the subscriber.

A ROS subscriber also communicates with roscore. To do so, it uses an object of class ros::Subscriber. This class has a member function called "subscribe" that requires an argument of the named topic. The programmer must be aware that a topic of interest exists and what is the name of the topic. Additionally, the subscriber function requires the name of a "callback" function. This provides the necessary hook to the ROS middleware, such that the callback function will start receiving messages. The callback function suspends until a new message has been published, and the designer may include code to operate on the newly-received message.

Subscriber functions can be launched before the corresponding publisher functions. ROS will allow the subscriber to register its desire to receive messages from a named topic, even though that topic does not exist. At some point, if/when a publisher informs roscore that it will publish to that named topic, the subscribers request will be honored, and the subscriber will start receiving the published messages. A node can be both a subscriber and a publisher. For example, a control node would need to receive sensor signals as a subscriber and send out control commands as a publisher. This only requires instantiating both a subscriber object and a publisher object within the node. It is also useful to pipeline messages for sequential processing. For example, as low-level image processing routine (e.g. for edge finding) could subscribe to raw camera data and publish low-level processed images. A higher-level node might subscribe to the edge-processed images, look for specific shapes within those images, and publish its results (e.g., identified shapes) for further use by still higher-level processes. A sequence of nodes performing successive levels of processing can be modified incrementally by replacing one node at a time. To replace one such link in a chain, the new node only needs to continue to use the same input and output topic names and message types. Although the implementation of algorithms within the modified node may be dramatically different, the other nodes within the chain will be unaffected.

The flexibility to launch publisher and subscriber nodes in any order eases system design. Additionally, individual nodes may be halted at any time and additional nodes may be "hot swapped" into the running system. This can be exploited, for example, to launch some specialized code when it is needed and then halt the (potentially expensive) computations when no longer needed. Additionally, diagnostic nodes (e.g. interpreting and reporting on published messages) may be run and terminated ad hoc. This can be useful e.g. to examine the output of selected sensors to confirm proper functioning.

It should be appreciated that the flexibility of launching and halting publishers and subscribers at any time within a running system can also be a liability. For time-critical codeparticularly control code that depends on sensor values to generate commands to actuatorsan interruption of the control code or of the critical sensor publishers could result in physical damage to the robot or its surroundings. It is up to the programmer to make sure that time-critical nodes remain viable. Disruptions of critical nodes should be tested and responded to appropriately (e.g. with halting all actuators).

From the system architecture point of view, ROS helps one implement a desired software architecture and supports teamwork in building large systems. Starting from a predetermined software architecture, one can construct a "skeleton" of a large system constructed as a collection of nodes. Initially, each of the nodes might be "dummy" nodes, capable of sending and receiving messages via predefined topics (the software interfaces). Each module in the architecture could then be upgraded incrementally by swapping out an older (or dummy) node for a newer replacementand no changes would be required elsewhere throughout the system. This supports distributed software development and incremental testing, which are essential for building large, complex systems.

1.2 WRITING ROS NODES

In this section, design of a minimal publisher node and a minimal subscriber node will be detailed. The concept of a ROS "package" is introduced, along with instructions on how to compile and link the code via the associated files "package.xml" and "CMakeLists." Several ROS tools and commands are introduced, including: `rosrun`, `rostopic`, `rosnode`, and `rqt_graph`. Specific C++ code examples for a publisher and a subscriber are detailed, and results of running the compiled nodes are shown.

The code examples used in this section are contained in the accompanying code repository, within the directory package `minimal_nodes` under `learning_ros/Part_1/minimal_nodes`. This introduction begins with instructions on how the example code was created. In following these instructions it is important to avoid ambiguity from naming conflicts. In this section, the names used will be altered (as necessary) from the provided code examples for the purpose of illustrating to how the example code was created. In subsequent sections, the example code may be used verbatim.

Before creating new ROS code, one must establish a directory (a ROS workspace) where ROS code will reside. One creates this directory somewhere convenient in the system (e.g., directly under home). A subdirectory called src must exist, and this is where source code (packages) will reside. The operating system must be informed of the location of your ROS workspace (typically done automatically through edits to the start-up script .bashrc). Setting up a ROS workspace (and automating defining ROS paths) only needs to be done once. The process is described here: `http://wiki.ros.org/ROS/Tutorials/InstallingandConfiguringROSEnvironment`. It is important that the necessary environment variables be set in Linux, or the OS will not know where to find your code for compilation and execution. Formerly (ROS Fuerte and older), ROS used its own build system called rosbuild. This has been replaced by the catkin build system, which is faster, but can be more complex. A useful simplification of the catkin system is `catkin_simple`, which reduced the detail required of the programmer to specify how to build a project.

For the following, we assume that you already have ROS Indigo installed, that you have a ROS workspace defined (called ros˙ws in the examples to follow), that it has a src subdirectory, and that the OS has been informed of the path to this workspace (via environment variables). We proceed with creating new code within this workspace.

1.2.1 Creating a ROS package

The first thing to do when starting to design new ROS code is to create a package. A package is a ROS concept of bundling multiple, necessary components together to ease building ROS code and coupling it with other ROS code. Packages should be logical groups of code, e.g. separate packages for low-level joint control, for high-level planning, for navigation, for mapping, for vision, etc. Although these packages normally would be developed separately, nodes from separate packages would ultimately be run together simultaneously on a robot (or on a network of computers collectively controlling the robot).

One creates a new "package" using the `catkin_create_pkg` command (a command that is part of the ROS installation). For a given package, this only needs to be done once. You can go back to this package and add more code incrementally (without needing to create the package again). However, the code added to a package should logically "belong" to that package (e.g., avoid putting low-level joint- control code in the same package as mapping code).

New packages should reside under the `src` directory of your catkin workspace (e.g. `ros_ws/src`). As a specific example, consider creation of a new package called `my_minimal_nodes`, which will contain source code in C++ and which will depend on the basic, pre-defined message types contained in std_msgs. To do so, open a terminal and navigate (cd) to the ros_ws directory. A shortcut for this is roscd, which will bring you to ~/ros_ws. From here, move to the subdirectory /src and enter the following command:

```
catkin_create_pkg   my_minimal_nodes   roscpp  std_msgs
```

The effect of this command is to create and populate a new directory: ~/ros_ws/src/my_minimal_nodes.

Every package name in ROS must be unique. By convention, package names should follow common C variable naming conventions: lower case, start with a letter, use underscore separators, e.g. grasp_planner. (See http://wiki.ros.org/ROS/Patterns/Conventions). Every package used within your system must be uniquely named. As noted at http://wiki.ros.org/ROS/Patterns/Conventions, you can check if a name is already taken via the listing here: http://www.ros.org/browse/list.php. For the present illustration, the name my_minimal_nodes was chosen so as not to conflict with the package minimal_nodes, which resides in the example code repository (under ~/ros_ws/src/learning_ros/Part_1/minimal_nodes).

If you move to the newly-created package directory, ~/ros_ws/src/my_minimal_nodes, you will see that it is already populated with package.xml, CMakeLists.txt and the subdirectories src and include. The catkin_create_pkg command has just created a new "package" by the name of my_minimal_nodes, which will reside in a directory of this name.

As we create new code, we will depend on some ROS tools and definitions. Two dependencies were listed during the catkin_create_pkg command: roscpp and std_msgs. The roscpp dependency establishes that we will be using a C++ compiler to create our ROS code, and we will need C++ compatible interfaces (such as the classes ros::Publisher and ros::Subscriber, referred to earlier). The std_msgs dependency says that we will need to rely on some datatype definitions (standard messages) that have been predefined in ROS. (an example is std_msgs::Float64).

The package.xml file: A ROS package is recognized by the build system by virtue of the fact that it has a package.xml file. A compatible package.xml file has a specific structure that names the package and lists its dependencies. For the new package my_minimal_nodes, a package.xml file was auto-generated, and its contents in Listing 1.1.

Listing 1.1: Contents of package.xml for minimal nodes package

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

  <!-- One maintainer tag required, multiple allowed, one person per tag -->
  <!-- Example:  -->
  <!-- <maintainer email="jane.doe@example.com">Jane Doe</maintainer> -->
  <maintainer email="wyatt@todo.todo">wyatt</maintainer>


  <!-- One license tag required, multiple allowed, one license per tag -->
  <!-- Commonly used license strings: -->
  <!--   BSD, MIT, Boost Software License, GPLv2, GPLv3, LGPLv2.1, LGPLv3 -->
  <license>TODO</license>


  <!-- Url tags are optional, but mutiple are allowed, one per tag -->
  <!-- Optional attribute type can be: website, bugtracker, or repository -->
  <!-- Example: -->
  <!-- <url type="website">http://wiki.ros.org/my_miminal_nodes</url> -->
```

```
  <!-- Author tags are optional, mutiple are allowed, one per tag -->
  <!-- Authors do not have to be maintianers, but could be -->
  <!-- Example: -->
  <!-- <author email="jane.doe@example.com">Jane Doe</author> -->


  <!-- The *_depend tags are used to specify dependencies -->
  <!-- Dependencies can be catkin packages or system dependencies -->
  <!-- Examples: -->
  <!-- Use build_depend for packages you need at compile time: -->
  <!--   <build_depend>message_generation</build_depend> -->
  <!-- Use buildtool_depend for build tool packages: -->
  <!--   <buildtool_depend>catkin</buildtool_depend> -->
  <!-- Use run_depend for packages you need at runtime: -->
  <!--   <run_depend>message_runtime</run_depend> -->
  <!-- Use test_depend for packages you need only for testing: -->
  <!--   <test_depend>gtest</test_depend> -->
  <buildtool_depend>catkin</buildtool_depend>
  <build_depend>roscpp</build_depend>
  <build_depend>std_msgs</build_depend>
  <run_depend>roscpp</run_depend>
  <run_depend>std_msgs</run_depend>


  <!-- The export tag contains other, unspecified, tags -->
  <export>
    <!-- Other tools can request additional information be placed here -->

  </export>
</package>
```

The package.xml file is merely ASCII text using XML formatting, and thus you can open it with any editor. (gedit will do). In the listing 1.1, most of the lines are merely comments, such as: <!-- Example: --> , where each comment is delimited by an opening of <!-- and closing of -->. The comments instruct you how to edit this file appropriately. It is recommended that you edit the values to enter your name and e-mail address as author of the code, particularly if you intend to share your contribution publicly.

The line <name>my_miminal_nodes</name> corresponds to the name of our new package. It is important that this name correspond to the name of your package. You cannot merely create a new directory (with a new name) and copy over the contents of another package. Because of the mis-match between your directory name and the package name in the package.xml file, ROS will be confused.

Within the package.xml file, the lines:

```
  <build_depend>roscpp</build_depend>
  <build_depend>std_msgs</build_depend>
  <run_depend>roscpp</run_depend>
  <run_depend>std_msgs</run_depend>
```

explicitly declare dependency on the package roscpp and on the package std_msgs, both of which were explicitly listed as dependencies upon creation of this package. Eventually, we will want to bring in large bodies of third-party code (other "packages"). In order to integrate with these packages (e.g., utilize objects and datatypes defined in these packages), we will want to add them to the package.xml file. This can be done by editing our package's package.xml file and adding build_depend and run_depend lines naming the new packages to be utilized, emulating the existing lines that declare dependence on roscpp and std_msgs.

The src directory is where we will put our user-written C++ code. We will write illustrative nodes minimal_publisher.cpp and minimal_subscriber.cpp as examples. It will be necessary to edit the CMakeLists.txt file to inform the compiler that we have new nodes to be compiled. This will be described further later.

1.2.2 Writing a minimal ROS publisher

In a terminal window, move to the `src` directory within the `my_minimal_nodes` package that has been created. Open up an editor, create a file called `minimal_publisher.cpp` and enter the following code: (note: if you attempt to copy/paste from an electronic version of this text, you likely will get copying errors, including undesired newline symbols, which will confuse the C++ compiler. Rather, refer to the corresponding example code on the associated github repository at `https://github.com/wsnewman/learning_ros`, under package `~/ros_ws/src/learning_ros/Part_1/minimal_nodes`).

Listing 1.2: Minimal Publisher

```
1  #include <ros/ros.h>
2  #include <std_msgs/Float64.h>
3
4  int main(int argc, char **argv) {
5      ros::init(argc, argv, "minimal_publisher"); // name of this node will be "↩
           minimal_publisher"
6      ros::NodeHandle n; // two lines to create a publisher object that can talk to ROS
7      ros::Publisher my_publisher_object = n.advertise<std_msgs::Float64>("topic1", 1);
8      //"topic1" is the name of the topic to which we will publish
9      // the "1" argument says to use a buffer size of 1; could make larger, if expect ↩
           network backups
10
11     std_msgs::Float64 input_float; //create a variable of type "Float64",
12     // as defined in: /opt/ros/indigo/share/std_msgs
13     // any message published on a ROS topic must have a pre-defined format,
14     // so subscribers know how to interpret the serialized data transmission
15
16     input_float.data = 0.0;
17
18     // do work here in infinite loop (desired for this example), but terminate if ↩
           detect ROS has faulted
19     while (ros::ok())
20     {
21         // this loop has no sleep timer, and thus it will consume excessive CPU time
22         // expect one core to be 100% dedicated (wastefully) to this small task
23         input_float.data = input_float.data + 0.001; //increment by 0.001 each ↩
               iteration
24         my_publisher_object.publish(input_float); // publish the value--of type ↩
               Float64--
25         //to the topic "topic1"
26     }
27 }
```

The above code is dissected here. On line 1,

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

is needed to bring in the header files for the core ROS libraries. This should be the first line of any ROS source code written in C++.

Line 2,

```
#include <std_msgs/Float64.h>
```

brings in a header file that describes objects of type: `std_msgs::Float64`, which is a message type we will use in this example code.

As you incorporate use of more ROS message types or ROS libraries, you will need to include their associated header files in your code, just as we have done with `std_msgs`.

Line 4,

```
int main ( int argc , char ** argv )
```

declares a "main" function. For all ROS nodes in C++, there must be one and only one `main()` function per node. Our `minimal_publisher.cpp` file has `main()` declared in the standard "C" fashion with generic argc, argv arguments. This gives the node the opportunity to use command-line options, which are used by ROS functions (and thus these arguments should always be included in `main()`).

The code in lines 5-7 all refer to functions or objects defined in the core ROS library.

Line 5:

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

is needed in every ROS node. The argument `"minimal_publisher"` will be the name that the new node will use to register itself with the ROS system upon start-up. (This name can be overridden, or "remapped," upon launch, but this detail is deferred for now). The node name is required, and every node in the system must have a unique name. ROS tools are provided that take advantage of the node names, e.g. to monitor which nodes are active and which nodes are publishing or subscribing to which topics.

Line 6 instantiates a ROS node handle object with the declaration:

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

A node handle is required for establishing network communications among nodes. The nodehandle name "n" is arbitrary. It will be used infrequently (typically, for initializing communications objects). This line can simply be included in every node's source code (and no harm done in the rare instances in which it is not needed).

Line 7:

```
ros::Publisher my_publisher_object = n.advertise<std_msgs::Float64>("topic1", 1);
```

instantiates an object that is chosen here to be called `my_publisher_object` (the name is the programmers choice). In instantiating this object, the ROS system is informed that the current node (here called `minimal_publisher`) intends to publish messages of type `std_msgs::Float64` on a topic named `topic1`. In practice, one should choose topic names that are helpful and descriptive of the type of information carried via that topic.

On line 11:

```
std_msgs::Float64 input_float;
```

the program then creates an object of type `std_msgs:Float64` and calls it `input_float`. One must consult the message definition in `std_msgs` to understand how to use this object. The object is defined as having a member called `data`. Details of this message type can be found by looking in the corresponding directory with: `roscd std_msgs`. The subdirectory `msg` contains various files defining the structure of numerous standard messages, including `Float64.msg`. Alternatively, one can examine the details of any message type with the command `rosmsg show` ..., e.g. from a terminal, entering the command:

```
rosmsg show std_msgs/Float64
```

will display the fields of this message, which results in the response:

```
float64 data
```

In this case, there is only a single field, named "`data`," which holds a value of type "`float64`" (a ROS primitive).

On line 16,

```
input_float.data = 0.0;
```

the program initializes the "data" member of `input_float` to the value 0.0. An infinite loop is then entered, though this loop will self terminate upon detecting that the ROS system has terminated, which is accomplished using the function `ros::ok()` in line 19:

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

This approach can be convenient for shutting down a collection of nodes by merely halting the ROS system (i.e., by killing "roscore").

Inside the "while" loop, the value of `input_float.data` is incremented by 0.001 each iteration. This value is then published (line 24) using:

```
my_publisher_object.publish(input_float);
```

It was previously established (upon instantiation of the object `my_publisher_object` from the class `ros::Publisher`) that the object `my_publisher_object` would publish messages of type `std_msg::Float64` to the topic called `topic1`. The publisher object, `my_publisher_object` has a member function "`publish`" to invoke publications. The publisher expects an argument of compatible type. Since the object `input_float` is of type `std_msgs::Float64`, and since the publisher object was instantiated to expect arguments of this type, the "publish" call is consistent.

The example ROS node has only 14 active lines of code. Much of this code is ROS-specific and may seem cryptic. However, most of the lines are common "boilerplate," and becoming familiar with these common lines will make other ROS code easier to read.

1.2.3 Compiling a ROS node

ROS nodes are compiled by running `catkin_make`. This command must be executed from a specific directory. In a terminal, navigate to your ros workspace (`~/ros_ws`). Then enter: `catkin_make`.

This will "make" (compile) all packages in your workspace. Compiling large collections of code can be time consuming, but compilation will be faster on subsequent edit/compile/test iterations. Although compilation is sometimes slow, the compiled code can be very efficient. Particularly for CPU-intensive operations (e.g. for point-cloud processing, image processing or intensive planning computations), compiled C++ code typically runs faster than Python code.

After building a catkin package, the executables will reside in a folder in `ros_ws/devel/lib` named according to the source package.

Before we can compile, however, we have to inform `catkin_make` of the existence of our new source code, `minimal_publisher.cpp`. To do so, edit the file `CMakeLists.txt`, which was created for us in the package `my_minimal_nodes` when we ran `catkin_create_pkg`. This file is quite long (187 lines, for our example), but consists almost entirely of comments.

The comments describe how to modify `CmakeLists.txt` for numerous variations. For the present, we only need to make sure we have our package dependencies declared, inform the compiler to compile our new source code, and link our compiled code with any necessary libraries.

catkin_package_create already fills in the fields:

Listing 1.3: Snippet from CMakeLists.txt

```
find_package(catkin REQUIRED COMPONENTS
  roscpp
  std_msgs
)

include_directories(
  ${catkin_INCLUDE_DIRS}
)
```

However, we need to make two modifications, as follows:

Listing 1.4: Adding the new node and linking it with libraries

```
## Declare a cpp executable
add_executable(my_minimal_publisher src/minimal_publisher.cpp)

## Specify libraries to link a library or executable target against
target_link_libraries(my_minimal_publisher ${catkin_LIBRARIES} )
```

These modifications inform the compiler of our new source code, as well as which libraries to link with.

In the above, the first argument to `add_executable` is a chosen name for the executable to be created (I chose to call this `my_minimal_publisher`, which happens to be the same root name as the source code).

The second argument is where to find the source code, relative to the package directory. Our source code is in the `src` subdirectory and it is called `minimal_publisher.cpp`. (It is typical for the source code to reside in the `src` sub-directory of a package).

There are a few idiosyncracies regarding the node name. In general, one cannot run two nodes with the same name. The ROS system will complain, will kill the currently running node, and will start the new (identically-named) node. ROS does allow for different packages to re-use node names–although only one of these at a time may be run. Although (executable) node names are allowed to be duplicated across packages, the `catkin_make` build system gets confused by such duplication (though the build can be forced by compiling packages one at a time). For simplicity, it is best to avoid replicating node names.

Having edited the `CMakeLists.txt` file, we can compile our new code. To do so, from a terminal, navigate to the `ros_ws` directory and enter:

```
catkin_make
```

This will invoke the C++ compiler to build all packages, including our new `my_minimal_nodes` package. If the compiler output complains, find and fix your bugs.

Assuming compilation success, if you look in the directory `ros_ws/devel/lib/my_minimal_nodes`, you will see a new, executable file there named `my_minimal_publisher`. This is the name that was chosen for the output file (executable node) with the addition of line `add_executable(my_minimal_publisher src/minimal_publisher.cpp)` in `CMakeLists.txt`.

1.2.4 Running a ROS node

As noted in Section 1.1, there must be one and only one instance of `roscore` running before any nodes can be started. In a terminal, enter:

```
roscore
```

This should respond with a page of text concluding with "started core services." You can then shrink this window and leave it alone. ROS nodes can be started and stopped at random without needing to start a new roscore. (If you kill "roscore" or the window running roscore, however, all nodes will stop running).

Next, start the new publisher node by entering (from a new terminal):

```
rosrun my_minimal_nodes my_minimal_publisher
```

The arguments to the command `rosrun` are the package name (`my_minimal_nodes`) and the executable name (`my_minimal_publisher`).

The `rosrun` command can seem confusing at times due to reuse of names. For example, if we wanted to make a LIDAR publisher node, we might have a package called `lidar_publisher`, a source file called `lidar_publisher.cpp`, an executable called `lidar_publisher`, and a node name (declared within the source code) of `lidar_publisher`. To run this node, we would type:

```
rosrun lidar_publisher lidar_publisher
```

This may seem redundant, but it still follows the format:

```
rosrun package_name executable_name
```

Once the command has been entered, the ROS system will recognize a new node by the name of `lidar_publisher`. This name reuse may seem to lead to confusion, but in many instances, there is no need to invent new names for the package, the source code, the executable name and the node name. In fact, this can help simplify recognizing named entities–as long as the context is clear (package, source code, executable, node name).

1.2.5 Examining the running minimal publisher node

After entering: `rosrun my_minimal_nodes my_minimal_publisher`, the result may seem disappointing. The window that launched this node seems to hang and provides no feedback to the user. Still, `minimal_publisher` is running. To see this, we can invoke some ROS tools.

Open up a new terminal and enter: `rostopic`. You will get the following response:

```
rostopic is a command-line tool for printing information about ROS Topics.

Commands:
rostopic bw display bandwidth used by topic
rostopic echo print messages to screen
rostopic find find topics by type
rostopic hz display publishing rate of topic
rostopic info print information about active topic
rostopic list list active topics
rostopic pub publish data to topic
rostopic type print topic type
```

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

This shows that the command `rostopic` has 8 options. If we type: `rostopic list`, the result is:

```
/rosout
/rosout_agg
/topic1
```

We see that there are three active topics–two that ROS created on its own and the topic created by our publisher, `topic1`.

Entering:

```
rostopic hz topic1
```

results in the following output:

```
average rate: 38299.882
min: 0.000s max: 0.021s std dev: 0.00015s window: 50000
average rate: 38104.090
min: 0.000s max: 0.024s std dev: 0.00016s window: 50000
```

This output shows that our minimal publisher (on this particular computer) is publishing its data at roughly 38kHz (with some jitter). Viewing the system monitor would show that one CPU core is fully saturated just running the minimal publisher. This is because the while-loop within our ROS node has no pauses in it. It is publishing as fast as it can.

Entering:

```
rostopic bw topic1
```

yields the following output:

```
average: 833.24KB/s
mean: 0.01KB min: 0.01KB max: 0.01KB window: 100
average: 1.21MB/s
mean: 0.00MB min: 0.00MB max: 0.00MB window: 100
average: 746.32KB/s
mean: 0.01KB min: 0.01KB max: 0.01KB window: 100
```

This display shows how much of our available communications bandwidth is being consumed by our minimal publisher (nominally 1 MB/s). This rostopic option can be useful for identifying nodes that are over-consuming communications resources.

Entering:

```
rostopic info topic1
```

yields:

```
Type: std_msgs/Float64

Publishers:
 * /minimal_publisher (http://Wall-E:56763/)

Subscribers: None
```

This tells us that the topic called `topic1` involves messages of type `std_msgs/Float64`. At present, there is a single publisher to this topic (which is the norm), and that publisher has a node name of `minimal_publisher`. As noted above, this is the name we assigned to the node within the source code on line 5:

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

Entering: `rostopic echo topic1` causes rostopic to try to print out everything published on `topic1`. A sample of the output is:

```
data: 860619.606909
---
data: 860619.608909
---
data: 860619.609909
---
data: 860619.612909
---
```

In this case, the display has no hope of keeping up with the publishing rate, and most of the messages are dropped between lines of display. If the echo could keep up, we would expect to see values that increase by increments of 0.001, which is the increment used in the while-loop of our source code.

The `rostopic` command tells us much about the status of our running system, even though there are no nodes receiving the messages sent out by our minimal publisher. Additional handy ROS commands are summarized in the "ROS cheat sheet" (see: `http://www.ros.org/news/2015/05/ros-cheatsheet-updated-for-indigo-igloo.html`)

For example, entering:

```
rosnode list
```

results in the following output:

```
/minimal_publisher
/rosout
```

We see that there are two nodes running: `rosout` (a generic process used for nodes to display text to a terminal, launched by default by roscore) and `minimal_publisher` (our node).

Although `minimal_publisher` does not take advantage of the capability of displaying output to its terminal, the link is nonetheless available through the topic `rosout`, which would get processed by the display node `rosout`. Using `rosout` can be helpful, since one's code does not get slowed down by output (e.g. cout) operations. Rather, messages get sent rapidly by publishing the output to the `rosout` topic, and a separate node (`rosout`) is responsible for user display. This can be important, e.g., in time-critical code where some monitoring is desired, but not at the expense of slowing down the time-critical node.

1.2.6 Scheduling node timing

We have seen that our example publisher is abusive of both CPU capacity and communications bandwidth. In fact, it would be unusual for a node within a robotic system to require updates at 30kHz. A more reasonable update rate for even time-critical, low-level nodes is 1kHz. In the present example, we will slow our publisher down to 1Hz using a ROS timer. A modified version of source code for `minimal_publisher.cpp`, called `sleepy_minimal_publisher.cpp`, is shown below:

Listing 1.5: Minimal Publisher with Timing

```
1 #include <ros/ros.h>
2 #include <std_msgs/Float64.h>
3
4 int main(int argc, char **argv) {
5     ros::init(argc, argv, "minimal_publisher2"); // name of this node will be "minimal_publisher2"
6     ros::NodeHandle n; // two lines to create a publisher object that can talk to ROS
7     ros::Publisher my_publisher_object = n.advertise<std_msgs::Float64>("topic1", 1);
8     //"topic1" is the name of the topic to which we will publish
9     // the "1" argument says to use a buffer size of 1; could make larger, if expect network backups
10
11     std_msgs::Float64 input_float; //create a variable of type "Float64",
12     // as defined in: /opt/ros/indigo/share/std_msgs
13     // any message published on a ROS topic must have a pre-defined format,
14     // so subscribers know how to interpret the serialized data transmission
15
16    ros::Rate naptime(1.0); //create a ros object from the ros Rate class;
17    //set the sleep timer for 1Hz repetition rate (arg is in units of Hz)
18
19     input_float.data = 0.0;
20
21     // do work here in infinite loop (desired for this example), but terminate if detect ROS has faulted
22     while (ros::ok())
23     {
24         // this loop has no sleep timer, and thus it will consume excessive CPU time
25         // expect one core to be 100% dedicated (wastefully) to this small task
26         input_float.data = input_float.data + 0.001; //increment by 0.001 each iteration
27         my_publisher_object.publish(input_float); // publish the value--of type Float64--
28         //to the topic "topic1"
29     //the next line will cause the loop to sleep for the balance of the desired period
30         // to achieve the specified loop frequency
31     naptime.sleep();
32     }
33 }
```

There are only two new lines in the above program: line 16

```
ros::Rate naptime(1); //set the sleep timer for 1Hz repetition rate
```

and line 31:

```
naptime.sleep();
```

The ROS class `Rate` is invoked to create a `Rate` object that was named “naptime”. In doing so, `naptime` is initialized with the value “1”, which is a specification of the desired frequency (1Hz). After creating this object, it is used within the while-loop, invoking the member function `sleep()`. This causes the node to suspend (thus ceasing to consume CPU time) until the balance of the desired period (1 second) has expired.

After re-compiling the modified code (with `catkin_make`), we can run it (with `rosrun`) and examine its behavior by entering `rostopic hz topic1` (from a new terminal), which produces the display below:

```
average rate: 1.000
min: 1.000s max: 1.000s std dev: 0.00000s window: 2
average rate: 1.000
min: 1.000s max: 1.000s std dev: 0.00006s window: 3
average rate: 1.000
min: 1.000s max: 1.000s std dev: 0.00005s window: 4
average rate: 1.000
```

This output indicates that the topic `topic1` is being updated at 1Hz with excellent precision and very low jitter. Further, an inspection of the system monitor shows that there is negligible CPU time being consumed by our modified publisher node.

If we enter `rostopic echo topic1` from a terminal, example output looks like the following:

```
data: 0.153
---
data: 0.154
---
data: 0.155
---
```

Each message sent by the publisher is displayed by rostopic echo, as evidenced by the increments of 0.001 between messages. This display is updated once per second, since that is the rate new data is now published.

1.2.7 Writing a minimal ROS subscriber

The complement to the publisher is a subscriber (a listener node). We will create this node in the same package, `my_minimal_nodes`. The source code will go in the subdirectory `src`.

Open an editor and create the file `minimal_subscriber.cpp` in the directory: `~/ros_ws/src/my_minimal_nodes/src`. Enter the following code (which may be found in `~/ros_ws/src/learning_ros/Part_1/minimal_nodes/src/minimal_subscriber.cpp`):

Listing 1.6: Minimal Subscriber

```
#include<ros/ros.h>
#include<std_msgs/Float64.h>
void myCallback(const std_msgs::Float64& message_holder)
{
  // the real work is done in this callback function
  // it wakes up every time a new message is published on "topic1"
  // Since this function is prompted by a message event,
  //it does not consume CPU time polling for new data
  // the ROS_INFO() function is like a printf() function, except
  // it publishes its output to the default rosout topic, which prevents
  // slowing down this function for display calls, and it makes the
  // data available for viewing and logging purposes
  ROS_INFO("received value is: %f",message_holder.data);
  //really could do something interesting here with the received data...but all we do is print it
}

int main(int argc, char **argv)
{
  ros::init(argc,argv,"minimal_subscriber"); //name this node
  // when this compiled code is run, ROS will recognize it as a node called "minimal_subscriber"
  ros::NodeHandle n; // need this to establish communications with our new node
  //create a Subscriber object and have it subscribe to the topic "topic1"
  // the function "myCallback" will wake up whenever a new message is published to topic1
  // the real work is done inside the callback function

  ros::Subscriber my_subscriber_object= n.subscribe("topic1",1,myCallback);

  ros::spin(); //this is essentially a "while(1)" statement, except it
  // forces refreshing wakeups upon new data arrival
  // main program essentially hangs here, but it must stay alive to keep the callback function alive
  return 0; // should never get here, unless roscore dies
}
}
```

Most of the code within the minimal subscriber is identical to that of the minimal publisher (though the node name in line 19 has been changed to "`minimal_subscriber`"). There are 4 important, new lines to examine.

Most notably, the subscriber is more complex than the publisher, since it requires a "callback", which is declared in line 3,

```
void myCallback(const std_msgs::Float64& message_holder)
```

This function has an argument of a reference pointer (indicated by the & sign) to an object of type `std_msgs::Float64`. This is the message type associated with the topic `topic1`, as published by our minimal publisher.

The importance of the callback function is that it is "woken up" when new data is available on its associated topic (which is set to "topic1" in this example). When new data is published to the associated topic, the callback function runs–and the published data appears in the argument "`message_holder`". (This message holder must be of a type compatible with the message type published on the topic of interest, in this case `std_msgs::Float64`).

Within the callback function, the only action taken is to display the received data, implemented on line 13.

```
 ROS_INFO("received value is: %f",message_holder.data);
```

Display is performed using `ROS_INFO()` instead of `cout` or `printf`. Using `ROS_INFO()` uses message publishing, which avoids slowing down time-critical code for display driving. Also, using `ROS_INFO()` makes the data available for logging or monitoring. However, as viewed from the terminal from which this node is run, the output is displayed equivalently to using `cout` or `printf`. The argument to `ROS_INFO()` is the same as `printf` in C.

An alternative to using `ROS_INFO()` is `ROS_INFO_STREAM()`. Line 13 could be replaced with:

```
ROS_INFO_STREAM("received value is: "<<message_holder.data<<std::endl);
```

which produces the same output, but uses syntax of `cout`.

Once the callback function executes, it goes back to sleep, ready to be re-awakened by arrival of a new message on "topic1."

In the main program, a key new concept is on line 26:

```
ros::Subscriber my_subscriber_object= n.subscribe("topic1",1,myCallback);
```

The use of `ros::Subscriber` is similar to the use of `ros::Publisher` earlier. An object is substantiated of type `Subscriber` – a class that exists within the ROS distribution. There are three arguments used in instantiating the subscriber object. The first argument is the topic name. `topic1` is chosen here, which is the topic to which our minimal publisher publishes. (For this example, we want our subscriber node to listen to the output of our example publisher node).

The second argument is the queue size. If the callback function has trouble keeping up with published data, the data may be queued up. In the present case, the queue size is set to one. If the callback function cannot keep up with publications, messages will be lost by being overwritten by newer messages. (Recall that in the first example, `rostopic echo topic1` could not keep up with the 30kHz rate of the original minimal publisher. Values displayed skipped many intermediate messages). For control purposes, typically only the most recent sensor value published is of interest. If a sensor publishes faster than the callback function can respond, there is no harm done in dropping messages, as only the most recent message would be needed. In this (and many cases) a queue size of 1 message is all that is needed.

The third argument for instantiating the Subscriber object is the name of the callback function that is to receive data from `topic1`. This argument has been set to `myCallback`, which is the name of our callback function, described earlier. Through this line of code, we associate our callback function with messages published on topic `topic1`.

Finally, line 28:

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

introduces a key ROS concept that is non-obvious but essential. The callback function should awaken whenever a new message appears on `topic1`. However, the main program must yield some time for the callback function to respond. This is accomplished with a "spin()" command. In the present case, a "spin" causes the main program to suspend, but keeps the callback function alive. If the main program were to run to conclusion, the callback function would no longer be poised to react to new messages. The "spin()" command keeps `main()` alive without consuming CPU time. As a result, the minimal subscriber is quite efficient.

1.2.8 Compiling and running the minimal subscriber

For our new node to get compiled, we must include reference to it in `CMakeLists.txt`. This requires adding two lines, very similar to what we did to enable compiling the minimal publisher. The first new line is simply:

```
add_executable(my_minimal_subscriber src/minimal_subscriber.cpp)
```

The arguments are the desired executable name (chosen to be `my_minimal_subscriber`). The relative path to the source code is the second argument: `src/minimal_subscriber.cpp`.

The second line added is:

```
 target_link_libraries(my_minimal_subscriber  ${catkin_LIBRARIES} )
```

which informs the compiler to link our new executable with the declared libraries.

After updating `CMakeLists.txt`, the code is newly compiled with the command `catkin_make` (which must be run from the `ros_ws` directory).

The code example should compile without error, after which a new executable, `my_minimal_subscriber`, will appear in the directory: `~/ros_ws/devel/lib/my_minimal_nodes`. Note: it is not necessary to recall this lengthy path to the executable file. The executable will be found when running "rosrun" with the package name and node name as arguments.

After recompiling, we now have two nodes to run. In one terminal, enter:

```
rosrun my_minimal_nodes sleepy_minimal_publisher
```

and in a second terminal enter:

```
rosrun my_minimal_nodes my_minimal_subscriber
```

It does not matter which is run first. An example of the display in the terminal of the `my_minimal_subscriber` node is shown below:

```
[ INFO] [1435555572.972403158]: received value is: 0.909000
[ INFO] [1435555573.972261535]: received value is: 0.910000
[ INFO] [1435555574.972258968]: received value is: 0.911000
```

This display was updated once per second, since the publisher published to `topic1` at 1Hz. The messages received differ by increments of 0.001, as programmed in the publisher code.

In another terminal, entering:

```
rosnode list
```

results in the output below.

```
/minimal_publisher2
/minimal_subscriber
/rosout
```

This shows that we now have 3 nodes running: the default `rosout`, our minimal publisher (which we named node `minimal_publisher2` for the timed version) and our minimal subscriber, `minimal_subscriber`.

In an available terminal, entering:

```
rqt_graph
```

produces a graphical display of the running system, which is shown in Fig 1.1.

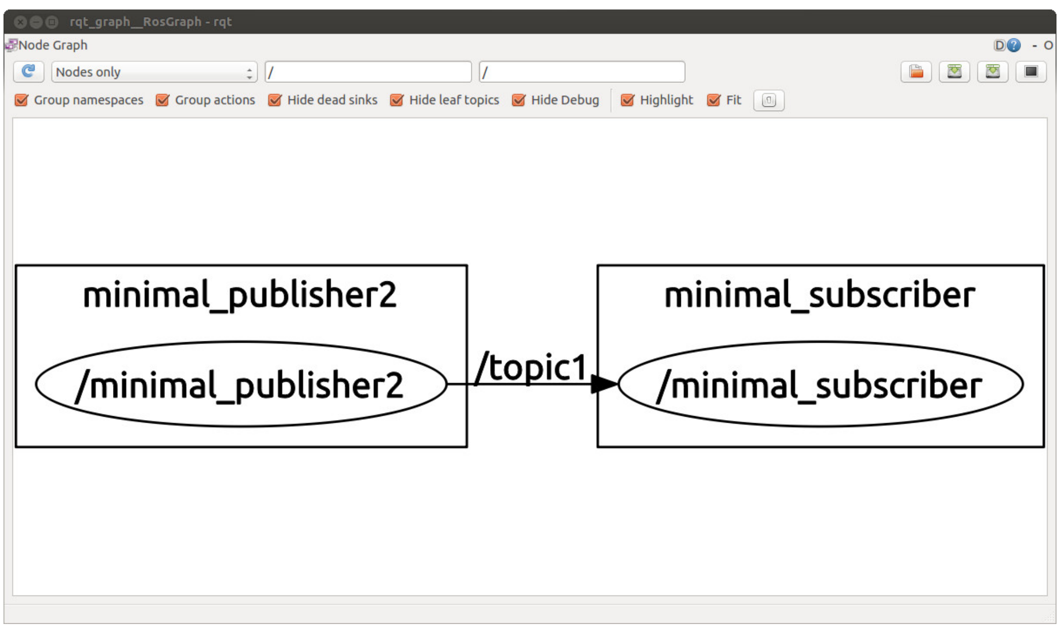


FIGURE 1.1: Node topology, as illustrated by `rqt_graph`

The graphical display shows our 2 nodes. Our minimal publisher node is shown publishing to topic `topic1` and our minimal subscriber is shown subscribing to this topic.

1.2.9 Minimal subscriber and publisher node summary

We have seen the basics for how to create our own publisher and subscriber nodes. A single node can subscribe to multiple topics by replicating the corresponding lines of code to create additional subscriber objects and callback functions. Similarly, a node can publish to multiple topics by instantiating multiple Publisher objects. A node can also be both a subscriber and a publisher.

In a ROS-controlled robot system, custom nodes must be designed that are device drivers that can read and publish sensor information and one or more nodes that subscribe to actuator (or controller setpoint) commands and impose these on actuators. Fortunately, there is already a large body of existing ROS drivers for common sensors, including LIDARs, cameras, the Kinect sensor, inertial measurement units, encoders, etc. These may be imported as packages and used as-is in your system (though perhaps requiring some tweaking to reference your com ports, etc). There are also some packages for driving some servos (i.e. hobby-servo style RC's and Dynamixel motors). There are also some "ROS-Industrial" interfaces to industrial robots, which only require publishing and subscribing to/from robot topics. In some cases, the user may need to design their own device driver nodes to interface to custom actuators. Further, hard real-time, high-speed servo loops may require a non-ROS, dedicated controller (although this might be as simple as an Arduino microcontroller). The user then assumes the responsibility for designing the hard-real-time controller and writing a ROS-compatible subscriber interface to run on the control computer.

1.3 SOME MORE ROS TOOLS: CATKIN_SIMPLE, ROSLAUNCH, RQT_CONSOLE, AND ROSBAG

Having introduced minimal ROS talkers (publishers) and listeners (subscribers), one can already begin to appreciate the value of some additional ROS tools. Some additional tools and facilities are introduced here: `catkin_simple`, `roslaunch`, `rqt_console` and `rosbag`.

1.3.1 Simplifying CMakeLists.txt with catkin_simple

As seen in section 1.2.3, the CMakeLists.txt file generated with a new package creation is quite long. While the required edits to this file were relatively simple, introducing additional features can require tedious, non-obvious changes. A package called "`catkin_simple`" helps to simplify the CMakeLists.txt file. This package can be found at `https://github.com/catkin/catkin_simple.git`. A copy has been cloned into our external-packages repository at `https://github.com/wsnewman/learning_ros_external_packages.git`, which should already be cloned into your `~/ros_ws/src` directory.

Additionally, the external-packages repository has a Python script that assists with creating new packages that use `catkin_simple`. To run this script, it is convenient to define an alias to point to this script as a command. Within a terminal, enter:

```
alias cs_create_pkg='~/ros_ws/src/learning_ros_external_packages/cs_create_pkg.py'
```

Subsequently, from this same terminal, you can create a package that uses `catkin_simple` with the command `cs_create_pkg`. For example, navigate to `~/ros_ws/src` and create a new package called "`my_minimal_nodes2`" by entering:

```
cs_create_pkg my_minimal_nodes2 roscpp std_msgs
```

Note that this command will only be recognized in the terminal from which the alias `cs_create_pkg` was defined. More conveniently, the alias definition should be including in your `.bashrc` file. To do so, navigate to your home directory and edit the (hidden) file `.bashrc`. Append the line

```
alias cs_create_pkg='~/ros_ws/src/learning_ros_external_packages/cs_create_pkg.py'
```

to this file and save it. Subsequently, this alias will be recognized by all new terminals that are opened.

After having invoked our `cs_create_pkg` command, the new package `my_minimal_nodes2` contains the expected structure, including subdirectories `src` and `include`, the `package.xml` file, a `CMakeLists.txt` file and a new file "`README.md`". The README file should be edited to describe the purpose of the new package, as well as a description of how to run examples within the package. The README file uses "markdown" formatting (see `https://guides.github.com/features/mastering-markdown/` for a description of Markdown). Such formatting allows your README file to be displayed with attractive formatting when viewed from your repository via a browser.

The `package.xml` file is similar to that created by `catkin_create_pkg`, except it includes the additional dependency:

```
<buildtool_depend>catkin_simple</buildtool_depend>
```

The `CMakeLists.txt` file is considerably simplified. A copy of the default generated file is:

Listing 1.7: CMakeLists.txt with `catkin_simple`

```
cmake_minimum_required(VERSION 2.8.3)
project(my_minimal_nodes2)

find_package(catkin_simple REQUIRED)

#uncomment next line to use OpenCV library
#find_package(OpenCV REQUIRED)

#uncomment the next line to use the point-cloud library
#find_package(PCL 1.7 REQUIRED)

#uncomment the following 4 lines to use the Eigen library
#find_package(cmake_modules REQUIRED)
#find_package(Eigen3 REQUIRED)
#include_directories(${EIGEN3_INCLUDE_DIR})
#add_definitions(${EIGEN_DEFINITIONS})

catkin_simple()

# example boost usage
# find_package(Boost REQUIRED COMPONENTS system thread)

# C++0x support - not quite the same as final C++11!
# use carefully;  can interfere with point-cloud library
# SET(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++0x")

# Libraries: uncomment the following and edit arguments to create a new library
# cs_add_library(my_lib src/my_lib.cpp)

# Executables: uncomment the following and edit arguments to compile new nodes
# may add more of these lines for more nodes from the same package
# cs_add_executable(example src/example.cpp)

#the following is required, if desire to link a node in this package with a library ↩
    created in this same package
# edit the arguments to reference the named node and named library within this package
# target_link_libraries(example my_lib)

cs_install()
cs_export()
```

The line `catkin_simple()` invokes actions to automatically perform much of the tedious editing of CMakeLists.txt. The commented lines are reminders of how to exploit CMake variations, including linking with popular libraries, such as “Eigen”, “PCL” and “OpenCV”. To modify CMakeLists.txt to compile our desired code, uncomment and edit the line:

```
cs_add_executable(example src/example.cpp)
```

Copying `minimal_publisher.cpp` to the new package, `my_minimal_nodes2`, we can specify that this node should be compiled by modifying the above line in CMakeLists.txt to:

```
cs_add_executable(minimal_publisher3 src/minimal_publisher.cpp)
```

This says that we wish to compile `minimal_publisher.cpp` and call the executable `minimal_publisher3` (a name that will not conflict with other instances of minimal publisher nodes). We can add commands to compile additional nodes by inserting more `cs_add_executable` lines of the same form. It is not necessary to specify linking with libraries. This may not seem to be much of a simplification at this point, but `catkin_simple` will become much more valuable when we start to link with more libraries, create libraries, and create custom messages.

1.3.2 Automating starting multiple nodes

In our minimal example, we ran two nodes: one publisher and one subscriber. To do so, we opened two separate terminals and typed in two `rosrun` commands. Since a complex system may have hundreds of nodes running, we need a more convenient way to bring up a system. This can be done using "launch" files and the command `roslaunch`. (see `http://ros.org/wiki/roslaunch` for more details and additional capabilities, such as setting parameters).

A launch file has the suffix ".launch". It is conventionally named the same as the package name (though this is not required). It is also conventionally located in a subdirectory of the package by the name of `launch` (although this is also not required). A launch file can also invoke other launch files to bring up multiple packages. However, we will start with a minimal launch file. In our package `my_minimal_nodes`, we may create a subdirectory `launch` and within this directory create a launch file `my_minimal_nodes.launch` containing the following lines:

```
<launch>
<node name="publisher" pkg="my_minimal_nodes" type="sleepy_minimal_publisher"/>
<node name="subscriber" pkg="my_minimal_nodes" type="my_minimal_subscriber"/>
</launch>
```

In the above, using XML syntax, we use the keyword "node" to tell ROS that we want to launch a ROS node (an executable program compiled by `catkin_make`). We must specify 3 key/value pairs to launch a node: the package name of the node (value of "pkg"), the binary executable name of the node (value of "type"), and the name by which the node will be recognized by ROS when launched (value of "name"). In fact, we had already specified node names within the source code (e.g. `ros::init(argc,argv, "minimal_publisher2")` within `sleepy_minimal_publisher.cpp`). The launch file gives you the opportunity to rename the node when it is launched. For example, by setting: `name= "publisher"` in the launch file, we would still start running an instance of the executable called `sleepy_minimal_publisher` within the package `my_minimal_nodes`, but it will be known to roscore by the name `publisher`. Similarly, by assiging `name= "subscriber"` to the executable `my_minimal_subscriber`, this node will be known to roscore as `subscriber`.

Assuming we can execute the launch file by typing:

```
roslaunch  my_minimal_nodes  my_minimal_nodes.launch
```

Recall that for using `rosrun` to start nodes, it was necessary that `roscore` be running first. When using `roslaunch`, it is not necessary to start `roscore` running. If `roscore` is already running, the `roslaunch` will launch the specified nodes. If a `roscore` is not already running, `roslaunch` will detect this and will start a `roscore` before launching the specified nodes.

The terminal in which we executed `roslaunch` displays the following:

```
SUMMARY
========

PARAMETERS
 * /rosdistro: indigo
 * /rosversion: 1.11.13

NODES
  /
    publisher (my_minimal_nodes/sleepy_minimal_publisher)
    subscriber (my_minimal_nodes/my_minimal_subscriber)

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

```
core service [/rosout] found
process[publisher-1]: started with pid [18793]
process[subscriber-2]: started with pid [18804]
```

From another terminal, entering:

```
rosnode list
```

produces the output:

```
/publisher
/rosout
/subscriber
```

which shows that nodes known as “publisher” and “subscriber” are running (using the new names assigned to these nodes by the launch file).

Most often, we will not want to change the name of the node from its original specification, but ROS launch files nonetheless require that this “option“ be used. To decline changing the node name, the default name (embedded in the source code) may be used as the desired node name.

After “launching” our nodes, `rostopic list` shows that the `topic1` is alive, and the command `rostopic info topic1` shows that the node `publisher` is publishing to this topic and the node `subscriber` has subscribed to this topic. Clearly, this will be useful when we have many nodes to launch.

One side effect, though, is that we no longer see the output from our subscriber, which formerly appeared in the terminal from which the subscriber was launched. However, since we used `ROS_INFO()` instead of `printf` or `cout`, we can still observe this output using the `rqt_console` tool.

1.3.3 Viewing output in a ROS console

A convenient tool to monitor ROS messages is `rqt_console`, which can be invoked from a terminal by entering: `rqt_console`. With our two nodes running, an example of using this tool is shown in figure 1.2.

In this instance, `rqt_console` shows values output by the minimal subscriber from the time `rqt_console` was started and until `rqt_console` was paused (using the `rqt_console` “pause” button). The lines displayed show that the messages are merely informational–not warnings or errors. The console also shows that the node responsible for posting the information is our minimal subscriber (known by the node name “subscriber”). `rqt_console` also notes the time-stamp at which the message was sent.

Multiple nodes using `ROS_INFO()` may be run simultaneously, and their messages may be viewed with `rqt_console`. `rqt_console` will also note new events, such as starting and stopping new nodes. Another advantage of using `ROS_INFO()` instead of `printf()` or `cout` is that the messages can be logged and run in playback. A facility for doing this is “rosbag.”

1.3.4 Recording and playing back data with `rosbag`

The `rosbag` command is extremely useful for debugging complex systems. One can specify a list of topics to record while the system is running, and `rosbag` will subscribe to these topics and record the messages published, along with timestamps, in a “bag” file. `rosbag` can also be used to play back “bag” files, thus recreating the circumstances of the recorded system.

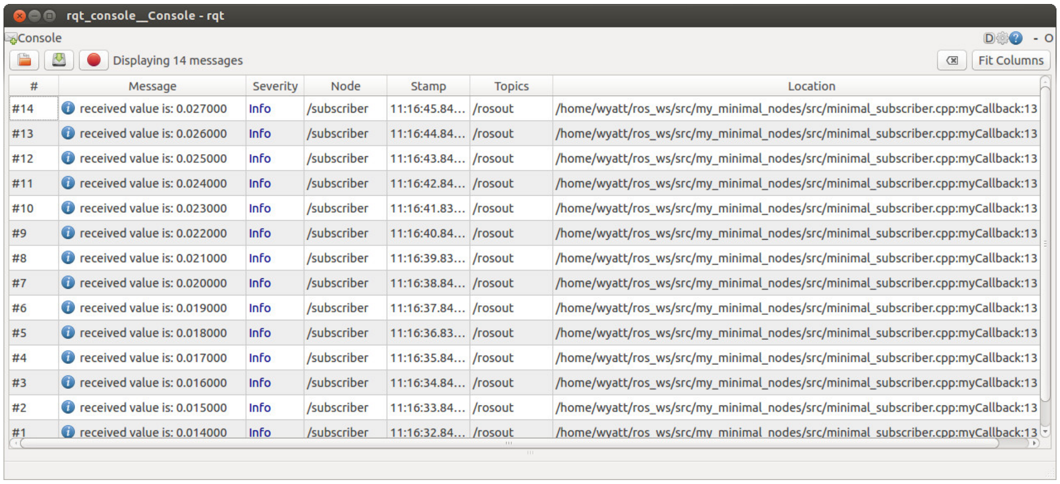


FIGURE 1.2: Output of `rqt_console` with minimal nodes launched

(This playback occurs at the same clock rate at which the original data was published, thus emulating the real-time system).

When running **`rosbag`**, the resulting log (bag) files will be saved in the same directory from which **`rosbag`** was launched. For our example, move to the **`my_minimal_nodes`** directory and create a new subdirectory **`bagfiles`**. With our nodes still running (which is optional; nodes can be started later), in a terminal, navigate to the**`bagfile`** directory (wherever you chose to store your bags) and enter:

```
rosbag record topic1
```

With this command, we have asked to record all messages published on `topic1`. Run `rqt_console`. `rqt_console` displays data from `topic1`, as reported by our subscriber node using `ROS_INFO()`. In the screenshot of `rqt_console` shown in Fig 1.3, the **`rosbag`** recording startup is noted at line number 34; at this instant, the value output from our subscriber node is 0.236 (i.e., 236 seconds after the nodes were launched).

`rosbag` was subsequently halted with a control-C in the terminal from which it was launched. Looking in the **`bagfiles`** directory (from which we launched **`rosbag`**), we see there is a new file, named according to the date/time of the recording, and with the suffix ".bag."

We can play back the recording using **`rosbag`** as well. To do so, first kill the running nodes, then type:

```
    rosbag  play  fname.bag
```

where "fname" is the file name of the recording. The **`rosbag`** terminal shows a playback time incrementing, but there is otherwise no noticeable effect. The screen output is:

```
rosbag play 2016-01-07-11-20-15.bag
[ INFO] [1452184943.589921994]: Opening 2016-01-07-11-20-15.bag

Waiting 0.2 seconds after advertising topics... done.

Hit space to toggle paused, or 's' to step.
```

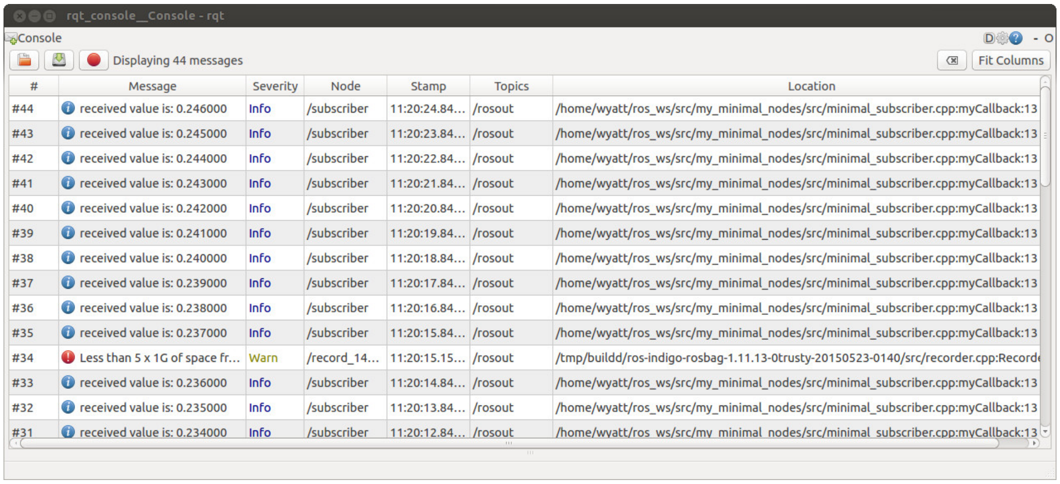


FIGURE 1.3: Output of rqt console with minimal nodes running and rosbag running

```
wyatt@Wall-E:~/ros_ws$ 452183621.541102   Duration: 5.701016 / 750.000001
```

`rqt_console` indicates that the bagfile has been opened, but no other information is displayed. What is happening at this point is that `rosbag` is replaying the recorded data by publishing the recorded values (from `topic1`) to the topic `topic1`, and it is doing so at the same rate as the data was recorded.

To see that this is taking place, do the following. Halt all nodes (including `rosbag`). Run `rqt_console`. In a terminal window, start up the subscriber, but not the publisher, using:

```
rosrun  my_minimal_nodes  my_minimal_subscriber
```

At this point, this terminal is suspended, because `topic1` is not yet active.

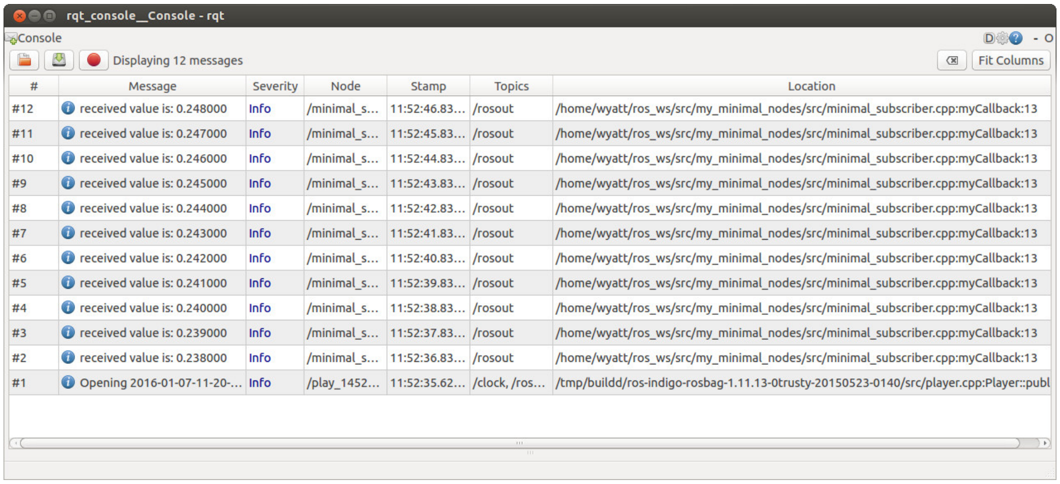
From another terminal, navigate to the `bagfiles` directory and enter:

```
rosbag  play fname.bag
```

where “fname” is (again) the name of the recording that was bagged previously. `rosbag` now assumes the role formerly taken by `sleepy_minimal_publisher`, recreating the messages that were formerly published. The `my_minimal_subscriber` window reports the recorded data, updating once per second. `rqt_console` also shows the data posted by the minimal subscriber. As can be seen from the screenshot in 1.4, the playback is corresponds to the original recording, with the first output (console line 2) displaying a value of 0.238 from the subscriber node.

Dynamically, these values are posted at the original 1Hz rate that `sleepy_minimal_publisher` had published them. The `rosbag` player terminates when it gets to the end of the recorded data.

Note that our subscriber is oblivious to what entity is publishing to `topic1`. Consequently, playback of previously recorded data is indistinguishable from receiving live data. This is very useful for development. For example, a robot may be teleoperated through an environment of interest while it publishes sensor data from cameras, LIDAR, etc. Using `rosbag`, this data may be recorded verbatim. Subsequently, sensor processing may be performed on the recorded data to test, e.g., machine vision algorithms. Once a sensory-interpretation node is shown to be effective on the recorded data, the same node may be


rqt_console__Console - rqt

Console

Displaying 12 messages

Fit Columns

#	Message	Severity	Node	Stamp	Topics	Location
#12	received value is: 0.248000	Info	/minimal_s...	11:52:46.83...	/rosout	/home/wyatt/ros_ws/src/my_minimal_nodes/src/minimal_subscriber.cpp:myCallback:13
#11	received value is: 0.247000	Info	/minimal_s...	11:52:45.83...	/rosout	/home/wyatt/ros_ws/src/my_minimal_nodes/src/minimal_subscriber.cpp:myCallback:13
#10	received value is: 0.246000	Info	/minimal_s...	11:52:44.83...	/rosout	/home/wyatt/ros_ws/src/my_minimal_nodes/src/minimal_subscriber.cpp:myCallback:13
#9	received value is: 0.245000	Info	/minimal_s...	11:52:43.83...	/rosout	/home/wyatt/ros_ws/src/my_minimal_nodes/src/minimal_subscriber.cpp:myCallback:13
#8	received value is: 0.244000	Info	/minimal_s...	11:52:42.83...	/rosout	/home/wyatt/ros_ws/src/my_minimal_nodes/src/minimal_subscriber.cpp:myCallback:13
#7	received value is: 0.243000	Info	/minimal_s...	11:52:41.83...	/rosout	/home/wyatt/ros_ws/src/my_minimal_nodes/src/minimal_subscriber.cpp:myCallback:13
#6	received value is: 0.242000	Info	/minimal_s...	11:52:40.83...	/rosout	/home/wyatt/ros_ws/src/my_minimal_nodes/src/minimal_subscriber.cpp:myCallback:13
#5	received value is: 0.241000	Info	/minimal_s...	11:52:39.83...	/rosout	/home/wyatt/ros_ws/src/my_minimal_nodes/src/minimal_subscriber.cpp:myCallback:13
#4	received value is: 0.240000	Info	/minimal_s...	11:52:38.83...	/rosout	/home/wyatt/ros_ws/src/my_minimal_nodes/src/minimal_subscriber.cpp:myCallback:13
#3	received value is: 0.239000	Info	/minimal_s...	11:52:37.83...	/rosout	/home/wyatt/ros_ws/src/my_minimal_nodes/src/minimal_subscriber.cpp:myCallback:13
#2	received value is: 0.238000	Info	/minimal_s...	11:52:36.83...	/rosout	/home/wyatt/ros_ws/src/my_minimal_nodes/src/minimal_subscriber.cpp:myCallback:13
#1	Opening 2016-01-07-11-20-...	Info	/play_1452...	11:52:35.62...	/clock, /ros...	/tmp/buildd/ros-indigo-rosbag-1.11.13-0trusty-20150523-0140/src/player.cpp:Player::publ

FIGURE 1.4: Output of rqt console with minimal subscriber and rosbag play of recorded (bagged) data

tried out verbatim on the robot system. Note that no changes to the developed node are needed. In live experiments, this node would merely be receiving messages published by the real-time system instead of by `rosbag playback`.

1.4 A MINIMAL SIMULATOR AND CONTROLLER EXAMPLE

Concluding this introduction, we consider a pair of nodes that both publish and subscribe. One of these nodes is a minimal simulator, and the other is a minimal controller. The minimal simulator simulates $F = ma$ by integrating acceleration to update velocities. The input force is published to the topic `force_cmd` by some entity (eventually, the controller). The resulting system state (the velocity) is published to the topic `velocity` by the minimal simulator.

The simulator code is shown in Listing 1.8, and the source code is in the accompanying repository, in `https://github.com/wsnewman/learning_ros/tree/master/Part_1/minimal_nodes/src`.

The main() function initializes both a publisher and a subscriber. Formerly, we saw nodes as dedicated publishers or dedicated subscribers, but a node can (and often does) perform both actions. Equivalently, the simulator node behaves likea link in a chain, processing incoming data and providing timely output publications.

Another difference from our previous publisher and subscriber nodes is that both the callback function and the main routine perform useful actions. The minimal simulator has a callback routine that checks for new data on the topic `force_cmd`. When the callback receives new data, it copies it to a global variable, `g_force`, so that the main program has access to it. The main() function iterates at a fixed rate (set to 100Hz). In order for the callback function to respond to incoming data, the main function must provide "spin" opportunities. Formerly, our minimal subscriber used the "spin()" function, but this resulted in the main() function ceasing to contribute new computations.

An important new feature in the minimal-simulator example is the use of the ROS function: `ros::spinOnce()`. This function, executed within the 100Hz loop of the simulator, allows the callback function to process incoming data at 10ms intervals. If a new input (a force stimulus) is received, it is stored in `g_force` by the callback function. In the complementary minimal-controller node, values of force stimulus are published at only 10Hz. Consequently, 9 out of 10 times there will be no new messages on the `force_command` topic. The callback function will not block, but neither will it update the value of `g_force` when there is no new transmission available. The main loop will still repeat its iterations at 100Hz, albeit re-using "stale" data in `g_force`. This behavior is realistic, since a simulator should emulate realistic physics. For an actuated joint that is digitally controlled, controller effort (force or torque) commands typically behave with a sample-and-hold output between controller updates (sample periods).

Listing 1.8: Minimal Simulator

```
// minimal_simulator node:
// wsn example node that both subscribes and publishes
// does trivial system simulation, F=ma, to update velocity given F specified on topic "force_cmd"
// publishes velocity on topic "velocity"
#include<ros/ros.h>
#include<std_msgs/Float64.h>
std_msgs::Float64 g_velocity;
std_msgs::Float64 g_force;
void myCallback(const std_msgs::Float64& message_holder)
{
// checks for messages on topic "force_cmd"
ROS_INFO("received force value is: %f",message_holder.data);
g_force.data = message_holder.data; // post the received data in a global var for access by
// main prog.
}
int main(int argc, char **argv)
```

```
{
ros::init(argc,argv,"minimal_simulator"); //name this node
// when this compiled code is run, ROS will recognize it as a node called "minimal_simulator"
ros::NodeHandle nh; // node handle
//create a Subscriber object and have it subscribe to the topic "force_cmd"
ros::Subscriber my_subscriber_object= nh.subscribe("force_cmd",1,myCallback);
//simulate accelerations and publish the resulting velocity;
ros::Publisher my_publisher_object = nh.advertise<std_msgs::Float64>("velocity",1);
double mass=1.0;
double dt = 0.01; //10ms integration time step
double sample_rate = 1.0/dt; // compute the corresponding update frequency
ros::Rate naptime(sample_rate);
g_velocity.data=0.0; //initialize velocity to zero
g_force.data=0.0; // initialize force to 0; will get updated by callback
while(ros::ok())
{
g_velocity.data = g_velocity.data + (g_force.data/mass)*dt; // Euler integration of
//acceleration
my_publisher_object.publish(g_velocity); // publish the system state (trivial--1-D)
ROS_INFO("velocity = %f",g_velocity.data);
ros::spinOnce(); //allow data update from callback
naptime.sleep(); // wait for remainder of specified period; this loop rate is faster than
// the update rate of the 10Hz controller that specifies force_cmd
// however, simulator must advance each 10ms regardless
}
return 0; // should never get here, unless roscore dies
}
```

The minimal simulator may be compiled and run. Running `rqt_console` shows that the velocity has a persistent value of 0.

The result can be visualized graphically with the ROS tool `rqt_plot`. To do so, use command-line arguments for the values to be plotted, e.g.:

```
rqt_plot velocity/data
```

will plot the velocity command vs time. This output will be boring, at present, since the velocity is always zero.

One can manually publish values to a topic from a command line. For example, enter the following command in a terminal:

```
rostopic pub r 10 force_cmd std_msgs/Float64 0.1
```

This will cause the value 0.1 to be published repeatedly on the topic `force_cmd` at a rate of 10Hz using the consistent message type: `std_msgs/Float64`. This can be confirmed (from another terminal) with:

```
rostopic echo force_cmd
```

which will show that the `force_cmd` topic is receiving the prescribed value.

Additionally, invoking:

```
rqt_plot velocity/data
```

will will show that the velocity is increasing linearly, and `rqt_console` will print out the corresponding values (for both force and velocity).

Instead of publishing force-command values manually, these can be computed and published by a controller. Listing 1.9 displays a compatible minimal controller node. (This code is also contained in the accompanying examples repository under `https://github.com/wsnewman/learning_ros/tree/master/Part_1/minimal_nodes/src`).

The minimal controller subscribes to 2 topics, (`velocity` and `vel_cmd`), and it publishes to the topic `force_cmd`. Each control cycle (set to 10Hz), the controller checks for the latest system state (velocity), checks for any updates to the commanded velocity, and it computes a proportional-error feedback to derive (and publish) a force command. This simple controller attempts to drive the simulated system to the user-commanded velocity setpoint.

Again, the `ros::spinOnce()` function is used to prevent blocking in the timed, main loop. Callback functions put received message data in the global variables `g_velocity` and `g_vel_cmd`.

Listing 1.9: Minimal Controller

```
// minimal_controller node:
// wsn example node that both subscribes and publishes--counterpart to minimal_simulator
// subscribes to "velocity" and publishes "force_cmd"
// subscribes to "vel_cmd"
#include<ros/ros.h>
#include<std_msgs/Float64.h>
//global variables for callback functions to populate for use in main program
std_msgs::Float64 g_velocity;
std_msgs::Float64 g_vel_cmd;
std_msgs::Float64 g_force; // this one does not need to be global...
void myCallbackVelocity(const std_msgs::Float64& message_holder)
{
// check for data on topic "velocity"
ROS_INFO("received velocity value is: %f",message_holder.data);
g_velocity.data = message_holder.data; // post the received data in a global var for access by
//main prog.
}
void myCallbackVelCmd(const std_msgs::Float64& message_holder)
{
// check for data on topic "vel_cmd"
ROS_INFO("received velocity command value is: %f",message_holder.data);
g_vel_cmd.data = message_holder.data; // post the received data in a global var for access by
//main prog.

 }
int main(int argc, char **argv)
{
ros::init(argc,argv,"minimal_controller"); //name this node
// when this compiled code is run, ROS will recognize it as a node called "minimal_controller"
ros::NodeHandle nh; // node handle
//create 2 subscribers: one for state sensing (velocity) and one for velocity commands
ros::Subscriber my_subscriber_object1= nh.subscribe("velocity",1,myCallbackVelocity);
ros::Subscriber my_subscriber_object2= nh.subscribe("vel_cmd",1,myCallbackVelCmd);
//publish a force command computed by this controller;
ros::Publisher my_publisher_object = nh.advertise<std_msgs::Float64>("force_cmd",1);
double Kv=1.0; // velocity feedback gain
double dt_controller = 0.1; //specify 10Hz controller sample rate (pretty slow, but
//illustrative)
double sample_rate = 1.0/dt_controller; // compute the corresponding update frequency
ros::Rate naptime(sample_rate); // use to regulate loop rate
g_velocity.data=0.0; //initialize velocity to zero
g_force.data=0.0; // initialize force to 0; will get updated by callback
g_vel_cmd.data=0.0; // init velocity command to zero
double vel_err=0.0; // velocity error
// enter the main loop: get velocity state and velocity commands
// compute command force to get system velocity to match velocity command
// publish this force for use by the complementary simulator
while(ros::ok())
{
vel_err = g_vel_cmd.data - g_velocity.data; // compute error btwn desired and actual
//velocities
g_force.data = Kv*vel_err; //proportional-only velocity-error feedback defines commanded
//force
```

```
my_publisher_object.publish(g_force); // publish the control effort computed by this
//controller
ROS_INFO("force command = %f",g_force.data);
ros::spinOnce(); //allow data update from callback;
naptime.sleep(); // wait for remainder of specified period;
}
return 0; // should never get here, unless roscore dies
}
```

Once the two nodes are compiled with `catkin_make` (which requires editing `CMakeLists.txt` to add these executables to the package), they can be run (with `rosrun`) from separate terminal windows (assuming `roscore` is running). Running `rqt_console` reveals that the force command is updated once for every 10 updates of velocity (as expected for the simulator at 100Hz and the controller at 10Hz).

The velocity command may be input from another terminal using a command line, e.g.:

```
rostopic pub r 10 vel_cmd  std_msgs/Float64 1.0
```

publishes the value “1.0” to the topic `vel_cmd` repeatedly at a rate of 10Hz. Watching the output on `rqt_console`, the velocity can be seen to converge exponentially on the desired value of `vel_cmd`.

The result can be visualized graphically with the ROS tool `rqt_plot`. To do so, use command- line arguments for the values to be plotted, e.g.:

```
rqt_plot vel_cmd/data,velocity/data,force_cmd/data
```

will plot the velocity command, the actual velocity and the force commands on the same plot vs time. For the minimal simulator and minimal controller, the velocity command was initially set to 0.0 via `rostopic pub`. Subsequently, the command was set to 2.0. A snippet of the resulting `rqt_plot` is shown below. The control effort (in red) reacts to accelerate the velocity closer to the goal of 2.0, then the control effort decreases. Ultimately, the system velocity converges on the goal value and the required control effort decreases to zero.

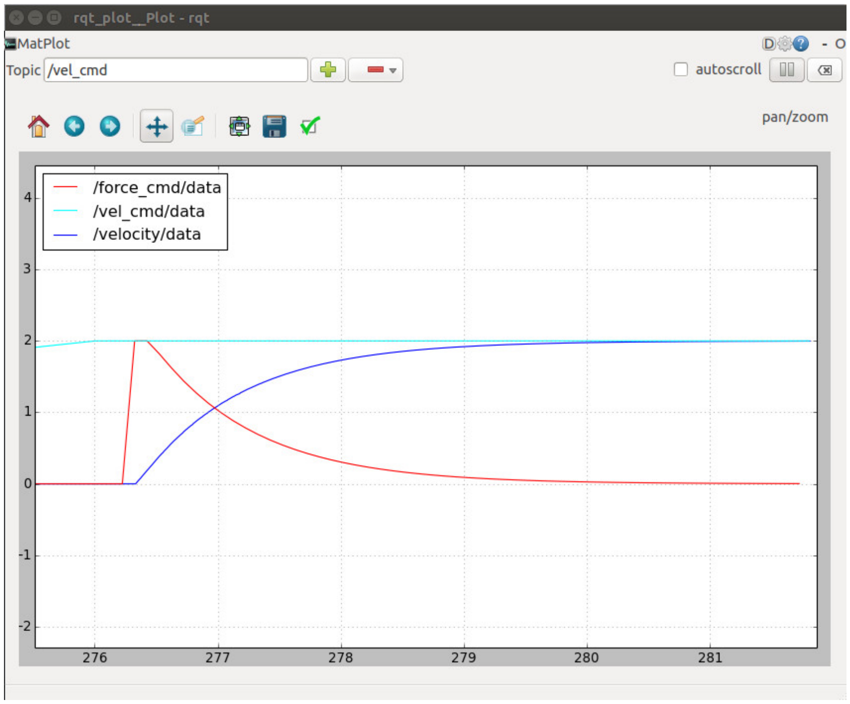


FIGURE 1.5: Output of rqt plot with minimal simulator and minimal controller and step velocity-command input via console

1.5 WRAP-UP

This chapter has introduced the reader to some basics of ROS. It has been shown that multiple, asynchronous "nodes" can communicate with each other through the mechanisms of "publish" and "subscribe" of "messages" on "topics," as coordinated by "roscore." Some ROS tools were introduced for the purpose of compiling nodes (`catkin_create_pkg`, `cs_create_pkg` and `catkin_make`), for logging and replaying published information (`rosbag`) and for visualizing ROS output (`rqt_console` and `rqt_plot`).

In the next chapter, additional communications topics are introduced, including defining custom message types, using client/server interactions, the common design paradigm of "action servers.," and the ROS parameter server.

CHAPTER 2

Messages, Classes and Servers

CONTENTS

INTRODUCTION
The previous chapter introduced how ROS nodes can communicate via publishing and subscribing. Example (trivial) message types were introduced. To make publishing/subscribing more general, it is helpful to be able to define one's own custom messages. Also, using publish/subscribe is not always appropriate. Sometimes a peer-to-peer communication is necessary. This capability is realized in ROS through a client/service interaction. Client/Service interactions address concerns of publish/subscribe in terms of knowledge of the source of communications and guaranteed receipt of messages. A limitation of client/service communications, though, is that these transactions are "blocking," and thus the client node is suspended until the service node responds. Often, the service to be performed can require significant time to complete, and in which cases it is desirable that the interaction be non-blocking. For this purpose, a third interaction mechanism is available: the "action client"/"action server." These three options will be covered in this Chapter.

2.1 DEFINING CUSTOM MESSAGES

ROS message types are based on 14 primitives (built-in types), plus fixed or variable-length arrays of these. (see `http://wiki.ros.org/msg`. Using these built-in types, more complex message types can be constructed.

Our minimal nodes illustrated use of standard messages (`std_msgs`) for communicating via publish/subscribe. The `std_msgs` package defines 32 message types, most of which correspond to a single built-in field (data) type. A notable exception is `Header.msg`, which is comprised of three fields, each of which corresponds to a built-in type.

More sophisticated message types can be defined by including other defined message types (not necessarily primitive, built-in types). For example, it is common to include the Header message type within higher-level message definitions, since including a time stamp is a common need. The ability to include defined message types within new message definitions is recursively extensible (with messages that include messages that include messages ...); ultimately, though, the higher-level message type is entirely definable in terms of the built-in primites.

Some useful (more complex) messages are defined in additional packages, such as: `geometry_msgs`, `sensor_msgs`, `nav_msgs`, `pcl_msgs`, `visualization_msgs`, `trajectory_msgs`, and `actionlib_msgs`.

Defined messages can be examined interactively using `rosmsg show`, followed by the package/messageName of the message of interest. E.g., entering:

```
rosmsg show std_msgs/Header
```

outputs:

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

which shows that "Header" is comprised of 3 fields: `seq`, `stamp` and `frame_id`. These message names store data of primitive types `uint32`, `time` and `string`, respectively.

If a message type already exists in the standard distribution of ROS, you should use that existing message. However, it is sometimes necessary to define one's own message. Defining custom messages is described at: `http://wiki.ros.org/ROS/Tutorials/DefiningCustomMessages`, which can be consulted for more details. The next section introduces the basics of how to define custom messages.

2.1.1 Defining a custom message

With reference to corresponding code in the accompanying repository within the package `example_ros_msg`.

The package `example_ros_msg` was created using:

```
cs_create_pkg example_ros_msg roscpp std_msgs
```

This creates a directory structure under `example_ros_msg`. By using `cs_create_pkg`, we will be able to use the the abbreviated `Cmakelists.txt`.

To define a new message type, we create a subdirectory in this package called `msg`. Within this `msg` directory, we create a new text file by the name of `ExampleMessage.msg`. The example message file contains only 3 relevant lines:

```
Header header
int32 demo_int
float64 demo_double
```

This message type will have three fields, which may be referred to by the names: `header`, `demo_int` and `demo_double`. Their types are `Header`, `int32` and `float64`, respectively, which are all message types defined in the package `std_msgs`.

To inform the compiler that we need to generate new message headers, the “package.xml” file must be edited. Insert (or uncomment) the following lines:

```
<build_depend>message_generation</build_depend>
```

and

```
<run_depend>message_runtime</run_depend>
```

The abbreviated `CMakeLists.txt` file (with unnecessary comments removed) is simply:

Listing 2.1: `CMakeLists.txt using catkin_simple`

```
cmake_minimum_required(VERSION 2.8.3)
project(example_ros_msg)

find_package(catkin_simple REQUIRED)

catkin_simple()

# Executables
#cs_add_executable(example_ros_message_publisher src/example_ros_message_publisher.cpp)

cs_install()
cs_export()
```

Note the `cs_add_executable` is commented out. We will enable this once we have our anticipated source code for a test node, `example_ros_message_publisher.cpp`.

Having defined a message type, we can generate corresponding header files suitable for C++ file inclusion. Compiling the code with `catkin_make` produces a header file, which it installs in the directory: `~/ros_ws/devel/include/example_ros_msg/ExampleMessage.h` (Reminder: here and throughout this text, it will be assumed that the “ros workspace” is called `ros_ws`).

Source code for nodes that want to use this new message type should depend on the package `example_ros_msg` (in the corresponding `package.xml` file) and should include the new header with the line:

```
#include <example_ros_msg/ExampleMessage.h>
```

in the C++ source code of the node using this message type. An illustrative example follows.

The accompanying code repository (in `https://github.com/wsnewman/learning_ros/tree/master/Part_1/minimal_nodes`) includes a source file under `example_ros_msg/src/example_ros_message_publisher.cpp`. The source code is as follows:

Listing 2.2: Example node using custom message type: `example_ros_message_publisher`

```
#include <ros/ros.h>
#include <example_ros_msg/ExampleMessage.h>
#include <math.h>

int main(int argc, char **argv) {
    ros::init(argc, argv, "example_ros_message_publisher"); // name of this node
    ros::NodeHandle n; // two lines to create a publisher object that can talk to ROS
    ros::Publisher my_publisher_object = n.advertise<example_ros_msg::ExampleMessage>("example_topic", 1);
    //"example_topic" is the name of the topic to which we will publish
    // the "1" argument says to use a buffer size of 1; could make larger, if expect network backups

    example_ros_msg::ExampleMessage  my_new_message;
    //create a variable of type "example_msg",
    // as defined in this package

   ros::Rate naptime(1.0); //create a ros object from the ros Rate class;
   //set the sleep timer for 1Hz repetition rate (arg is in units of Hz)

   // put some data in the header.  Do: rosmsg show std_msgs/Header
   //  to see the definition of "Header" in std_msgs
    my_new_message.header.stamp = ros::Time::now(); //set the time stamp in the header;
    my_new_message.header.seq=0; // call this sequence number zero
    my_new_message.header.frame_id = "base_frame"; // would want to put true reference frame name here, if needed for coord transforms
    my_new_message.demo_int= 1;
    my_new_message.demo_double=100.0;

    double sqrt_arg;
    // do work here in infinite loop (desired for this example), but terminate if detect ROS has faulted
    while (ros::ok())
    {
       my_new_message.header.seq++; //increment the sequence counter
       my_new_message.header.stamp = ros::Time::now(); //update the time stamp
       my_new_message.demo_int*=2.0; //double the integer in this field
       sqrt_arg = my_new_message.demo_double;
       my_new_message.demo_double = sqrt(sqrt_arg);

        my_publisher_object.publish(my_new_message); // publish the data in new message format on topic "example_topic"
    //the next line will cause the loop to sleep for the balance of the desired period
        // to achieve the specified loop frequency
    naptime.sleep();
    }
}
```

This node uses the new message type as follows. It defines a publisher object as:

```
        ros::Publisher my_publisher_object = n.advertise<example_ros_msg::ExampleMessage>("example_topic", 1);
```

which says that topic `example_topic` will carry messages of type `example_ros_msg::ExampleMessage`. (The message type is identified by referring to the package that contains it, `example_ros_msg`, followed by the preamble of the file name that details the format of the message, i.e. `ExampleMessage` taken from filename `ExampleMessage.msg`).

We also instantiate an object of type `example_ros_msg::ExampleMessage` with the line:

```
     example_ros_msg::ExampleMessage  my_new_message;
```

Note that when referring to the header file, we use the notation `example_ros_msg/ExampleMessage.h` (path to the header file), but when instantiating an object based on this definition (or referring to the datatype for publication), we use the class notation: `example_ros_msg::ExampleMessage`.

Within the source code of `example_ros_message_publisher.cpp` the various fields of the new message object, `my_new_message`, are populated, and then this message is published.

Populating fields of the new message type is simple, e.g.:

```
    my_new_message.demo_int= 1;
```

Accessing elements of hierarchical fields requires drilling down deeper, as in:

```
my_new_message.header.stamp = ros::Time::now(); //set the time stamp in the header;
```

Here, "stamp" is a field within the "Header" message type for the field "header." Additionally, this line of code illustrates another useful ROS function: `ros::Time::now()`. This looks up the current time and returns it in a form compatible with "header" (consisting of separate fields for seconds and nanoseconds). Note: the absolute time is essentially meaningless. However, differences in time can be used as valid time increments.

By uncommenting the line in CmakeLists.txt,

```
cs_add_executable(example_ros_message_publisher src/example_ros_message_publisher.cpp)
```

and re-running `catkin_make`, a new node is created, with the name `example_ros_message_publisher`.

Running this node (assuming roscore is running):

```
rosrun example_ros_msg example_ros_message_publisher
```

produces no output. However, (from a separate terminal), running:

```
    rostopic list
```

reveals that there is a new topic, `example_topic`. We can examine the output of this topic with:

```
rostopic echo example_topic
```

which produces the following output:

```
header:
  seq: 1
  stamp:
    secs: 1452225386
    nsecs: 619262393
  frame_id: base_frame
demo_int: 4
demo_double: 3.16227766017
---
header:
  seq: 2
  stamp:
    secs: 1452225387
    nsecs: 619259445
  frame_id: base_frame
demo_int: 8
demo_double: 1.77827941004
---
```

```
header:
  seq: 3
  stamp:
    secs: 1452225388
    nsecs: 619234854
  frame_id: base_frame
demo_int: 16
demo_double: 1.33352143216
---
```

We see that our new node successfully uses the new message type. Sequence numbers increase monotonically. The `demo_int` field is doubled each iteration (per the logic of the source code). The `demo_double` field displays sequential square-roots (starting from 100). The "secs" field of the header increments by 1 second each iteration (since the iteration rate timer was set to 1Hz). The string `base_frame` appears in the `frame_id` field.

Following the same process, one can create more customized message types. After defining a new message type, nodes within the same package or nodes in other packages can use the new message type, provided the external packages list `example_ros_msg` as a dependency (in the corresponding package.xml file).

2.1.2 Defining a variable-length message

One very useful extension of the ROS message primitives is the ability to send/receive arbitrary-length vectors of message types. To illustrate how to do this, refer to the package `custom_msgs` within the `Part_1` folder of the accompanying code repository. The package `custom_msgs` contains no source code, but it does have files `CMakeLists.txt` and `package.xml`, as well as a "msg" folder. The `msg` folder contains a message file, `VecOfDoubles.msg` with the following contents (a single line):

```
float64[] dbl_vec
```

Although our new package has no source files to compile, we still need to execute the process that generates header files from `*.msg` files. To invoke this, uncomment two lines in the `package.xml` file:

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

With these edits, the new package can by built by running `catkin_make`. This results a new header file, `VecOfDoubles.h`, appearing in the folder `:~/ros_ws/devel/include/custom_msgs`. One can confirm that the new message has been installed by entering:

```
rosmsg show custom_msgs/VecOfDoubles
```

which yields the response:

```
float64[] dbl_vec
```

When this message is received by a subscriber, it can be interpreted as a C++ "vector" of doubles.

An example publisher using the new message type is `vector_publisher.cpp`, which resides in the `src` folder of the `example_ros_msg` package. The code listing of the `vector_publisher.cpp` follows:

Listing 2.3: Publisher node using custom message type: `VecOfDoubles`

```
1  #include <ros/ros.h>
2  //next line requires a dependency on custom_msgs within package.xml
3  #include <custom_msgs/VecOfDoubles.h> //this is the message type we are testing
4
5  int main(int argc, char **argv) {
6      ros::init(argc, argv, "vector_publisher"); // name of this node
7      ros::NodeHandle n; // two lines to create a publisher object that can talk to ROS
8      ros::Publisher my_publisher_object = n.advertise<custom_msgs::VecOfDoubles>("vec_topic", 1);
9
10     custom_msgs::VecOfDoubles vec_msg; //create an instance of this message type
11     double counter=0;
12     ros::Rate naptime(1.0); //create a ros object from the ros Rate class; set 1Hz rate
13
14     vec_msg.dbl_vec.resize(3); //manually resize it to hold 3 doubles
15     //After setting the size, one can access elements of this array conventionally, e.g.
16     vec_msg.dbl_vec[0]=1.414;
17     vec_msg.dbl_vec[1]=2.71828;
18     vec_msg.dbl_vec[2]=3.1416;
19
20     //Alternatively, one can use the vector member function push_back() to append data to an existing array, e.g.:
21     vec_msg.dbl_vec.push_back(counter); // this makes the vector longer, to hold additional data
22     while(ros::ok())  {
23     counter+=1.0;
24     vec_msg.dbl_vec.push_back(counter);
25     my_publisher_object.publish(vec_msg);
26     naptime.sleep();
27     }
28 }
```

On line 2 of Listing 2.3, the message header for our new vector message is included:

```
#include <custom_msgs/VecOfDoubles.h> //this is the message type we are testing
```

On line 8, a publisher is instantiated, as before, but this time using on our new message type.

```
ros::Publisher my_publisher_object = n.advertise<custom_msgs::VecOfDoubles>("vec_topic", 1);
```

The topic to which it will publish is `vec_topic`.

An instance of our new message type, named `vec_msg` is declared on line 10:

```
custom_msgs::VecOfDoubles vec_msg; //create an instance of this message type
```

The variable-length message can be resized, if desired, as in line 14:

```
vec_msg.dbl_vec.resize(3); //manually resize it to hold 3 doubles
```

Note that one must refer to the field name `dbl_vec` within the message `vec_msg`, from which one can invoke member functions of the vector object, including `size()`, `resize()`, and `push_back()`. After resizing the vector, individual elements may be accessed by indexing it. Three values are stored in the first three locations with lines 16-18:

```
    vec_msg.dbl_vec[0]=1.414;
    vec_msg.dbl_vec[1]=2.71828;
    vec_msg.dbl_vec[2]=3.1416;
```

Alternatively, one may append data to the vector, which is automatically resized to accommodate the addition data. This is done with the push_back member function, as in line 21:

```
vec_msg.dbl_vec.push_back(counter); // this makes the vector longer, to hold ↩
    additional data
```

The example program then goes into a timed loop, incrementing a counter by 1 each iteration, appending this value to the variable-length vector, and publishing the result via line 25

```
my_publisher_object.publish(vec_msg);
```

In order to compile this new node in our package example_ros_msg, both the package.xml and the CMakeLists.txt within this package must be edited. The package.xml must be informed of the new dependency on our new message. This is done by inserting the lines:

```
  <build_depend>custom_msgs</build_depend>
  <run_depend>custom_msgs</run_depend>
```

Logically, these can be inserted following the same tags for std_msgs, although the order is flexible.

The second required change is to edit CMakeLists.txt to invoke compiling of our new node. This is done by inserting the line:

```
cs_add_executable(vector_publisher src/vector_publisher.cpp)
```

With these changes, the new node is compiled with catkin_make. The new node can be run with:

```
rosrun example_ros_msg vector_publisher
```

This results in publishing a variable-length vector to the topic vec_topic once per second. This effect can be observed by running:

```
rostopic echo vec_topic
```

which produces the following output:

```
dbl_vec: [1.414, 2.71828, 3.1416, 0.0, 1.0, 2.0]
---
dbl_vec: [1.414, 2.71828, 3.1416, 0.0, 1.0, 2.0, 3.0]
---
dbl_vec: [1.414, 2.71828, 3.1416, 0.0, 1.0, 2.0, 3.0, 4.0]
---
dbl_vec: [1.414, 2.71828, 3.1416, 0.0, 1.0, 2.0, 3.0, 4.0, 5.0]
---
dbl_vec: [1.414, 2.71828, 3.1416, 0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0]
---
dbl_vec: [1.414, 2.71828, 3.1416, 0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0]
```

This shows that the variable-length vector message is increasing in length each iteration, due to the push_back operations.

A complementary subscriber node is also include in the `src` folder of the `example_ros_msg` package: `vector_subscriber.cpp`. It largely follows the example `minimal_subscriber.cpp`. The contents of `vector_subscriber.cpp` is given in Listing 2.4.

Listing 2.4: Subscriber node using custom message type: `VecOfDoubles`

```
1  #include<ros/ros.h>
2  #include <custom_msgs/VecOfDoubles.h> //this is the message type we are testing
3  void myCallback(const custom_msgs::VecOfDoubles& message_holder)
4  {
5    std::vector <double> vec_of_doubles = message_holder.dbl_vec; //can copy contents of message to a C++ vector like this
6    int nvals = vec_of_doubles.size(); //ask the vector how long it is
7    for (int i=0;i<nvals;i++) {
8      ROS_INFO("vec[%d] = %f",i,vec_of_doubles[i]); //print out all the values
9    }
10   ROS_INFO("\n");
11 }
12
13 int main(int argc, char **argv)
14 {
15   ros::init(argc,argv,"vector_subscriber"); //default name of this node
16   ros::NodeHandle n; // need this to establish communications with our new node
17
18   ros::Subscriber my_subscriber_object= n.subscribe("vec_topic",1,myCallback);
19
20   ros::spin();
21   return 0; // should never get here, unless roscore dies
22 }
```

In Listing 2.4, line 2 brings in the header file of our new message, as was done with the publisher as well:

```
#include <custom_msgs/VecOfDoubles.h> //this is the message type we are testing
```

A subscriber is instantiated in the main() function, line 18:

```
ros::Subscriber my_subscriber_object= n.subscribe("vec_topic",1,myCallback);
```

This is identical to the earlier `minimal_subscriber.cpp` example, except that the topic name has been changed to `vec_topic`, to be consistent with our vector publisher node.

The callback function, starting on line 3:

```
void myCallback(const custom_msgs::VecOfDoubles& message_holder)
```

is defined to accept an argument based on our new variable-length vector message.

The callback function is woken up by the receipt of a new transmission on the `vec_topic` topic. The received data is contained within `message_holder`, the contents of which can be copied to a C++ vector, as shown in line 5:

```
std::vector <double> vec_of_doubles = message_holder.dbl_vec;
```

Subsequently, this vector object can be used with all of its member functions. For example, the length of the vector can be found, as in line 6:

```
int nvals = vec_of_doubles.size(); //ask the vector how long it is
```

The callback function then iterates through all components of the vector, displaying them with `ROS_INFO()`.

To compile the subscriber node, add the following line to the`CMakeLists.txt` file:

```
cs_add_executable(vector_subscriber src/vector_subscriber.cpp)
```

then compile with `catkin_make`.

To test the operation of the compementary nodes, (with roscore running) in one terminal, enter:

```
rosrun example_ros_msg vector_publisher
```

In a second terminal, start the subscriber with:

```
rosrun example_ros_msg vector_subscriber
```

As a result, the subscriber displays the following example output:

```
[ INFO] [1452450667.220847072]: vec[0] = 1.414000
[ INFO] [1452450667.220985967]: vec[1] = 2.718280
[ INFO] [1452450667.221041396]: vec[2] = 3.141600
[ INFO] [1452450667.221097413]: vec[3] = 0.000000
[ INFO] [1452450667.221151168]: vec[4] = 1.000000
[ INFO] [1452450667.221208351]: vec[5] = 2.000000
[ INFO] [1452450667.221265818]:

[ INFO] [1452450668.220694702]: vec[0] = 1.414000
[ INFO] [1452450668.220802727]: vec[1] = 2.718280
[ INFO] [1452450668.220898220]: vec[2] = 3.141600
[ INFO] [1452450668.220989838]: vec[3] = 0.000000
[ INFO] [1452450668.221095457]: vec[4] = 1.000000
[ INFO] [1452450668.221200559]: vec[5] = 2.000000
[ INFO] [1452450668.221308814]: vec[6] = 3.000000
[ INFO] [1452450668.221393760]:

[ INFO] [1452450669.220683996]: vec[0] = 1.414000
[ INFO] [1452450669.220777258]: vec[1] = 2.718280
[ INFO] [1452450669.220820689]: vec[2] = 3.141600
[ INFO] [1452450669.220884177]: vec[3] = 0.000000
[ INFO] [1452450669.220937618]: vec[4] = 1.000000
[ INFO] [1452450669.220999832]: vec[5] = 2.000000
[ INFO] [1452450669.221057884]: vec[6] = 3.000000
[ INFO] [1452450669.221115040]: vec[7] = 4.000000
[ INFO] [1452450669.221172102]:
```

This shows that our publisher and subscriber are communicating with each other successfully using the new variable-length message type.

In practice, when using variable-length messages, one must be careful of two issues:

1. Make sure your variable-length vectors do not get overly large. It is all too easy to forget that the vector continues to grow, and thus you could consume all of your memory and all of your communications bandwidth as the message gets too large.

2. As with conventional arrays, if you try to access memory that has not been allocated, you will get errors (typically a segmentation fault). E.g., if your received message copied to `vec_of_doubles` is 3 elements long, trying to access `vec_of_doubles[3]` (the 4th element) will result in run-time errors (but no compiler warnings).

This concludes the introduction to defining message types. As will be shown, the same process is used for defining additional message types used for ROS services and ROS action servers.

2.2 INTRODUCTION TO ROS SERVICES

So far, our primary means of communications among nodes has been publish/subscribe. In this mode of communications, the publisher is unaware of its subscribers, and a subscriber knows only of a topic, not which node might publish to that topic. Messages are sent at unknown intervals, and subscribers might miss messages. This style of communications is appropriate for repetitive messages, such as publication of sensor values. For such cases, the sensor publisher should not need to know which or how many nodes subscribe to its output, nor will the publisher change its message in response to any requests from consumers. This form of communications is simple and flexible. Provided the preceding restrictions are not of concern, publish/subscribe is preferred.

Alternatively, it is sometimes necessary to establish bi-directional, one-to-one, reliable communication. In this case, the "client" sends a "request" to a "service," and the service sends back a "response" to the client. The question/answer interaction is on demand, and the client is aware of the name of the service provider.

It should be noted that ROS "services" are intended to be fast responses. When a client sends a request, the client is suspended until an answer returns. ROS code should be tolerant of such delays. If the request involves extensive calculations or delays to respond, then you probably should use yet another alternative, the action server/action client mechanism (to be introduced in later notes).

The example below is contained in the package `example_ros_service` in the accompanying repository. It is introduced by first describing what is a "service" message, then how to construct a service provider node, and finally how to construct a service client node.

2.2.1 Service messages

Defining a custom service message requires describing the data types and field names for both a request and a response. This is done by a ROS template, called a "*.srv" file. Contents of this file auto-generate C++ headers that can be included in C++ files.

To create a new service message, within the desired package (in this example, `example_ros_service`), create a sub-directory called "srv." (This directory is at the same level as "src"). Inside this directory, create a text file named `*.srv`. In the current example, this text file has been named "ExampleServicMsg.srv." The contents of the example service message are given below:

```
string name
---
bool on_the_list
bool good_guy
int32 age
string nickname
```

In the above, the request structure is defined by the lines above "`---`", and the response structure is defined by the lines below "`---`". The request (for this simple example) consists of a single component, referenced by the name "name", and the datatype contained in this field is a (ROS) string.

For the response part of this example, there are 4 fields, named `on_the_list`, `good_guy`, `age` and `nickname`. These have respective datatypes of `bool`, `bool`, `int32` and `string` (all of which are defined as built-in ROS message types).

Although the service message is described very simply in a text file, the compiler will be instructed to parse this file and build a C++ header file that can be included in a C++ program. Just as we did with defining custom message types for publish/subscribe, we inform the compiler that we need to generate new message headers via the `package.xml` file. This file must be edited to insert (or uncomment) the following lines:

```
<build_depend>message_generation</build_depend>
```

and

```
<run_depend>message_runtime</run_depend>
```

The package `example_ros_service` was created with the help of `cs_create_pkg`, which uses `catkin_simple` to simplify the `CMakeLists.txt` file. Anticipating that we will want to compile two test nodes, `example_ros_service.cpp` and `example_ros_client.cpp`, corresponding lines in the CMakeLists.txt file specify that these source files should be compiled. The CMakeLists.txt file (with instructional comments removed, and with the `cs_add_executable` lines temporarily commented out) is:

Listing 2.5: `CMakeLists.txt using catkin_simple for the example client and service nodes`

```
cmake_minimum_required(VERSION 2.8.3)
project(example_ros_service)

find_package(catkin_simple REQUIRED)

catkin_simple()

#cs_add_executable(example_ros_service src/example_ros_service.cpp)
#cs_add_executable(example_ros_client src/example_ros_client.cpp)

cs_install()
cs_export()
```

Having defined the service message, one can test the package compilation even before any C++ source code has been written. The new package can be compiled by running `catkin_make` (as always, from the `ros_ws` directory).

Although no CMakeLists.txt has no active `cs_add_executable` instructions, the catkin build system will recognize that there is a new "srv" file, and it will auto-generate C++ compatible header files. In fact, it is common to have ROS packages that contain message definitions, but no C++ code. The pre-defined messages in `std_msgs` are contained within a package (named "`std_msgs`"), although this package contains no C++ source code. This is a common technique for defining messages that may be useful among multiple packages, since the messages thus do not require compiling any unnecessary code to be re-used with new nodes.

The header files that have been auto-generated reside within the directory `catkin/devel/include/example_ros_service`. It is not necessary to remember this directory path–`catkin_make` will know where to look for these headers if you list a dependency on the package `example_ros_service`. As with the creation of new message types for publish/subscribe, the auto-generated header file shares the same base name as the user-define "*.srv" file. For our example, after compilation the directory `catkin/devel/include/example_ros_service` contains a header file called "ExampleServiceMsg.h". What is slightly more complex than a publish/subscribe message type is that there are also two more service header files created in the `catkin/devel/include/example_ros_service` directory: "ExampleServiceMsgRequest.h" and "ExampleServiceMsgResponse.h". Both of these header files are included within the “ExampleServiceMsg.h” header file. Note that the names of these header files also contain the base name, “ExampleServiceMsg”, but they also have an appended component to their names, either “Request” or “Service.”

To use the new service message type in a node, one must include its associated header file in the C++ source code with the line:

```
#include <example_ros_service/ExampleServiceMsg.h>
```

Note that this header inclusion references the name of the package where the message is defined (`example_ros_service`), as well as the base name of the message itself (`ExampleServiceMsg`), with the suffix `.h` appended. The delimiters `<...>` tell the compiler to look for this file in the expected locations (in this case, `catkin/devel/include`).

In the present example, the source code for our illustrative nodes is part of the same `example_ros_service` package. If we wanted to use our new service message within nodes of a separate package, we would include `#include <example_ros_service/ExampleServiceMsg.h>` in our C++ code identically. However, there will be two differences in the new `package.xml` file. For one, it will not be necessary to depend on `message_generation` or `message_runtime`, since the message type needs to be generated only once (and this is performed with the package `example_ros_service` is compiled). The second difference is that the new package’s `package.xml` file must list a dependency on package `example_ros_service` by including the line: `<build_depend>example_ros_service</build_depend>`.

2.2.1.1 A ROS service node

An example ROS service node is defined in the package `example_ros_service`, in the subdirectory “src”, named `example_ros_service.cpp`. The `CMakeLists.txt` file is edited to instruct the compiler to compile this code by adding (or uncommenting) the line:

```
cs_add_executable(example_ros_service src/example_ros_service.cpp)
```

The source code of `example_ros_service.cpp` is given in Listing 2.6.

Listing 2.6: Example service node

```
//example ROS service:
// run this as: rosrun example_ROS_service example_ROS_service
// in another window, tickle it manually with (e.g.):
//    rosservice call lookup_by_name "Ted"


#include <ros/ros.h>
#include <example_ros_service/ExampleServiceMsg.h>
#include <iostream>
#include <string>
using namespace std;

bool callback(example_ros_service::ExampleServiceMsgRequest& request, example_ros_service::ExampleServiceMsgResponse& response)
{
    ROS_INFO("callback activated");
    string in_name(request.name); // convert this to a C++-class string, so can use member funcs
    //cout<<"in_name:"<<in_name<<endl;
    response.on_the_list=false;

    // here is a dumb way to access a stupid database...
    // hey: this example is about services, not databases!
    if (in_name.compare("Bob")==0)
```

```
 {
        ROS_INFO("asked about Bob");
        response.age = 32;
        response.good_guy=false;
        response.on_the_list=true;
        response.nickname="BobTheTerrible";
    }
     if (in_name.compare("Ted")==0)
 {
        ROS_INFO("asked about Ted");
        response.age = 21;
        response.good_guy=true;
        response.on_the_list=true;
        response.nickname="Ted the Benevolent";
    }

  return true;
}

int main(int argc, char **argv)
{
  ros::init(argc, argv, "example_ros_service");
  ros::NodeHandle n;

  ros::ServiceServer service = n.advertiseService("lookup_by_name", callback);
  ROS_INFO("Ready to look up names.");
  ros::spin();

  return 0;
}
```

In the example service code, note the inclusion of the new header file:

```
#include <example_ros_service/ExampleServiceMsg.h>
```

Within the body of main(), the line:

```
ros::ServiceServer service = n.advertiseService("lookup_by_name", callback);
```

is similar to creating a publisher in ROS. In this case, a service is created, and it will be known by the name `lookup_by_name`. When a request to this service arrives, the named callback function (in this case, simply named "callback") will be invoked. The service does not have a timed loop. Rather (using `ros::spin();` in the main function), it sleeps until a request comes in, and incoming requests are serviced by the callback function.

A less obvious attribute of the service node construction is the type declarations of the arguments of the service callback.

```
bool callback(example_ros_service::ExampleServiceMsgRequest& request, ↩
    example_ros_service::ExampleServiceMsgResponse& response)
```

The argument:

```
example_ros_service::ExampleServiceMsgRequest& request
```

declares that the argument "request" is a reference pointer of type `example_ros_service::ExampleServiceMsgReques` Similarly, the second argument of the callback function is:

```
example_ros_service::ExampleServiceMsgResponse& response
```

which declares that the argument named "response" is a reference pointer to an object of type `example_ros_service::ExampleServiceMsgResponse`.

This may seem strange, since we did not define a datatype called `ExampleServiceMsgRequest` within our package `example_ros_service`. The build system created this datatype as part of auto-generating the message header file. The name ExampleServiceMsgRequest is created by appending the name Request to our service message name, "ExampleServiceMsg" (and similarly for "Response"). When you define a new service message, you can assume that the system will create these two new datatypes for you.

When the service callback function is invoked, the callback can examine the contents of the incoming request. In the present example, the request has only one field in its definition, called "name." The line:

```
string in_name(request.name);
```

creates a C++ style "string" object from the characters contained in "request.name." With this string object, we can invoke the member function "compare()", e.g. as in line 22 of the code to test if the name is identical to "Bob."

```
if (in_name.compare("Bob")==0)
```

Within the callback routine, fields are filled in for the "response." When the callback returns, this response message is transmitted back to the client that invoked the request. The mechanism for performing this communication is hidden from the programmer; it is performed as part of the ROS service paradigm (e.g., you do not have to invoke an action similar to "publish(response)").

Once the ROS service example is compiled, it can be run (assuming a `roscore` is running) with:

```
rosrun example_ros_service example_ros_service
```

2.2.1.2 Manual interaction with a ROS service

Once the node `example_ros_service` is running, we can see that a new service is available. From a command prompt, enter:

```
    rosservice list
```

The response will show that there is a service by the name of `/lookup_by_name` (which we declared to be the service name of our node).

We can interact manually with the service from the command line, e.g. by typing:

```
    rosservice call lookup_by_name 'Ted'
```

The response to this is:

```
on_the_list: True
good_guy: True
age: 21
nickname: Ted the Benevolent
```

We see that the service reacted appropriately to our request. More generally, service requests would be invoked from other ROS nodes.

2.2.1.3 An example ROS service client

To interact with a ROS service programmatically, one composes a ROS "client." An example is given in Listing 2.7 (`example_ros_client.cpp` in the package `example_ros_service`).

Listing 2.7: Example client node

```
1  //example ROS client:
2  // first run: rosrun example_ROS_service example_ROS_service
3  // then start this node:  rosrun example_ROS_service example_ROS_client
4
5
6
7  #include <ros/ros.h>
8  #include <example_ros_service/ExampleServiceMsg.h> // this message type is defined in ↩
       the current package
9  #include <iostream>
10 #include <string>
11 using namespace std;
12
13 int main(int argc, char **argv) {
14     ros::init(argc, argv, "example_ros_client");
15     ros::NodeHandle n;
16     ros::ServiceClient client = n.serviceClient<example_ros_service::ExampleServiceMsg↩
           >("lookup_by_name");
17     example_ros_service::ExampleServiceMsg srv;
18     bool found_on_list = false;
19     string in_name;
20     while (ros::ok()) {
21         cout<<endl;
22         cout << "enter a name (x to quit): ";
23         cin>>in_name;
24         if (in_name.compare("x")==0)
25             return 0;
26         //cout<<"you entered "<<in_name<<endl;
27         srv.request.name = in_name; //"Ted";
28         if (client.call(srv)) {
29             if (srv.response.on_the_list) {
30                 cout << srv.request.name << " is known as " << srv.response.nickname ↩
                       << endl;
31                 cout << "He is " << srv.response.age << " years old" << endl;
32                 if (srv.response.good_guy)
33                     cout << "He is reported to be a good guy" << endl;
34                 else
35                     cout << "Avoid him; he is not a good guy" << endl;
36             } else {
37                 cout << srv.request.name << " is not in my database" << endl;
38             }
39
40         } else {
41             ROS_ERROR("Failed to call service lookup_by_name");
42             return 1;
43         }
44     }
45     return 0;
46 }
47 %
```

In this program, we include the same message header as in the service node:

```
#include <example_ROS_service/ExampleServiceMsg.h>
```

Note that if this node were defined within another package, we would need to list the package dependency `example_ros_service` within the `package.xml` file.

There are two key lines in the client program. First, line 16:

```
    ros::ServiceClient client =
```

```
        n.serviceClient<example_ROS_service::ExampleServiceMsg>("lookup_by_name");
```

creates a ROS "ServiceClient." This service client expects to communicate requests and responses as defined in: `example_ros_service::ExampleServerMsg`. Also, this service client expects to communicate with a named service, called `lookup_by_name`. (This is the service name we had defined inside of our example service node).

Second, on line 17 we instantiate an object of a consistent type for requests and responses with:

```
example_ros_service::ExampleServiceMsg srv;
```

The above type specifies the package name in which the service message is defined (`example_ros_service`), and the name of the service message itself (`ExampleServiceMsg`). In this case "srv" contains both a field for request and a field for response. To send out a service request, we first populate the fields of the request message (in this case, the request message has only a single component), as per line 27:

```
srv.request.name = in_name; //e.g., manually test with contents: "Ted";
```

We then perform the transaction with the named service through the following call (line 28):

```
    client.call(srv)
```

This call will return a boolean to let us know if the call was successful or not. If the call is successful, then we may expect that the components of `srv.response` will be filled in (as provided by the ROS service). These components are examined and displayed by the example code. In this client example, the fields `on_the_list`, `name`, `age` and `good_guy` are evaluated, as shown in lines 31-33. E.g. line 31:

```
cout << "He is " << srv.response.age << " years old" << endl;
```

looks up and reports the `age` field of the service response message.

2.2.1.4 Running the example service and client

With `roscore` running, start the service with:

```
rosrun example_ros_service example_ros_service
```

The service displays "Ready to look up names," then it suspends action until a service requests comes in. Although the service is active, it consumes negligible resources (CPU cycles and bandwidth) while waiting for service requests.

In another terminal, start up the example client with:

```
rosrun example_ros_service example_ros_client
```

This results in prompting the user with `enter a name (x to quit):`. The following output resulted from responding with Ted, then Bob, Amy then x.

```
enter a name (x to quit): Ted
```

```
Ted is known as Ted the Benevolent
He is 21 years old
He is reported to be a good guy

enter a name (x to quit): Bob
Bob is known as BobTheTerrible
He is 32 years old
Avoid him; he is not a good guy

enter a name (x to quit): Amy
Amy is not in my database

enter a name (x to quit): x
```

While the client node is active, it can send repeated requests to the service, as shown above. The `ros::ServiceClient` only needs to be instantiated once, and `client.call(srv)` can be re-invoked with updated values in the `srv` message. Although the client may run to conclusion, the service would remain active (unless it is deliberately killed). Other client nodes may use the service as well, and such additional clients may run concurrently. The peer-to-peer communication scheme used in the service/client construction assures that the service will return its response only to the unique client that issued the corresponding request.

2.2.1.5 Conclusion

Using services can be appropriate for one-to-one, guaranteed communications. Although a sensor (e.g. a joint encoder) can simply keep publishing its current value, this would not be appropriate for a command such as `close_gripper`. In the latter case, we would want to send the command once and be assured that the command was received and acted on. A client/server interaction would be appropriate for such needs.

Some extra work is required to define the service message and to make sure both the server and the client include the corresponding message headers. ROS does include a package `std_srvs` with pre-defined service messages. However, unlike the fairly rich set of ROS messages for publish/subscribe use, there are only two service messages defined in this package, and they are of limited value. Typically, every time one composes a new ROS service node, it is also typically required that a corresponding service message be defined.

Services should be used only as quick request/response interactions. For interactions that can require long durations until a response is ready (e.g., `plan_path`), a more appropriate interface is action-servers/action-clients. Before introducing action servers, it is important to see how we can start to use C++ classes in constructing our nodes.

2.3 USING C++ CLASSES IN ROS

ROS code can quickly become overly long. To improve productivity and code re-use, it is desirable to use classes.

Use of classes in C++ is detailed in any text on C++. In general, it is desirable to:

- define a class in a header file
  - define prototypes for all member functions of the class
  - define prototypes for all member functions of the class
  - define a prototype for a class constructor function
- write a separate implementation file that:
  - includes the above header file
  - contains the working code for the declared member functions
  - contains code to encapsulate necessary initializations in the constructor
  - contains a main() function that instantiates an instance of the new class

An example is provided in the example code repository within the package: `example_ros_class`. The header file of this example is given in Listing 2.8:

Listing 2.8: Header file of `example_ros_class`

```
// example_ros_class.h header file //
// wsn; Feb, 2015
// include this file in "example_ros_class.cpp"

// here's a good trick--should always do this with header files:
// create a unique mnemonic for this header file, so it will get included if needed,
// but will not get included multiple times
#ifndef EXAMPLE_ROS_CLASS_H_
#define EXAMPLE_ROS_CLASS_H_

//some generically useful stuff to include...
#include <math.h>
#include <stdlib.h>
#include <string>
#include <vector>

#include <ros/ros.h> //ALWAYS need to include this

//message types used in this example code;  include more message types, as needed
#include <std_msgs/Bool.h>
#include <std_msgs/Float32.h>
#include <std_srvs/Trigger.h> // uses the "Trigger.srv" message defined in ROS

// define a class, including a constructor, member variables and member functions
class ExampleRosClass
{
public:
    ExampleRosClass(ros::NodeHandle* nodehandle); //"main" will need to instantiate a ROS nodehandle, then pass it to the constructor
    // may choose to define public methods or public variables, if desired
private:
    // put private member data here;  "private" data will only be available to member functions of this class;
    ros::NodeHandle nh_; // we will need this, to pass between "main" and constructor
    // some objects to support subscriber, service, and publisher
    ros::Subscriber minimal_subscriber_; //these will be set up within the class constructor, hiding these ugly details
    ros::ServiceServer minimal_service_;
    ros::Publisher  minimal_publisher_;

```

```
38     double val_from_subscriber_; //example member variable: better than using globals; convenient way to pass data from a subscriber to other member functions
39     double val_to_remember_; // member variables will retain their values even as callbacks come and go
40 
41     // member methods as well:
42     void initializeSubscribers(); // we will define some helper methods to encapsulate the gory details of initializing subscribers, publishers and services
43     void initializePublishers();
44     void initializeServices();
45 
46     void subscriberCallback(const std_msgs::Float32& message_holder); //prototype for callback of example subscriber
47     //prototype for callback for example service
48     bool serviceCallback(std_srvs::TriggerRequest& request, std_srvs::TriggerResponse& response);
49 }; // note: a class definition requires a semicolon at the end of the definition
50 
51 #endif  // this closes the header-include trick...ALWAYS need one of these to match #ifndef
```

The header file in Listing 2.8 defines the structure of a new class, `ExampleRosClass`. This class defines prototypes for a constructor (declared on line 28: `ExampleRosClass(ros::NodeHandle* nodehandle);`), and a variety of private member objects and functions, as well as declares member variables. These entities are all accessible by member functions of the class.

Defining objects for "publisher," "subscriber" and "service" allows set-up for these to be performed by the constructor, thus simplifying the main program.

Prototypes for the member functions are defined there, declaring the names, return types and argument types. The executable code constituting the "implementation" of these functions is contained in one or more separate `*.cpp` files. One should avoid putting implementation code in a header (except for very short implementations, such as "get()" and "set()" functions).

In the header file, private member objects and variables, e.g. `minimal_publisher_`, are named with a trailing underscore to provide a reminder to the user that these variables are only accessible to member functions within the class `ExampleRosClass`.

Note that the entire header file is contained within a compiler macro starting with `#ifndef EXAMPLE_ROS_CLASS_H_` and ending with `#endif`. This trick should always be used in writing header files. It helps the compiler to avoid including redundant copies of headers.

The implementation code corresponding to the above header file of Listing 2.8 is shown in Listing 2.9.

Listing 2.9: Implementation of `example_ros_class`

```
//example_ros_class.cpp:
//wsn, Jan 2016
//illustrates how to use classes to make ROS nodes
// constructor can do the initialization work, including setting up subscribers, publishers and services
// can use member variables to pass data from subscribers to other member functions

// can test this function manually with terminal commands, e.g. (in separate terminals):
// rosrun example_ros_class example_ros_class
// rostopic echo exampleMinimalPubTopic
// rostopic pub -r 4 exampleMinimalSubTopic std_msgs/Float32 2.0
// rosservice call exampleMinimalService 1


// this header incorporates all the necessary #include files and defines the class "ExampleRosClass"
#include "example_ros_class.h"

```

```
//CONSTRUCTOR:  this will get called whenever an instance of this class is created
// want to put all dirty work of initializations here
// odd syntax: have to pass nodehandle pointer into constructor for constructor to build subscribers, etc
ExampleRosClass::ExampleRosClass(ros::NodeHandle* nodehandle):nh_(*nodehandle)
{ // constructor
    ROS_INFO("in class constructor of ExampleRosClass");
    initializeSubscribers(); // package up the messy work of creating subscribers; do this overhead in constructor
    initializePublishers();
    initializeServices();

    //initialize variables here, as needed
    val_to_remember_=0.0;

    // can also do tests/waits to make sure all required services, topics, etc are alive
}

//member helper function to set up subscribers;
// note odd syntax: &ExampleRosClass::subscriberCallback is a pointer to a member function of ExampleRosClass
// "this" keyword is required, to refer to the current instance of ExampleRosClass
void ExampleRosClass::initializeSubscribers()
{
    ROS_INFO("Initializing Subscribers");
    minimal_subscriber_ = nh_.subscribe("example_class_input_topic", 1, &ExampleRosClass::subscriberCallback,this);
    // add more subscribers here, as needed
}

//member helper function to set up services:
// similar syntax to subscriber, required for setting up services outside of "main()"
void ExampleRosClass::initializeServices()
{
    ROS_INFO("Initializing Services");
    minimal_service_ = nh_.advertiseService("example_minimal_service",
                                                   &ExampleRosClass::serviceCallback,
                                                   this);
    // add more services here, as needed
}

//member helper function to set up publishers;
void ExampleRosClass::initializePublishers()
{
    ROS_INFO("Initializing Publishers");
    minimal_publisher_ = nh_.advertise<std_msgs::Float32>("example_class_output_topic", 1, true);
    //add more publishers, as needed
    // note: COULD make minimal_publisher_ a public member function, if want to use it within "main()"
}



// a simple callback function, used by the example subscriber.
// note, though, use of member variables and access to minimal_publisher_ (which is a member method)
void ExampleRosClass::subscriberCallback(const std_msgs::Float32& message_holder) {
    // the real work is done in this callback function
    // it wakes up every time a new message is published on "exampleMinimalSubTopic"

    val_from_subscriber_ = message_holder.data; // copy the received data into member variable, so ALL member funcs of ExampleRosClass can access it
    ROS_INFO("myCallback activated: received value %f",val_from_subscriber_);
    std_msgs::Float32 output_msg;
    val_to_remember_ += val_from_subscriber_; //can use a member variable to store values between calls; add incoming value each callback
    output_msg.data= val_to_remember_;
    // demo use of publisher--since publisher object is a member function
    minimal_publisher_.publish(output_msg); //output the square of the received value;
}


//member function implementation for a service callback function
```

```
82 bool ExampleRosClass::serviceCallback(std_srvs::TriggerRequest& request, std_srvs::TriggerResponse& response) {
83     ROS_INFO("service callback activated");
84     response.success = true; // boring, but valid response info
85     response.message = "here is a response string";
86     return true;
87 }
88
89
90
91 int main(int argc, char** argv)
92 {
93     // ROS set-ups:
94     ros::init(argc, argv, "exampleRosClass"); //node name
95
96     ros::NodeHandle nh; // create a node handle; need to pass this to the class constructor
97
98     ROS_INFO("main: instantiating an object of type ExampleRosClass");
99     ExampleRosClass exampleRosClass(&nh);  //instantiate an ExampleRosClass object and pass in pointer to nodehandle for constructor to use
100
101     ROS_INFO("main: going into spin; let the callbacks do all the work");
102     ros::spin();
103     return 0;
104 }
```

In Listing 2.9, the ROS service definition and the ROS publisher are similar to the examples in our `minimal_nodes` package introduced in Chapter 1. Note, though, that the subscriber is able to invoke the publisher, since the publisher is a member object that is accessible to all member functions. Further, the subscriber can copy its received data to member variables (`val_from_subscriber_` , in this example), making this data available to all member functions. In addition, the subscriber callback function illustrates using a member variable (`val_to_remember_`) to store results between calls, since this member variable persistently holds its data between calls to the subscriber. The member variables thus behave similarly to global variables, but these variables are only available to class member methods, and thus this construction is preferred.

The constructor is responsible for setting up the example publisher, example subscriber and example service. A somewhat odd notation is required, though. Main() must instantiate a node handle for the constructor to use, and this value must be passed to the constructor. But the constructor is responsible for initializing class variables, which creates a chicken-and-egg problem. This is resolved with the somewhat odd notation using constructor initializer lists (line 20):

```
ExampleRosClass::ExampleRosClass(ros::NodeHandle* nodehandle):nh_(*nodehandle)
```

This initialization technique allows the main program to create an instance of the class ExampleRosClass, passing into the constructor (a pointer to) the nodehandle created by main (line 98):

```
    ExampleRosClass exampleRosClass(&nh);
```

Note also the somewhat odd set-up of the subscriber and the service, which is performed within the constructor:

```
    minimal_subscriber_ = nh_.subscribe("example_class_input_topic", 1, &ExampleRosClass::subscriberCallback,this);
```

and

```
minimal_service_ = nh_.advertiseService("example_minimal_service",&ExampleRosClass::↩
    serviceCallback,this);
```

The notation: `&ExampleRosClass::subscriberCallback` provides a pointer to the callback function to be used with the subscriber. This callback function is defined as a member function of the current class. Additionally, the keyword "this" tells the compiler that we are referring to the current instance of this class.

Note that the callback function associated with the service uses a pre-defined service message (line 82):

```
bool ExampleRosClass::serviceCallback(std_srvs::TriggerRequest& request, std_srvs::↩
    TriggerResponse& response)
```

The service message "Trigger" is defined in `std_srvs`. To compile the example ROS class package, a dependency on `std_srvs` is listed in `package.xml`. For custom service messages (as with custom ROS messages defined generally), it can be convenient to put these into a separate package, as has been done with the ROS-provided `std_srvs` package.

Although the example class notation is somewhat cumbersome, it is convenient to design our ROS nodes in this fashion. The constructor is thus able to encapsulate the details of setting up publishers, subscribers and services, as well as initialize all important variables. The constructor may also test that required topics and services from companion nodes are active and healthy before releasing control to the "main" program.

In the present example, the main program is very short. It merely creates an instance of the new class, and then it goes into a "spin." All of the program work is then performed by callbacks of the new class object.

The example code can be tested (with `roscore` running) with the command:

```
rosrun example_ros_class example_ros_class
```

Upon start-up, this nodes displays the following output:

```
[ INFO] [1452372936.675150947]: main: instantiating an object of type ExampleRosClass
[ INFO] [1452372936.675272621]: in class constructor of ExampleRosClass
[ INFO] [1452372936.675323574]: Initializing Subscribers
[ INFO] [1452372936.682326226]: Initializing Publishers
[ INFO] [1452372936.683708338]: Initializing Services
[ INFO] [1452372936.685131573]: main: going into spin; let the callbacks do all the work
```

With this node running, one can peek/poke the various I/O options from a command line. From a separate terminal, enter:

```
rosservice call example_minimal_service
```

This terminal responds with:

```
success: True
message: here is a response string
```

And the terminal running `example_ros_class` displays:

```
service callback activated
```

This shows the expected behavior. When a client (in this case, invoked manually from the command line) sends a request to the service named `example_minimal_service`, the corresponding callback function within the node `example_ros_class` is awakened. Within the callback function (line 83), the callback function outputs the text "service callback activated." It then populates the fields of the response message (lines 84-85):

```
    response.success = true; // boring, but valid response info
    response.message = "here is a response string";
```

and these values are returned to the service client via the service response message. The received values are displayed by the (manually invoked) service client, appearing as:

```
success: True
message: here is a response string
```

The subscriber callback function within the `example_ros_class` node can also be tested manually. From a separate terminal, enter:

```
rostopic pub -r 2 example_class_input_topic  std_msgs/Float32 2.0
```

The `rostopic` command accepts arguments in YAML syntax (see `http://wiki.ros.org/ROS/YAMLCommandLine` for more detail). The option `-r 2` means that the publication will be repeated with a rate of 2Hz. After running this command, the output in terminal of node `example_ros_class` appears as:

```
[ INFO] [1452374662.370650465]: myCallback activated: received value 2.000000
[ INFO] [1452374662.870560823]: myCallback activated: received value 2.000000
[ INFO] [1452374663.370577645]: myCallback activated: received value 2.000000
```

With this minimal example, it is shown how to incorporate ROS publishers, subscribers and services within a C++ class. This example ROS class can be used as a starting point to create new ROS classes. Starting with this example, one would rename the class and its services and topics. More services, topics, subscribers and publishers may be added, as desired. Additional member methods can be declared in the header and implementations provided in the main cpp file. Using classes helps to encapsulate some of the tedious "boilerplate" of ROS, allowing the programmer to focus on algorithms. Additionally, by using classes, one can create "libraries" that promote ease of code re-use–which is introduced next.

2.4 CREATING LIBRARY MODULES IN ROS

So far, we have created independent packages that take advantage of existing libraries. However, as one's source code gets lengthier, it is desirable to break it up into smaller modules. If work may be re-used by future modules, then it is best to create a new "library." This section describes the steps to creating a library. It will refer to the examples in the accompanying code repository within two separate packages: `creating_a_ros_library` and `using_a_ros_library`.

The package `creating_a_ros_library` was created using:

```
cs_create_pkg creating_a_ros_library roscpp std_msgs std_srvs
```

This sets up the expected package structure, including folders for `src` and `include` and files `CMakeLists.txt`, `package.xml` and `README.md`.

From the previous section, we borrow code from the package `example_ros_class`. The implementation source file `example_ros_class.cpp` is copied to the `src` subdirectory of the new package `creating_a_ros_library`. This code is edited to remove the main() program. Also, the header file inclusion is modified to refer to:

```
#include <creating_a_ros_library/example_ros_class.h>
```

The source code file is otherwise identical to Listing 2.9. For completeness, the edited file is shown in 2.10.

Listing 2.10: Implementation of library `example_ros_class`

```

//example_ros_class.cpp:
//wsn, Jan 2016
//illustrates how to use classes to make ROS nodes
// constructor can do the initialization work, including setting up subscribers, publishers and services
// can use member variables to pass data from subscribers to other member functions

// can test this function manually with terminal commands, e.g. (in separate terminals):
// rosrun example_ros_class example_ros_class
// rostopic echo exampleMinimalPubTopic
// rostopic pub -r 4 exampleMinimalSubTopic std_msgs/Float32 2.0
// rosservice call exampleMinimalService 1


// this header incorporates all the necessary #include files and defines the class "ExampleRosClass"
#include <creating_a_ros_library/example_ros_class.h>

//CONSTRUCTOR:  this will get called whenever an instance of this class is created
// want to put all dirty work of initializations here
// odd syntax: have to pass nodehandle pointer into constructor for constructor to build subscribers, etc
ExampleRosClass::ExampleRosClass(ros::NodeHandle* nodehandle):nh_(*nodehandle)
{ // constructor
    ROS_INFO("in class constructor of ExampleRosClass");
    initializeSubscribers(); // package up the messy work of creating subscribers; do this overhead in constructor
    initializePublishers();
    initializeServices();

    //initialize variables here, as needed
    val_to_remember_=0.0;

    // can also do tests/waits to make sure all required services, topics, etc are alive
```

```
}

//member helper function to set up subscribers;
// note odd syntax: &ExampleRosClass::subscriberCallback is a pointer to a member function of ExampleRosClass
// "this" keyword is required, to refer to the current instance of ExampleRosClass
void ExampleRosClass::initializeSubscribers()
{
    ROS_INFO("Initializing Subscribers");
    minimal_subscriber_ = nh_.subscribe("example_class_input_topic", 1, &ExampleRosClass::subscriberCallback,this);
    // add more subscribers here, as needed
}

//member helper function to set up services:
// similar syntax to subscriber, required for setting up services outside of "main()"
void ExampleRosClass::initializeServices()
{
    ROS_INFO("Initializing Services");
    minimal_service_ = nh_.advertiseService("example_minimal_service",
                                              &ExampleRosClass::serviceCallback,
                                              this);
    // add more services here, as needed
}

//member helper function to set up publishers;
void ExampleRosClass::initializePublishers()
{
    ROS_INFO("Initializing Publishers");
    minimal_publisher_ = nh_.advertise<std_msgs::Float32>("example_class_output_topic", 1, true);
    //add more publishers, as needed
    // note: COULD make minimal_publisher_ a public member function, if want to use it within "main()"
}



// a simple callback function, used by the example subscriber.
// note, though, use of member variables and access to minimal_publisher_ (which is a member method)
void ExampleRosClass::subscriberCallback(const std_msgs::Float32& message_holder) {
    // the real work is done in this callback function
    // it wakes up every time a new message is published on "exampleMinimalSubTopic"

    val_from_subscriber_ = message_holder.data; // copy the received data into member variable, so ALL member funcs of ExampleRosClass can access it
    ROS_INFO("myCallback activated: received value %f",val_from_subscriber_);
    std_msgs::Float32 output_msg;
    val_to_remember_ += val_from_subscriber_; //can use a member variable to store values between calls; add incoming value each callback
    output_msg.data= val_to_remember_;
    // demo use of publisher--since publisher object is a member function
    minimal_publisher_.publish(output_msg); //output the square of the received value;
}


//member function implementation for a service callback function
bool ExampleRosClass::serviceCallback(std_srvs::TriggerRequest& request, std_srvs::TriggerResponse& response) {
    ROS_INFO("service callback activated");
    response.success = true; // boring, but valid response info
    response.message = "here is a response string";
    return true;
}
```

The header file with the class prototype needs to reside in a location where `catkin_make` can find it when it is desired to link our library to nodes in new packages. By convention, the header file is located under the `/include` directory–but not directly. Instead, a subdirectory of the `/include` directory is created (with the same name as the package) and the library's header file is placed here, i.e. in : `.../creating_a_ros_library/include/creating_a_` ros_library. The original header file, example_ros_class.h, is copied to this directory. Subsequently, nodes may import this header file by including the line:

```
#include <creating_a_ros_library/example_ros_class.h>
```

(the same as has been done with the library itself).

The CMakeLists.txt file is edited to include (or uncomment/edit) the line:

```
cs_add_library(example_ros_library src/example_ros_class.cpp)
```

This informs catkin_make that a new library is to be created, that this new library will be named example_ros_library, and the source code for this library resides in src/example_ros_class.cpp.

At this point, the library can be compiled by running catkin_make (from the ros_ws directory). After compilation, a new file appears in the directory ~/ros_ws/devel/lib, called libexample_ros_library.so. The base name, example_ros_library.so, follows from the name assignment in cs_add_library, and the build system prepends the name "lib." However, specifying linking to this library does not require knowledge of this new name. It is only necessary to note the package dependency in package.xml and to include the associated header file with #include <creating_a_ros_library/example_ros_class.h>.

To see if our new library is working, we can create a test "main" program in the same package. The file example_ros_class_test_main.cpp, shown in Listing 2.11.

Listing 2.11: Implementation of library example_ros_class

```
1  #include <creating_a_ros_library/example_ros_class.h>
2  
3  int main(int argc, char** argv)
4  {
5      // ROS set-ups:
6      ros::init(argc, argv, "example_lib_test_main"); //node name
7  
8      ros::NodeHandle nh; // create a node handle; need to pass this to the class constructor
9  
10     ROS_INFO("main: instantiating an object of type ExampleRosClass");
11     ExampleRosClass exampleRosClass(&nh);  //instantiate an ExampleRosClass object and pass in pointer to nodehandle for constructor to use
12 
13     ROS_INFO("main: going into spin; let the callbacks do all the work");
14     ros::spin();
15     return 0;
16 }
```

This test program is identical to the corresponding main() function in example_ros_class/src/example_ros_class.cpp, except that the header file is included as: #include <creating_a_ros_library/example_ros_class.h> (line 1).

To compile the test main, two changes to CMakeLists.txt are required. The first is to specify compilation of a new executable, using the line:

```
cs_add_executable(ros_library_test_main src/example_ros_class_test_main.cpp)
```

The second change specifies that our new node should be linked with our new library, which requires the line:

```
target_link_libraries(ros_library_test_main example_ros_library)
```

This linker command references our executable-file name, `ros_library_test_main`, as the first argument and our new library name, `example_ros_library`, as the second argument.

We can run our new test node with the command:

```
rosrun creating_a_ros_library example_ros_class_test_main
```

which, as usual, references our package name (`creating_a_ros_library`) and the name of an executable in that package (`example_ros_class_test_main`). The result is an output identical (except for the time stamps) to that obtained in the previous section:

```
[ INFO] [1452395548.594901501]: main: instantiating an object of type ExampleRosClass
[ INFO] [1452395548.595099162]: in class constructor of ExampleRosClass
[ INFO] [1452395548.595126811]: Initializing Subscribers
[ INFO] [1452395548.599454035]: Initializing Publishers
[ INFO] [1452395548.600454593]: Initializing Services
[ INFO] [1452395548.601531005]: main: going into spin; let the callbacks do all the work
```

Our example main, `example_ros_class_test_main`, shows how to use our new library within a node. Typically, however, new code will need to reference a library that exists in a separate package. Doing so is quite simple, as illustrated with the separate package `using_a_ros_library`. This package (included in the accompanying code respository) was created with:

```
cs_create_pkg using_a_ros_library roscpp std_msgs std_srvs creating_a_ros_library
```

Note that the above use of `cs_create_pkg` declares a dependency on the package `creating_a_ros_library`. This was not done when creating the package `creating_a_ros_library` (since this would have been circular, attempting to depend on a package that is the package being created). Because of this, the `CMakeLists.txt` file required the linker line `target_link_libraries(ros_library_test_main example_ros_library)` to note that the test main should be linked with the library created in the same package.

The test-main program `example_ros_class_test_main.cpp` was copied from the `src` folder of the package `creating_a_ros_library` to the `src` folder of the new package `using_a_ros_library`. No changes were made to the test-main source file. To compile this new node, the `CMakeLists.txt` file was edited to include the line:

```
cs_add_executable(ros_library_external_test_main src/example_ros_class_test_main.cpp)
```

However, using `catkin_simple`, it is not necessary to include a command to link with the new library. Since the `package.xml` file already notes a dependency on the package `creating_a_ros_library`, and since the source code includes the header file `#include <creating_a_ros_library/example_ros_class.h>`, `catkin_simple` recognizes that the executable should be linked with the library `example_ros_library` from package `creating_a_ros_library`. Using `catkin_make` without `catkin_simple` would require editing several additional lines of `CMakeLists.txt` to declare header locations, libraries to link in, and (in more general cases) making of custom messages.

Our new test node can be run with:

```
rosrun using_a_ros_library ros_library_external_test_main
```

The output and the behavior are identical to the corresponding cases in packages `example_ros_class` and `creating_a_ros_library`.

This above explanation illustrates how to create a ROS library. As one's code becomes more complex, using libraries is increasingly valuable for encapsulating detail and re-using software.

2.5 INTRODUCTION TO ACTION SERVERS AND ACTION CLIENTS

We have seen ROS communications mechanisms of publish/subscribe and service/client. A third important communications paradigm in ROS is the action-server/action-client pattern. The action-server/action-client approach is similar to service/client communications, in that there is peer-to-peer communications between the service and the client. The client always knows the recipient (service) of the client's requests, and the service always knows the client to which it should respond. A limitation of the service/client approach is that the client "blocks" waiting on the service to respond. If the desired behavior requires a long period of time (e.g., `clear_the_table`), then it is desirable that the client can continue to run other important tasks (e.g. `check_the_battery_voltage`) while the requested behavior is being performed. Action-servers/action-clients fill this role.

There are many options and variations in construction action servers and action clients; only simple examples are treated here. Further details can be found on-line at: `http://wiki.ros.org/actionlib`. Example code corresponding to the present introduction can be found in the accompanying repository within the package `example_action_server`.

From `http://wiki.ros.org/actionlib`:

> In any large ROS based system, there are cases when someone would like to send a request to a node to perform some task, and also receive a reply to the request. This can currently be achieved via ROS services.
>
> In some cases, however, if the service takes a long time to execute, the user might want the ability to cancel the request during execution or get periodic feedback about how the request is progressing. The actionlib package provides tools to create servers that execute long-running goals that can be preempted. It also provides a client interface in order to send requests to the server.

A common use of an action server is for execution of a pre-planned trajectory. If one has computed a sequence of joint-space poses to be realized at specified time steps, this entire message can be delivered to an action server for execution. (This is the approach used in ROS-Industrial, to transmit desired trajectories to industrial robots, where the trajectory is subsquently executed using the native controller). Designing with action servers allows one to exploit "SMACH" (see `http://wiki.ros.org/smach`) for higher-level state-machine programming, as well as alternative decision-making coordination packages.

2.5.1 Creating an action-server package

A new action-server package may be created by navigating to `ros_ws/src` (or some subdirectory from here) and (using the `catkin_simple` option) invoking:

```
cs_create_pkg example_action_server roscpp actionlib
```

This has already been done in the accompanying example-code repository (see `https://github.com/wsnewman/learning_ros/tree/master/Part_1/example_action_server`). The package `example_action_server` uses the library `actionlib`.

Invoking "`cs_create_pkg`" (or the long version of CMakeLists.txt, via "`catkin_create_pkg`") does much of the preparatory work for us, including establishing a `package.xml` file, a `CmakeLists.txt` file and subdirectories `include` and `src`. However, we will need an additional subdirectory to define a custom "action message." Creation of an action message is similar to creation of a service message. Navigate to within the new package directory and invoke:

```
mkdir action
```

We will use this directory to define our communications message format between our new server and its future clients. To get the new action message pre-processed by `catkin_make`, it is also necessary to edit the package.xml file (as was done previously for ".msg" and ".srv" message creation) and uncomment the lines:

```
    <build_depend>message_generation</build_depend>
```

and

```
    <run_depend>message_runtime</run_depend>
```

This will result in auto-generating various header files to define message types for action client/server communications.

2.5.2 Defining custom action-server messages

We have seen previously how to define custom messages for publish/subscribe communications, as well as custom service messages for client/service communications. The required steps included creating a corresponding folder in the package directory (`msg` or `srv` for messages and service messages, respectively) and composing a simple text file describing the format of the desired messages (`*.msg` or `*.srv` files, respectively). Service messages are slightly more complex than simple messages, since service messages require specification of both a "request" and a "response."

Action-server/action-client messages are created similarly. Within the "action" subdirectory of a package, create a new file with the suffix ".action." Like service messages, action messages have multiple regions, with three regions for action messages (vs. two regions for service messages). These regions are the "goal","result" and "feedback" components.

In our example, the "action" folder contains the file "demo.action" with the following text:

```
#goal definition
#the lines with the hash signs are merely comments
#goal, result and feedback are defined by this fixed order, and separated by 3 hyphens
int32 input
---
#result definition
int32 output
int32 goal_stamp
---
#feedback
int32 fdbk
```

In the above, we have prescribed three fields: `goal`, `result` and `feedback`. The goal definition, in this simple case, contains only a single component, called "input", which is of type `int32`. Note that the "#" sign is a comment delimiter. The labels for `goal`, `result` and `feedback` are just reminders. Message generation will ignore these comments and will assume that there are three fields, in this fixed order (goal, result, feedback), separated by three dashes.

Following three dashes, the "result" message is defined. In this example, the `result` definition consists of two components: `output` and `goal_stamp`, both of which are of type int32. The final definition, "feedback", is also separated by three dashes, and the example contains only one field, called `fdbk`, also of type `int32`.

An action message must be written in the above format. The dashes and the order are important; the comments are optional, but helpful.

The components defined in these fields can be comprised of any existing message definitions, provided the corresponding message packages have been named in the `package.xml` file and the corresponding headers are included in the source code file (to be composed, as described below).

Because our `package.xml` file has `message_generation` enabled, once we perform `catkin_make` on our new package, the build system will create multiple new *.h header files, although this is not obvious. These files will be located in `~/ros_ws/devel/include` within a subdirectory corresponding to our package name, `example_action_server`. Within this directory, there will be seven *.h files created, each with a name that starts with `demo` (the name we chose for our "action" message specification). Six of these are included within the seventh header file, `demoAction.h`. By including this composite header in action node code, one may then refer to messages types such as "demoGoal" and "demoResult."

The example action server source code, `example_action_server.cpp`, is shown in Listing 2.12.

Listing 2.12: `example_action_server`

```
// example_action_server: a simple action server
// this version does not depend on actionlib_servers hku package
// wsn, October, 2014

#include<ros/ros.h>
#include <actionlib/server/simple_action_server.h>
//the following #include refers to the "action" message defined for this package
// The action message can be found in: .../example_action_server/action/demo.action
// Automated header generation creates multiple headers for message I/O
// These are referred to by the root name (demo) and appended name (Action)
#include<example_action_server/demoAction.h>

int g_count = 0;
bool g_count_failure = false;

class ExampleActionServer {
private:

    ros::NodeHandle nh_;  // we'll need a node handle; get one upon instantiation

    // this class will own a "SimpleActionServer" called "as_".
    // it will communicate using messages defined in example_action_server/action/demo.action
    // the type "demoAction" is auto-generated from our name "demo" and generic name "Action"
    actionlib::SimpleActionServer<example_action_server::demoAction> as_;

    // here are some message types to communicate with our client(s)
    example_action_server::demoGoal goal_; // goal message, received from client
    example_action_server::demoResult result_; // put results here, to be sent back to the client when done w/ goal
    example_action_server::demoFeedback feedback_; // not used in this example;
    // would need to use: as_.publishFeedback(feedback_); to send incremental feedback to the client



public:
    ExampleActionServer(); //define the body of the constructor outside of class definition

     ExampleActionServer(void) {
    }
```

```
    // Action Interface
    void executeCB(const actionlib::SimpleActionServer<example_action_server::demoAction>::GoalConstPtr& goal);
};

//implementation of the constructor:
// member initialization list describes how to initialize member as_
// member as_ will get instantiated with specified node-handle, name by which this server will be known,
//  a pointer to the function to be executed upon receipt of a goal.
//
// Syntax of naming the function to be invoked: get a pointer to the function, called executeCB, which is a member method
// of our class exampleActionServer.  Since this is a class method, we need to tell boost::bind that it is a class member,
// using the "this" keyword.  the _1 argument says that our executeCB takes one argument
// the final argument  "false" says don't start the server yet.  (We'll do this in the constructor)

ExampleActionServer::ExampleActionServer() :
   as_(nh_, "example_action", boost::bind(&ExampleActionServer::executeCB, this, _1),false)
// in the above initialization, we name the server "example_action"
//  clients will need to refer to this name to connect with this server
{
    ROS_INFO("in constructor of exampleActionServer...");
    // do any other desired initializations here...specific to your implementation

    as_.start(); //start the server running
}

//executeCB implementation: this is a member method that will get registered with the action server
// argument type is very long.  Meaning:
// actionlib is the package for action servers
// SimpleActionServer is a templated class in this package (defined in the "actionlib" ROS package)
// <example_action_server::demoAction> customizes the simple action server to use our own "action" message
// defined in our package, "example_action_server", in the subdirectory "action", called "demo.action"
// The name "demo" is prepended to other message types created automatically during compilation.
// e.g.,  "demoAction" is auto-generated from (our) base name "demo" and generic name "Action"
void ExampleActionServer::executeCB(const actionlib::SimpleActionServer<example_action_server::demoAction>::GoalConstPtr& goal) {
    //ROS_INFO("in executeCB");
    //ROS_INFO("goal input is: %d", goal->input);
    //do work here: this is where your interesting code goes

    //....

    // for illustration, populate the "result" message with two numbers:
    // the "input" is the message count, copied from goal->input (as sent by the client)
    // the "goal_stamp" is the server's count of how many goals it has serviced so far
    // if there is only one client, and if it is never restarted, then these two numbers SHOULD be identical...
    // unless some communication got dropped, indicating an error
    // send the result message back with the status of "success"

    g_count++; // keep track of total number of goals serviced since this server was started
    result_.output = g_count; // we'll use the member variable result_, defined in our class
    result_.goal_stamp = goal->input;

    // the class owns the action server, so we can use its member methods here

    // DEBUG: if client and server remain in sync, all is well--else whine and complain and quit
    // NOTE: this is NOT generically useful code; server should be happy to accept new clients at any time, and
    // no client should need to know how many goals the server has serviced to date
```

```
95      if (g_count != goal->input) {
96          ROS_WARN("hey--mismatch!");
97          ROS_INFO("g_count = %d; goal_stamp = %d", g_count, result_.goal_stamp);
98          g_count_failure = true; //set a flag to commit suicide
99          ROS_WARN("informing client of aborted goal");
100         as_.setAborted(result_); // tell the client we have given up on this goal; send the result message as well
101     }
102     else {
103          as_.setSucceeded(result_); // tell the client that we were successful acting on the request, and return the "result" message
104     }
105 }
106
107 int main(int argc, char** argv) {
108     ros::init(argc, argv, "demo_action_server_node"); // name this node
109
110     ROS_INFO("instantiating the demo action server: ");
111
112     ExampleActionServer as_object; // create an instance of the class "ExampleActionServer"
113
114     ROS_INFO("going into spin");
115     // from here, all the work is done in the action server, with the interesting stuff done within "executeCB()"
116     // you will see 5 new topics under example_action: cancel, feedback, goal, result, status
117     while (!g_count_failure) {
118         ros::spinOnce(); //normally, can simply do: ros::spin();
119         // for debug, induce a halt if we ever get our client/server communications out of sync
120     }
121
122     return 0;
123 }
```

In Listing 2.12, line 6:

```
#include <actionlib/server/simple_action_server.h>
```

brings in the header file associated with the `simple_action_server` package, which is necessary to use this library. Line 11:

```
#include<example_action_server/demoAction.h>
```

brings in the action message descriptions, which are auto-generated header files created by `catkin_make`, as prescribed in our `action` subdirectory of our new package, `example_action_server`.

Line 16,

```
class ExampleActionServer {
```

begins the definition of class "ExampleActionServer." This class includes an object from the `SimpleActionServer` class defined in the `actionlib` library. This object, given the name `as_`, is declared on line 24:

```
actionlib::SimpleActionServer<example_action_server::demoAction> as_;
```

The prototype for class `ExampleActionServer` also incorporates three message objects: `goal_`, `result_` and `feedback_`, declared in lines 27-29:

```
    example_action_server::demoGoal goal_; // goal message, received from client
```

```
  example_action_server::demoResult result_; // put results here, to be sent back to↩
      the client when done w/ goal
  example_action_server::demoFeedback feedback_; // not used in this example;
```

Both our client and server code will need to refer to these new messages to communicate with each other. To further illuminate, line 28 of the example C++ code instantiates a variable called `result_` that is of type `demoResult` as defined in the package `example_action_server`. Subsequently, we may refer to components of `result_`, such as in the line 87:

```
result_.output = g_count;
```

When the server returns the goal message to its client, the value assigned to `result_.output` will be received by the client (as well as other fields of the result that are populated by the server).

Our class `ExampleActionServer` has a constructor, declared on line 35, the body of which is implemented in lines 53-62.

The most important component of the class class `ExampleActionServer` is the callback function, the prototype for which appears on line 40.

```
void executeCB(const actionlib::SimpleActionServer<example_action_server::demoAction↩
    >::GoalConstPtr& goal);
```

This callback function will be associated with the SimpleActionServer object, `as_`. The implemenation of the callback function (lines 72-105, in this example) contains the heart of the code that will get executed to perform requested services. The prototype for this callback function is fairly long-winded. The method name `executeCB` is our own, arbitrary choice. The argument of this function is a pointer to a goal message. Declaration of the goal message refers to the actionlib library, the templated class `SimpleActionServer`, and our own action message, referred to by `example_action_server::demoAction`, which further has an auto-generated type `GoalConstPtr`. This syntax is tedious. However, the example may be copied verbtim and the programmer may change the specific function, package and message names, as desired.

The constructor implementation, which starts at line 53, is also fairly cryptic:

```
ExampleActionServer::ExampleActionServer() :
   as_(nh_, "example_action", boost::bind(&ExampleActionServer::executeCB, this, _1),↩
       false)
```

The purpose of this line is to initialize the object `as_`. The initializer arguments specify that the new action server will be known to the ROS system by the name `example_action`.

Additionally, we wish to affiliate a callback function with the action server. Our class `ExampleActionServer` has a member method `executeCB` that we wish to use. This is accomplished in initialization of `as_` using `boost::bind`. In the arguments of `boost::bind` we specify that the callback function `executeCB` defined within the namespace of class "ExampleActionServer" is to be used. The keyword "this" indicates that `executeCB` is a member of the current object. The argument "`_1`" is used by `boost::bind` to tell the simple action server object that our defined callback function takes one argument.

The final initialization argument, "false", tells the constructor that we do not yet wish to start the new action server running. Instead, our action server is started running by line 61 in the constructor:

```
as_.start(); //start the server running
```

Action-server start-up is deferred to this point to avoid a race condition, assuring that initialization of the server is completed before it is started running.

The "main()" function is composed in lines 107-123. Line 112:

```
ExampleActionServer as_object; // create an instance of the class "↩
    ExampleActionServer"
```

instantiates an object of your new class `ExampleActionServer`. As is the case with simpler ROS services, the main program can now simply go into a "spin()", letting the callback function of the new action server do all the work. Also like ROS services, this action-server node consumes negligle CPU or bandwidth resources while waiting for action-client goal requests.

In the callback function `executeCB()`, the code may refer to the information contained within the goal message. Contents of the goal message is what is transmitted by an action client to the action server. The callback function may have information to be returned to the client; such information should returned to the client by populating the expected fields of `result_`.

The simple example here does not illustrate how to send feedback messages to the client. (for further details, consult the on-line ROS tutorials). The purpose of the feedback message is to provide the client with status reports while goal execution is in progress. Feedback messages are not required, but they are often helpful.

Once the important work within `executeCB` has been performed, this function must conclude by invoking either `as_.setAborted(result_)` or `as_.setSucceeded(result_)`. In either case, providing the argument `result_` causes these member methods to transmit the message `result_` back to the client. Additionally, the client is informed whether the server concluded successfully or aborted.

In the minimal example provided, the action server merely does a simple communications diagnostic (as implemented in the callback function). Within `executeCB()`, the server keeps track of how many times it has serviced goals, and it copies this value into the `output` field of the result message (lines 86-87):

```
g_count++;
result_.output = g_count;
```

Also, on line 88, the `input` field of the goal message is copied to the `goal_stamp` field of the `result_` message.

```
result_.goal_stamp = goal->input;
```

The client will receive back the values in these fields of the `result_` message.

For this simple, diagnostic server, if the server's record of number of goals achieved differs from the client's input, the server outputs an error message and shuts down (lines 95-101). Otherwise, the server reports success and returns the result message to the client (line 103):

```
as_.setSucceeded(result_);
```

In designing a new action server, the preceding complexity largely can be ignored by copying/pasting the provided example code. The heart of the server is contained within the implementation of "executeCB," and this is where new code should be composed for new server behaviors.

Aside from the callback function implementation, designing of a new action server can re-use the example code here, provided necessary name changes are performed consistently. The new action server would reside in a new package, and this package name should replace each instance of `example_action_server` in the example code. Also, a more mnemonic class name should be chosen, and this new class name should be substituted for each occurrence of `ExampleActionServer`.

A mnemonic action message appropriate for the new action server should be created, and the base name of this action message should be substituted everywhere for the example name "demo" (e.g., change the name `::demoAction`, etc to use the new action message base name).

The variable names `goal_`, `result_` and `feedback_` can be changed, but these names should be valid to stay as-is. The callback function name, `executeCB`, may be changed if desired, but the example name could be retained without confusion.

Importantly, the server name (currently `example_action`) should be changed to a name that is meaningful for the new server. Finally, the node name in "main" (`demo_action_server_node`) should also be changed to something relevant and mnemonic.

2.5.3 Designing an action client

A compatible client of our new action server also resides in the same example package, `example_action_server`, with source file called `example_action_client.cpp`. The code is shown in Listing 2.13.

Listing 2.13: `example_action_client`

```cpp
// example_action_client:
// wsn, October, 2014

#include<ros/ros.h>
#include <actionlib/client/simple_action_client.h>

//this #include refers to the new "action" message defined for this package
// the action message can be found in: .../example_action_server/action/demo.action
// automated header generation creates multiple headers for message I/O
// these are referred to by the root name (demo) and appended name (Action)
// If you write a new client of the server in this package, you will need to include example_action_server in your package.xml,
// and include the header file below
#include<example_action_server/demoAction.h>


// This function will be called once when the goal completes
// this is optional, but it is a convenient way to get access to the "result" message sent by the server
void doneCb(const actionlib::SimpleClientGoalState& state,
        const example_action_server::demoResultConstPtr& result) {
    ROS_INFO(" doneCb: server responded with state [%s]", state.toString().c_str());
    int diff = result->output - result->goal_stamp;
    ROS_INFO("got result output = %d; goal_stamp = %d; diff = %d",result->output,result->goal_stamp,diff);
}

int main(int argc, char** argv) {
        ros::init(argc, argv, "demo_action_client_node"); // name this node
        int g_count = 0;
        // here is a "goal" object compatible with the server, as defined in example_action_server/action
        example_action_server::demoGoal goal;

        // use the name of our server, which is: example_action (named in example_action_server.cpp)
        // the "true" argument says that we want our new client to run as a separate thread (a good idea)
```

```
33         actionlib::SimpleActionClient<example_action_server::demoAction> action_client("example_action", true);
34 
35         // attempt to connect to the server:
36         ROS_INFO("waiting for server: ");
37         bool server_exists = action_client.waitForServer(ros::Duration(5.0)); // wait for up to 5 seconds
38         // something odd in above: does not seem to wait for 5 seconds, but returns rapidly if server not running
39         //bool server_exists = action_client.waitForServer(); //wait forever
40 
41         if (!server_exists) {
42             ROS_WARN("could not connect to server; halting");
43             return 0; // bail out; optionally, could print a warning message and retry
44         }
45 
46 
47         ROS_INFO("connected to action server");  // if here, then we connected to the server;
48 
49         while(true) {
50         // stuff a goal message:
51         g_count++;
52         goal.input = g_count; // this merely sequentially numbers the goals sent
53         //action_client.sendGoal(goal); // simple example--send goal, but do not specify callbacks
54         action_client.sendGoal(goal,&doneCb); // we could also name additional callback functions here, if desired
55         //    action_client.sendGoal(goal, &doneCb, &activeCb, &feedbackCb); //e.g., like this
56 
57         bool finished_before_timeout = action_client.waitForResult(ros::Duration(5.0));
58         //bool finished_before_timeout = action_client.waitForResult(); // wait forever...
59         if (!finished_before_timeout) {
60             ROS_WARN("giving up waiting on result for goal number %d",g_count);
61             return 0;
62         }
63         else {
64           //if here, then server returned a result to us
65         }
66 
67         }
68 
69     return 0;
70 }
```

Like the action server, the action client must bring in a header file from the actionlib library, specifically for the `simple_action_client` (line 5)

```
#include <actionlib/client/simple_action_client.h>
```

The action client (like the action server) must also refer to the custom action message, defined in the current package (code line 13):

```
#include<example_action_server/demoAction.h>
```

Often, different client programs may use the same server, and these clients might be defined in separate packages. In that case, it is necessary to specify in `package.xml` a dependency on the package that contains the action file that defines the action message to be used (and the corresponding header file must be included in the client's source code).

The main program of our action client (starting on line 25) instantiates an object of type `example_action_server::demoGoal` on line 29:

```
example_action_server::demoGoal goal;
```

This object is used to communicate "goals" to the action server.

An object called `action_client` is instantiated from the (templated) class `actionlib::SimpleActionClient`. We specialize this object to use our defined action messages with the template specification (line 33):

```
actionlib::SimpleActionClient<example_action_server::demoAction> action_client("↩
    example_action", true);
```

The new action client is constructed with the argument `example_action`. This is the name chosen in the design of our action server, as was specified in the class constructor:

```
exampleActionServer::exampleActionServer() :
as_(nh_, "example_action", boost::bind(&exampleActionServer::executeCB, this, _1),↩
    false)
```

Action-client nodes (like servce clients) need to know the name of their respective server in order to connect.

One of the member functions of an action client is "waitForServer()". Line 37:

```
bool server_exists = action_client.waitForServer(ros::Duration(5.0)); // wait up to 5 ↩
    sec
```

causes the action client to attempt to connect to the named server. The format allows for waiting only up to some time limit, or waiting indefinitely (if the duration argument is omitted). This method returns "true" if successful connection is made.

In the example program, the main program keeps a count of the number of goals sent to the action server, and the goal message has its "input" field filled with the current iteration count (lines 51-52):

```
        g_count++;
        goal.input = g_count;
```

The server is requested to perform its service, as specified in the "goal" message, by invoking the member function (line 54):

```
action_client.sendGoal(goal,&doneCb);
```

This form of "sendGoal" includes reference to a callback function (arbitrarily) named "doneCb".

The client then invokes (line 57):

```
bool finished_before_timeout =
action_client.waitForResult(ros::Duration(5.0));
```

which causes the client to suspend, waiting on the server, but with a specified time-out limit. If the server returns within the specified time limit, the `goalCB` function will be triggered. This function receives the "result" message provided by the server.

In the example code, the callback function (lines 18-23) compares the number of goals serviced by the server so far to the number of goals this client has requested. It prints out the difference between the two. If only this client requests goals from the server, and if the client and server are both started in the same session, then we should expect that the value of "diff" is always zero, since the number of goals served by the server and the number of goals requested by this client should be identical. However, if either the client or server are stopped and restarted (or if a second client is started), there will be a count mismatch.

This client example may be re-used for desiging new action clients. For a new client package, the client code must refer to the respective server's package name, its respective server name, and its corresponding action file. In place of all instances of `example_action_server`, substitute the new client's package name. For all occurrences of action message name `demo`, substitute the new action message name. The new client's node name (within `ros::init()` on line 26) should be changed to something appropriate, mnemonic, and unique. In instantiating the action client (line 33), the name of the desired action server should be substituted for `example_action`. Most importantly, the `doneCb()` function (which may or may not be renamed) should contain someuseful application code that accomplishes your objectives.

2.5.4 Running the example code

To run the example code, first make sure there is a `roscore` running. The server and client can then be started in either order with the following commands in separate terminals:

```
rosrun example_action_server example_action_server
```

and

```
rosrun example_action_server example_action_client
```

The output of the action client looks like:

```
[ INFO] [1452388037.084421813]: got result output = 21756; goal_stamp = 21756; diff = 0
[ INFO] [1452388037.086694318]:  doneCb: server responded with state [SUCCEEDED]
[ INFO] [1452388037.086733133]: got result output = 21757; goal_stamp = 21757; diff = 0
[ INFO] [1452388037.088981490]:  doneCb: server responded with state [SUCCEEDED]
[ INFO] [1452388037.089020387]: got result output = 21758; goal_stamp = 21758; diff = 0
[ INFO] [1452388037.091277514]:  doneCb: server responded with state [SUCCEEDED]
[ INFO] [1452388037.091318054]: got result output = 21759; goal_stamp = 21759; diff = 0
```

With these nodes running, there will be 5 new topics under `/example_action`: cancel, feedback, goal, result, and status. You can watch the nodes communicate by running:

```
rostopic echo example_action/goal
```

which produces an output like:

```
---
header:
  seq: 21758
  stamp:
    secs: 1452388037
    nsecs: 89057580
  frame_id: ''
goal_id:
  stamp:
    secs: 1452388037
    nsecs: 89057840
  id: /demo_action_client_node-21759-1452388037.89057840
goal:
  input: 21759
```

```
---
header:
  seq: 21759
  stamp:
    secs: 1452388037
    nsecs: 91357355
  frame_id: ''
goal_id:
  stamp:
    secs: 1452388037
    nsecs: 91357624
  id: /demo_action_client_node-21760-1452388037.91357624
goal:
  input: 21760
---
```

or examine the "result"message with:

```
rostopic echo example_action/result
```

which produces an output like:

```
---
header:
  seq: 21759
  stamp:
    secs: 1452388037
    nsecs: 92533965
  frame_id: ''
status:
  goal_id:
    stamp:
      secs: 1452388037
      nsecs: 91357624
    id: /demo_action_client_node-21760-1452388037.91357624
  status: 3
  text: ''
result:
  output: 21760
  goal_stamp: 21760
---
```

You can try killing and restarting these nodes to observe the behaviors to timeouts or goal-count mismatches.

An idiosyncracy of action servers in ROS is that reconnection (or new connection) from client to server appears to be somewhat unreliable. It can be helpful to include automatic connection re-tries to achieve more reliable connection. For example, the following code snippet persists in re-trying to connect a client `cart_move_action_client_` to its associated action server.

```
    // attempt to connect to the server:
    ROS_INFO("waiting for server: ");
    bool server_exists = false;
    while ((!server_exists)&&(ros::ok())) {
        server_exists = cart_move_action_client_.waitForServer(ros::Duration(0.5)); //
        ros::spinOnce();
```

```
        ros::Duration(0.5).sleep();
        ROS_INFO("retrying...");
    }
    ROS_INFO("connected to action server"); // if here, then we connected to the server;
```

A second action-server/action-client illustrative example is given in `example_action_server_w_fdbk.cpp` and `timer_client.cpp`. The contents of the action-server code is in Listing 2.14.

Listing 2.14: `example_action_server_w_fdbk.cpp`

```
// example_action_server: 2nd version, includes "cancel" and "feedback"
// expects client to give an integer corresponding to a timer count, in seconds
// server counts up to this value, provides feedback, and can be cancelled any time
// re-use the existing action message, although not all fields are needed
// use request "input" field for timer setting input,
// value of "fdbk" will be set to the current time (count-down value)
// "output" field will contain the final value when the server completes the goal request

#include<ros/ros.h>
#include <actionlib/server/simple_action_server.h>
//the following #include refers to the "action" message defined for this package
// The action message can be found in: .../example_action_server/action/demo.action
// Automated header generation creates multiple headers for message I/O
// These are referred to by the root name (demo) and appended name (Action)
#include<example_action_server/demoAction.h>

int g_count = 0;
bool g_count_failure = false;

class ExampleActionServer {
private:

    ros::NodeHandle nh_;  // we'll need a node handle; get one upon instantiation

    // this class will own a "SimpleActionServer" called "as_".
    // it will communicate using messages defined in example_action_server/action/demo.action
    // the type "demoAction" is auto-generated from our name "demo" and generic name "Action"
    actionlib::SimpleActionServer<example_action_server::demoAction> as_;

    // here are some message types to communicate with our client(s)
    example_action_server::demoGoal goal_; // goal message, received from client
    example_action_server::demoResult result_; // put results here, to be sent back to the client when done w/ goal
    example_action_server::demoFeedback feedback_; // for feedback
    //  use: as_.publishFeedback(feedback_); to send incremental feedback to the client
    int countdown_val_;


public:
    ExampleActionServer(); //define the body of the constructor outside of class definition

     ExampleActionServer(void) {
    }
    // Action Interface
    void executeCB(const actionlib::SimpleActionServer<example_action_server::demoAction>::GoalConstPtr& goal);
};

//implementation of the constructor:
// member initialization list describes how to initialize member as_
// member as_ will get instantiated with specified node-handle, name by which this server will be known,
//  a pointer to the function to be executed upon receipt of a goal.
//
// Syntax of naming the function to be invoked: get a pointer to the function, called executeCB,
```

```
// which is a member method of our class exampleActionServer.
// Since this is a class method, we need to tell boost::bind that it is a class member,
// using the "this" keyword.  the _1 argument says that our executeCB function takes one argument
// The final argument,  "false", says don't start the server yet.  (We'll do this in the constructor)

ExampleActionServer::ExampleActionServer() :
   as_(nh_, "timer_action", boost::bind(&ExampleActionServer::executeCB, this, _1),false)
// in the above initialization, we name the server "example_action"
//  clients will need to refer to this name to connect with this server
{
    ROS_INFO("in constructor of exampleActionServer...");
    // do any other desired initializations here...specific to your implementation

    as_.start(); //start the server running
}

//executeCB implementation: this is a member method that will get registered with the action server
// argument type is very long.  Meaning:
// actionlib is the package for action servers
// SimpleActionServer is a templated class in this package (defined in the "actionlib" ROS package)
// <example_action_server::demoAction> customizes the simple action server to use our own "action" message
// defined in our package, "example_action_server", in the subdirectory "action", called "demo.action"
// The name "demo" is prepended to other message types created automatically during compilation.
// e.g.,  "demoAction" is auto-generated from (our) base name "demo" and generic name "Action"
void ExampleActionServer::executeCB(const actionlib::SimpleActionServer<example_action_server::demoAction>::GoalConstPtr& goal) {
    ROS_INFO("in executeCB");
    ROS_INFO("goal input is: %d", goal->input);
    //do work here: this is where your interesting code goes
    ros::Rate timer(1.0); // 1Hz timer
    countdown_val_ = goal->input;
    //implement a simple timer, which counts down from provided countdown_val to 0, in seconds
    while (countdown_val_>0) {
       ROS_INFO("countdown = %d",countdown_val_);

       // each iteration, check if cancellation has been ordered
       if (as_.isPreemptRequested()){
          ROS_WARN("goal cancelled!");
          result_.output = countdown_val_;
          as_.setAborted(result_); // tell the client we have given up on this goal; send the result message as well
          return; // done with callback
        }

       //if here, then goal is still valid; provide some feedback
       feedback_.fdbk = countdown_val_; // populate feedback message with current countdown value
       as_.publishFeedback(feedback_); // send feedback to the action client that requested this goal
       countdown_val_--; //decrement the timer countdown
       timer.sleep(); //wait 1 sec between loop iterations of this timer
    }
    //if we survive to here, then the goal was successfully accomplished; inform the client
    result_.output = countdown_val_; //value should be zero, if completed countdown
    as_.setSucceeded(result_); // return the "result" message to client, along with "success" status
}

int main(int argc, char** argv) {
    ros::init(argc, argv, "timer_action_server_node"); // name this node

    ROS_INFO("instantiating the timer_action_server: ");

```

```
111     ExampleActionServer as_object; // create an instance of the class "ExampleActionServer"
112
113     ROS_INFO("going into spin");
114     // from here, all the work is done in the action server, with the interesting stuff done within "executeCB()"
115     // you will see 5 new topics under example_action: cancel, feedback, goal, result, status
116     ros::spin();
117
118     return 0;
119 }
```

Listing 2.14 is mostly identical to `example_action_server.cpp`, except for the name by which ROS will know the action server, `timer_action` (line 58), and the body of the callback function (lines 77-104). In the callback function, the goal is interpreted as a timer value, and the job of the callback function is to count down by this many seconds then return.

The callback function has two important new elements. First, lines 88-93 test for whether the client has requested that the current goal be preempted. If so, the current goal is aborted, the client is informed of this status, and the callback function concludes. It is the designer's responsibility to test for this status within the action server at whatever frequency is appropriate.

Checking for goal cancellation can be quite important. For example, a trajectory command might be sent to a robot that is subsequently found to be a bad idea (e.g., a person walks into the robot's workspace). It may be necessary to stop the commanded motion suddenly, based on an alarm computed via a separate node. Alternatively, the commanded motion might be stalled, e.g. by a barrier blocking a specified navigation path. This could result in a deadlock, with the action-server never returning a result to the action client. The client should be able to evaluate progress of a commanded goal and cancel it, if necessary.

The second important feature of this action server appears in lines 96-97. Here, the feedback message is updated and published. Such feedback can be used by the action client to evaluate progress, e.g. to detect and resolve deadlocks. It is up to the designer to provide a corresponding client subscription to receive and evaluate such feedback.

One could design an equivalent node that subscribed to a goal topic and published to feedback and result topics. However, this could create confusion when there are multiple client nodes attempting to communicate with this action node. Feedback and result messages that are published would be received by all clients, not just the client for which the current goal is being processed. Using the action-server/action-client means of communication, this confusion is avoided. Only the client of the current goal receives feedback and result information. When a goal is completed, any other client can request a new goal.

This example uses the "simple" action-server library. The simple version does not attempt to handle multiple goals in parallel, nor does it attempt to queue up requests. Any goal requested while a goal is still in process will cause pre-emption of the goal in process. This is true whether the conflicting goals come from a single client or from multiple clients running in parallel. The designer must consider implicatins of this behavior when using simple action servers.

A corresponding action client that interacts with `example_action_server_w_fdbk.cpp` is `timer_client.cpp`, shown in Listing 2.15.

Listing 2.15: `timer_client.cpp`

```
// timer_client: works together with action server called "timer_action"
// in source: example_action_server_w_fdbk.cpp
// this code could be written using classes instead (e.g. like the corresponding server)
```

```
//  see: http://wiki.ros.org/actionlib_tutorials/Tutorials/Writing%20a%20Callback%20Based%20Simple%20Action%20Client

#include<ros/ros.h>
#include <actionlib/client/simple_action_client.h>
#include<example_action_server/demoAction.h> //reference action message in this package

using namespace std;

bool g_goal_active = false; //some global vars for communication with callbacks
int g_result_output = -1;
int g_fdbk = -1;

// This function will be called once when the goal completes
// this is optional, but it is a convenient way to get access to the "result" message sent by the server
void doneCb(const actionlib::SimpleClientGoalState& state,
        const example_action_server::demoResultConstPtr& result) {
    ROS_INFO(" doneCb: server responded with state [%s]", state.toString().c_str());
    ROS_INFO("got result output = %d",result->output);
    g_result_output= result->output;
    g_goal_active=false;
}

//this function wakes up every time the action server has feedback updates for this client
// only the client that sent the current goal will get feedback info from the action server
void feedbackCb(const example_action_server::demoFeedbackConstPtr& fdbk_msg) {
    ROS_INFO("feedback status = %d",fdbk_msg->fdbk);
    g_fdbk = fdbk_msg->fdbk; //make status available to "main()"
}

// Called once when the goal becomes active; not necessary, but could be useful diagnostic
void activeCb()
{
  ROS_INFO("Goal just went active");
  g_goal_active=true; //let main() know that the server responded that this goal is in process
}

int main(int argc, char** argv) {
        ros::init(argc, argv, "timer_client_node"); // name this node
        ros::NodeHandle n;
        ros::Rate main_timer(1.0);
        // here is a "goal" object compatible with the server, as defined in example_action_server/action
        example_action_server::demoGoal goal;

        // use the name of our server, which is: timer_action (named in example_action_server_w_fdbk.cpp)
        // the "true" argument says that we want our new client to run as a separate thread (a good idea)
        actionlib::SimpleActionClient<example_action_server::demoAction> action_client("timer_action", true);

        // attempt to connect to the server: need to put a test here, since client might launch before server
        ROS_INFO("attempting to connect to server: ");
        bool server_exists = action_client.waitForServer(ros::Duration(1.0)); // wait for up to 1 second
        // something odd in above: sometimes does not wait for specified seconds,
        //  but returns rapidly if server not running; so we'll do our own version
        while (!server_exists) { // keep trying until connected
            ROS_WARN("could not connect to server; retrying...");
            server_exists = action_client.waitForServer(ros::Duration(1.0)); // retry every 1 second
        }
        ROS_INFO("connected to action server");  // if here, then we connected to the server;

        int countdown_goal = 1; //user will specify a timer value
        while(countdown_goal>=0) {
           cout<<"enter a desired timer value, in seconds (0 to abort, <0 to quit): ";
```

```
65             cin>>countdown_goal;
66             if (countdown_goal==0) { //see if user wants to cancel current goal
67               ROS_INFO("cancelling goal");
68               action_client.cancelGoal(); //this is how one can cancel a goal in process
69             }
70             if (countdown_goal<0) { //option for user to shut down this client
71                ROS_INFO("this client is quitting");
72                return 0;
73             }
74             //if here, then we want to send a new timer goal to the action server
75             ROS_INFO("sending timer goal= %d seconds to timer action server",countdown_goal);
76             goal.input = countdown_goal; //populate a goal message
77             //here are some options:
78             //action_client.sendGoal(goal); // simple example--send goal, but do not specify callbacks
79             //action_client.sendGoal(goal,&doneCb); // send goal and specify a callback function
80             //or, send goal and specify callbacks for "done", "active" and "feedback"
81             action_client.sendGoal(goal, &doneCb, &activeCb, &feedbackCb);
82 
83             //this example will loop back to the the prompt for user input.  The main function will be
84             // suspended while waiting on user input, but the callbacks will still be alive
85             //if user enters a new goal value before the prior request is completed, the prior goal will
86             // be aborted and the new goal will be installed
87 
88         }
89     return 0;
90 }
```

This client illustrates several additional features. Instead of one callback function, there are three. The callback function `activeCb()` (lines 34-38) is invoked when the action server accepts the client's goal request. This callback function is not required, but it can be useful for diagnostics. In the example, the callback function sets a flag to inform the main() function that the new goal is acknowledged and in process.

The callback function `feedbackCb()` (lines 23-31) receives feedback messages from the action server. It copies the results to global variables to make them accessible to main(). From such updates, the main function can confirm that the action server is still alive and progressing. A useful example of action-server feedback would be to report back on accomplishment of subgoals, e.g. from a goal request to navigate to a sequence of intermediate points in space or a goal request to perform a sequence of manipulations.

In instantiating the action client, the corresponding action server, `timer_action`, is named (line 49).

Lines 53-60 implement attempting to connect to the named action server, with re-tries once per second until the connection is successful. This is a useful construction, since launching code may result in action clients starting up before their respective action servers. Retrying connections will allow this node to wait patiently until a successful connection is achieved. An alternative is for the client to wait forever with the function:

```
action_client.waitForServer();
```

A limitation of this, however, is that there will be no output from the client, which might be suspended indefinitely without warning to the operator.

To associate the feedback functions with the goal request, the feedback functions are named in the sendGoal command (line 81):

```
action_client.sendGoal(goal, &doneCb, &activeCb, &feedbackCb);
```

With this construction, the action server will communicate its results back to this action-client node.

Another important feature of this action client is the absence of the call `action_client.waitForResult()`. Our client can proceed with other business while its requested goal is being carried out by the action server. For example, the client might be a coordinator responsible for checking evolving status, such as battery voltage getting low, detection of an anomaly that is important to investigate, failure of a navigation goal to progress, or detection of an error in the status of an on-going assembly. If the coordinator is not "blocked" waiting on the action server, it can evaluate such conditions and, as desired, abort the current goal and re-plan to generate an alternative goal.

In the simple timer client example, the client prompts an operator for a keyboard input corresponding to a timer duration (an integer number of seconds). If the goal is 0, this interpreted (in this example) to mean that the operator wishes to cancel the current goal. This is done with the call (line 68):

```
action_client.cancelGoal();
```

If the requested goal is positive, then the client sends this value as a new goal. The default behavior of the simple action server is to accept the new goal, pre-empting the previous goal if it is not concluded.

As with service servers and service clients, the action server can remain alive indefinitely, serving various clients that may come and go.

Figure 2.1 is a screenshot displaying the output from the client and server interaction for the timer example, along with a display of `rqt_console`. In this example, the client was started first. The console display, message number 1, shows that the node `timer_client_node` output that it was attempting to connect to the server. Messages 2 through 6 report unsuccessful retries to connect. Message 7 is reported by the timer action server, which was started after the client. At this point, the server is waiting for goals and the client is waiting for user input.

After about about 8 seconds (as shown by the time stamps on messages 8 and 9), the user enters 100, and the client echoes that the value 100 is being sent as a goal to the action server (console message 9). The action server reports that its callback function is activated (message 10). With acceptance of the goal, the client node's corresponding callback function is activated (message 13). Both the server and client display the countdown status (messages 12-18). This shows that the client is successfully receiving feedback messages from the server (although the client's main function is ignoring this information, in this example).

Note that, during this countdown, the client's main program is suspended, waiting on new input from the user. However, the client's callback functions continue to respond to messages from the action server–in spite of the fact that there are no `ros::spin()` nor `ros::spinOnce()` calls. This is because, when the action client was instantiated (Listing 2.15 line 49):

```
actionlib::SimpleActionClient<example_action_server::demoAction> action_client("↩
    timer_action", true);
```

the argument "true" specified that the action client should be run as a separate thread. As a result, the callback functions continue to respond to messages even when the main program is blocked, and no "spin" calls are required.

While the client is suspended, the action server continues to count down, and the client's callback functions continue to receive feedback. After a few seconds of countdown, the user enters "0" to the client timer prompt, and the console messages 19 and 20 show that the client interprets this as a request to cancel the current goal. Message 22 shows that the action server acknowledges the cancellation request. The client and server are both ready to accept new goals.

A second client may be started up while the first client is still connected–with or without a goal-completion pending. However, if the second client attempts to send a goal while the first client's goal is still in process, the server cancels the first goal, and the first client's "doneCB()" callback function receives notification and prints out:

```
doneCb: server responded with state [ABORTED]
```

The designer should be aware of this behavior. It can be useful to cancel an ongoing goal initiated by one client then cancelled by another client. This can help dislodge a deadlock, if the first client misbehaves.

An action client that sends a goal can conclude (or be killed) without stopping the action server from continuing to process the goal. Even though the action server will attempt to report back to the (deceased) action client, this communication is merely a publication, and failure to receive this publication does not impeded the action server from continuing along normally.

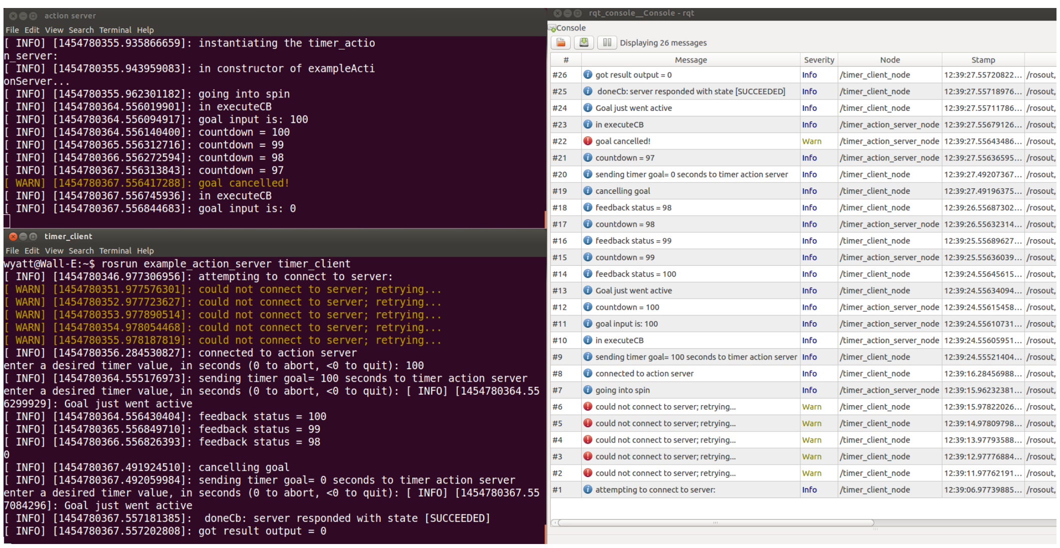


FIGURE 2.1: Output of rqt console and action client and action server terminals with timer example

2.6 WRAP-UP

This concludes the introduction to action servers. A fourth means of ROS communications–the "parameter server" is introduced next.

2.7 INTRODUCTION TO THE PARAMETER SERVER

Sending a message, a service request or an action-server goal are all transactions that are typically very fast. At the other end of the spectrum, the ROS "parameter server" is useful for sharing values that are infrequently changed. From `http://wiki.ros.org/Parameter%20Server`:

> A parameter server is a shared, multi-variate dictionary that is accessible via network APIs. Nodes use this server to store and retrieve parameters at runtime. As it is not designed for high-performance, it is best used for static, non-binary data such as configuration parameters.

While a publisher "broadcasts" its messages, and service and action-server communications are peer-to-peer, the parameter server is more like a shared memory. Via code within nodes, via terminal commands or via launch files, any process can can read, write or alter parameter values on the parameter server. This communication mechanism can be very convenient, but it must also be used appropriately and with care.

The parameter server is commonly used to set configuration parameters or specifications, including:

- Controller gains for joint servos
- Coordinate transform parameters, e.g. for tool transforms
- Sensor intrinsic and extrinsic calibration parameters
- Robot models

It is frequently convenient to store parameter settings in YAML files, which can be loaded into the parameter server via a command line, or (more commonly) as procedures within a "launch" file. With this approach, one can make parameter changes and retest a system without touching or recompiling the source code. If a sensor mount is changed, the modified coordinate transform data can be incorporated in the system by editing the corresponding configuration file and re-launching the system. If a robot's end-effector is changed, the corresponding new tool transform can be incorporated by changing its corresponding configuration file. Parameter changes to a robot's model can also be introduced without changing any source code.

On the other hand, the parameter server should not be used for dynamically-changing values. Unlike a subscriber, a service or an action service, incoming data is not constantly detected. Rather, nodes typically consult the parameter server once, upon start-up, and they are subsequently unaware of changes to the parameter server.

2.7.1 Rosparam

`rosparam` is a command-line tool for getting and setting ROS Parameters on the Parameter Server using YAML-encoded information (see `http://wiki.ros.org/rosparam`). Options for `rosparam` are shown by running:

```
rosparam
```

which results in displaying:

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

Commands:
```

```
rosparam set set parameter
rosparam get get parameter
rosparam load load parameters from file
rosparam dump dump parameters to file
rosparam delete delete parameter
rosparam list list parameter names
```

A common example would be to set proportional, derivative and integral-error gains for a servo control algorithm. Values of interest could be put on the parameter server manually, e.g. setting the p,i,d values to 1,2,3, respectively, with the command:

```
 rosparam set /gains "p: 1.0
i: 2.0
d: 3.0"
```

The result of this action can be seen by entering:

```
rosparam list
```

which produces the output:

```
/gains/d
/gains/i
/gains/p
/rosdistro
/roslaunch/uris/host_wall_e__54538
/rosversion
/run_id
```

The command:

```
rosparam get /gains/d
```

displays the expectd result of 3.0. Alternatively, all of the gains can be seen by commanding:

```
rosparam get /gains
```

which displays:

```
{d: 3.0, i: 2.0, p: 1.0}
```

The entire set of parameter values can be obtained by dumping it to a file with, e.g.:

```
 rosparam dump param_dump
```

where `param_dump` is a chosen file name in which to dump the data. The contents of `param_dump` at this point is:

```
gains: {d: 3.0, i: 2.0, p: 1.0}
rosdistro: 'indigo

  '
roslaunch:
  uris: {host_wall_e__54538: 'http://Wall-E:54538/'}
rosversion: '1.11.13

  '
run_id: f8504ae0-b747-11e5-af4f-c48508582a82
```

Alternatively, the parameter values could be loaded from a file. As an example, the file `test_param.yaml` with contents:

```
joint1_gains: {p: 4.0, i: 5.0, d: 6.0}
```

can be loaded onto the parameter server with the command:

```
rosparam load jnt1_gains.yaml
```

After loading this data, it can be confirmed the desired values are present on the parameter server with the command:

```
rosparam get /joint1_gains
```

which results in displaying:

```
{d: 6.0, i: 5.0, p: 4.0}
```

To automate loading configuration files, ROS "launch" files accept a parameter tag. As an example, a launch file, called `load_gains.launch` can be composed with the contents:

```
roslaunch load_gains.launch
```

can be run with the command:

```
<launch>
<rosparam command="load" file="jnt1_gains.yaml" />
</launch>
```

The effect of this command is identical to running: `rosparam load jnt1_gains.yaml`.

While it is convenient to be able to load parameters onto the parameter server automatically through launch files, the value of the parameter server is realized when nodes are able to access the data. Means to access parameter data using C++ code is described in `http://wiki.ros.org/roscpp/Overview/Parameter%20Server`. An example package illustrating accessing parameter data is in the accompanying repository under `example_parameter_server`. This package contains `src` and `launch` subdirectories. The launch directory contains the launch file `load_gains.launch`, introduced above. The YAML file `jnt1_gains.yaml` also resides in this directory. The desired data can be loaded into the parameter server with the command:

```
roslaunch example_parameter_server load_gains.launch
```

Within the package `example_parameter_server`, the `src` folder contains a file named `read_param_from_node.cpp` with contents:

Listing 2.16: C++ code to illustrate getting parameters from the parameter server

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

int main(int argc, char **argv) {
    ros::init(argc, argv, "param_reader"); // name of this node will be "↩
        minimal_publisher"
    ros::NodeHandle nh; // two lines to create a publisher object that can talk to ROS
    double P_gain,D_gain,I_gain;

```

```
    if (nh.getParam("/joint1_gains/p", P_gain)) {
    ROS_INFO("proportional gain set to %f",P_gain);
    }
    else
    {
    ROS_WARN("could not find parameter value /joint1_gains/p on parameter server");
    }
    if (nh.getParam("/joint1_gains/d", D_gain)) {
    ROS_INFO("proportional gain set to %f",D_gain);
    }
    else
    {
    ROS_WARN("could not find parameter value /joint1_gains/d on parameter server");
    }
    if (nh.getParam("/joint1_gains/i", I_gain)) {
    ROS_INFO("proportional gain set to %f",I_gain);
    }
    else
    {
    ROS_WARN("could not find parameter value /joint1_gains/i on parameter server");
    }
}
```

The code is compiled with the CMakeLists.txt line `cs_add_executable(read_param_from_node src/read_param_` To test the resulting executable, first make sure the parameters are loaded by running:

```
roslaunch example_parameter_server load_gains.launch
```

Then, run the test node with:

```
rosrun example_parameter_server read_param_from_node
```

The resulting output displayed is:

```
[ INFO] [1452404560.596815928]: proportional gain set to 7.000000
[ INFO] [1452404560.598031455]: proportional gain set to 9.000000
[ INFO] [1452404560.599246423]: proportional gain set to 8.000000
```

To see what happens when the data is not present on the parameter server, delete the gain values with the command:

```
rosparam delete /joint1_gains
```

Re-running `rosrun example_parameter_server read_param_from_node` results in the output:

```
[ WARN] [1452404731.031529362]: could not find parameter value /joint1_gains/p on parameter server
[ WARN] [1452404731.032923909]: could not find parameter value /joint1_gains/d on parameter server
[ WARN] [1452404731.034251408]: could not find parameter value /joint1_gains/i on parameter server
```

The function “`ROS_WARN()`” displays output in a yellow color and carries the tag “WARN.” Such messages are also highlighted in `rqt_console`. The line:

```
if (nh.getParam("/joint1_gains/p", P_gain))
```

tests for the existence of the parameter `/joint1_gains/p` on the parameter server. This capability is useful to make sure start-up is consistent, with all required parameters specified before they are used in operation.

Parameters can also be set programmatically, although such use is less common.

Although the example provided here is (deliberately) trivial, the parameter server is capable of handling much larger and more complex data. A “parameter” used in almost all ROS applications is robot_description, which contains a full description of a robot model. In section ??, robot "URDF" models will be introduced. Within a corresponding launch file, the lines:

```
  <!-- send robot urdf to param server -->
<param name="robot_description"
textfile="$(find minimal_robot_description)/minimal_robot_w_sensor.urdf"/>
```

induces roslaunch to search for the package minimal_robot_description, find the file minimal_robot_w_sensor.urdf within this package, and load the contents onto the parameter server with the associated parameter name robot_description. The robot model is then available both for control algorithms and for visualization displays.

2.8 WRAP-UP

This concludes our introduction to ROS foundations. The material presented in Part I describes the philosophy of ROS, including packages, nodes, messages, and services. With this framework, large systems can be built collaboratively more easily. Contributions can be broken up by packages containing messages, libraries and/or nodes. By defining the interface options among nodes (publish/subscribe, service/client, action-server/action-client or parameter server), collaborators can make their work compatible with large systems, simplifying integration. Even within a single-person project, breaking up the design task in terms of nodes promotes faster design with simpler integration and supports growth, extensions, upgrades and software re-use. These attributes make ROS attractive for robot software design.

Standardization on the ROS infrastructure is compelling enough to make it a preferred approach to robot software design. In addition, though, ROS has extensive modeling, simulation and visualization capabilities that further enhance attraction to ROS. Modeling, simulation and visualization in ROS will be introduced in Part II.

II

Simulation and Visualization in ROS

CHAPTER 3

Simulation in ROS

CONTENTS

INTRODUCTION
A strong asset of ROS is its simulation and visualization capabilities. When software development can only be performed on specific hardware (e.g. a particular robot), productivity plummets. A physical robot is typically a bottlneck for development, because it is a limited resource presumably shared by a team. Further, some robots, including vehicles and humanoids, require a team to run experiments. Most often, the robot and team are idling while a programming is seeking errors in their code under test.

With a suitable robot simulator, code can be developed under simulation, allowing work to proceed in parallel and without imposing on a support staff for experiments. If the simulator is high fidelity (providing a good approximation of the actual robot's behavior), then few or no changes may be required to run the same code on the physical system.

One of the common barriers of simulation is that code written for simulation may need to be modified significantly to run on the physical system. This is an issue that has been addressed in ROS. Typically, code written for simulation in ROS requires no changes (not even recompilation) to run on the target physical system. Using ROS's messaging system, nodes are indifferent to where other nodes reside in the system, as long as the message types, topic names, aser server names are consistent. A ROS-enabled robot should publish its sensory data to named topics with defined message types–and the corresponding simulator should publish the same (simulated) information on the same topics with the same message types. If this is achieved, then switching a perceptual-processing system from simulation to a physical system only requires that the physical system (rather than the simualator) start publishing the information. In the other direction, one controls a robot by publications to topics (or ROS service requests or ROS action-server goal requests). This interface is defined in terms of topic names, message types and server names. If the simulator responds to these publications or requests in the same manner as the physical system, then controller code that runs on the simulation would run identically using the actual robot. Similarly, Human/Machine Interfaces may be developed with respect to a robot (and environment) simulator, and the developed code may be used verbatim with the physical system.

The success of developing software in simulation depends on the quality of the simulator. Fortunately, ROS has powerful simulation capabilities, both for robot dynamics and for sensor emulation in virtual worlds. These capabilities will be introduced here.

3.1 THE SIMPLE TWO-DIMENSIONAL ROBOT SIMULATOR

A helpful place to start the introduction to simulation in ROS is with the "Simple Two-Dimensional Robot simulator" (STDR). This package is described at `http://wiki.ros.org/stdr_simulator`, with accompanying tutorials at `http://wiki.ros.org/stdr_simulator/Tutorials`. If the STDR package is not already installed, it may be installed with the command:

```
sudo apt-get install ros-$ROS_DISTRO-stdr-simulator
```

With STDR installed, in may be launched with the command:

```
roslaunch stdr_launchers server_with_map_and_gui_plus_robot.launch
```

This results in a screen display as shown in Fig 3.1 A small circle in the lower-left corner of

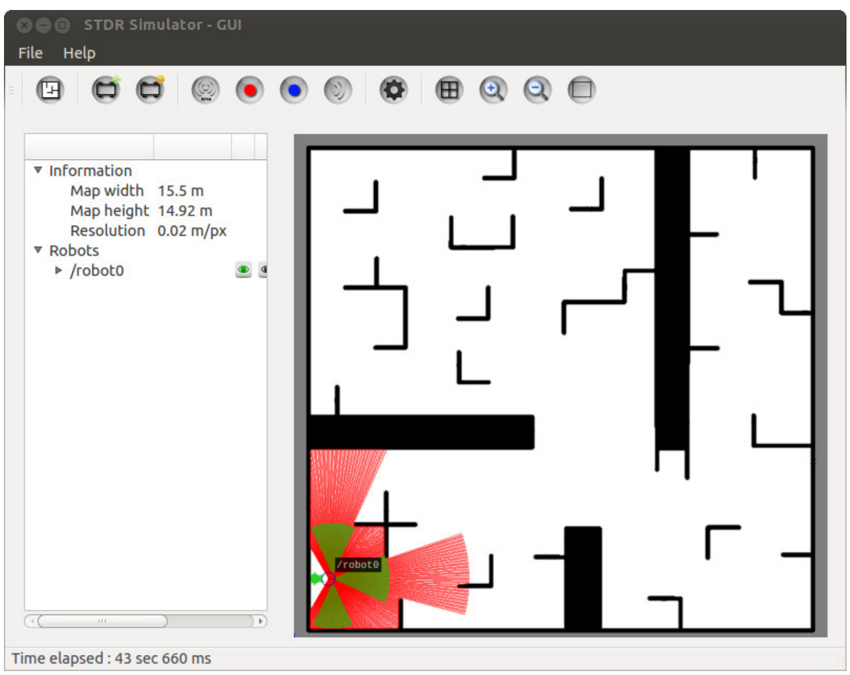


FIGURE 3.1: Display from STDR launch

the maze represents an abstracted mobile robot. The red rays illustrate simulated laser lines from a hypothetical LIDAR sensor (see, e.g., `https://en.wikipedia.org/wiki/Lidar` for a description of LIDAR sensors). The LIDAR rays extend from the robot out to the first point of reflection in the environment–subject to a maximum sensing range. The green cones represent simulated sonar signals. The distance reported by a sonar sensor corresponds to the range to the first sound-reflecting object in the environment within the sensing angle of the sonar (again, subject to a sensing range limitation). Sensor values from the robot simulator are published to topics. The robot subscribes to a command topic, via which one can control motion of the simulated robot. In fact, running:

```
rostopic list
```

shows that there are over 30 different topics active from the STDR launch. Additionally,

```
rosservice list
```

shows 24 services running.

With this simple simulator, one can develop code that can interpret the sensor signals, both to avoid collisions and to help identify the robot's location in space. Based on sensor interpretation, one can command incremental robot motions to achieve a desired behavior, such as navigating to specified coordinates in the map. We will defer sensory processing for now and focus on controlling the robot's motion (open loop, without benefit of sensor information). The topic of interest for controlling the robot is: `/robot0/cmd_vel`. The name `cmd_vel` is (by convention, though not requirement) the topic name used for controlling robots in velocity-command mode. This topic lives within the namespace of `/robot0`. The `cmd_vel` topic has been made a subset of `/robot0` to allow for launching multiple robots within the same simulation. If (as is typical) a ROS system is dedicated to controller a single robot, the `/robot0` namespace might be omitted.

We can examine the topic `/robot0/cmd_vel` with:

```
rostopic info /robot0/cmd_vel
```

which displays:

```
Type: geometry_msgs/Twist

Publishers: None

Subscribers:
 * /robot_manager (http://Wall-E:58336/)
 * /stdr_gui_node_Wall_E_15095_8750187360825501198 (http://Wall-E:60301/)
```

This reveals that the `/robot_manager` node is listening to this topic, but at present there are no publishers to this topic. Further, we see that messages on this topic are of type `geometry_msgs/Twist`. We can examine the twist message with:

```
rosmsg show geometry_msgs/Twist
```

which displays:

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

```
float64 x
float64 y
float64 z
```

The Twist message contains the equivalent of two vectors: a speed vector (with x, y and z components) and an angular-velocity (rotation rate) vector, also with x, y and z components. This message type is sufficient for specifying arbitrary velocities in space, both translating and spinning/tumbling. It is often useful to have full 6-D command capability, e.g. for aerial vehicles, submersible vehicles or the end-effector of an arm.

For our 2-D mobile robot, only 2 of these 6 components are viable. Constraining the motion of the robot to a plan, the robot can only move forward (relative to its own heading) and/or spin about its center. These two components, speed and spin, correspond to the x-component of linear velocity and the z-component of angular velocity.

We can command a velocity to the robot manually from a terminal by entering:

```
rostopic pub -r 2 /robot0/cmd_vel geometry_msgs/Twist  '{linear:  {x: 0.5, y: 0.0, z: ↩
    0.0}, angular: {x: 0.0,y: 0.0,z: 0.0}}'
```

The above command instructs the robot to move forward at a velocity of 0.5 m/s with constant heading (zero angular velocity). (Note: by default, all units in ROS messages are MKS). As a result of this command, the robot moves forward until it hits an obstacle, as shown in Fig 3.2. Note that the LIDAR rays and sonar cones have changed to be consistent

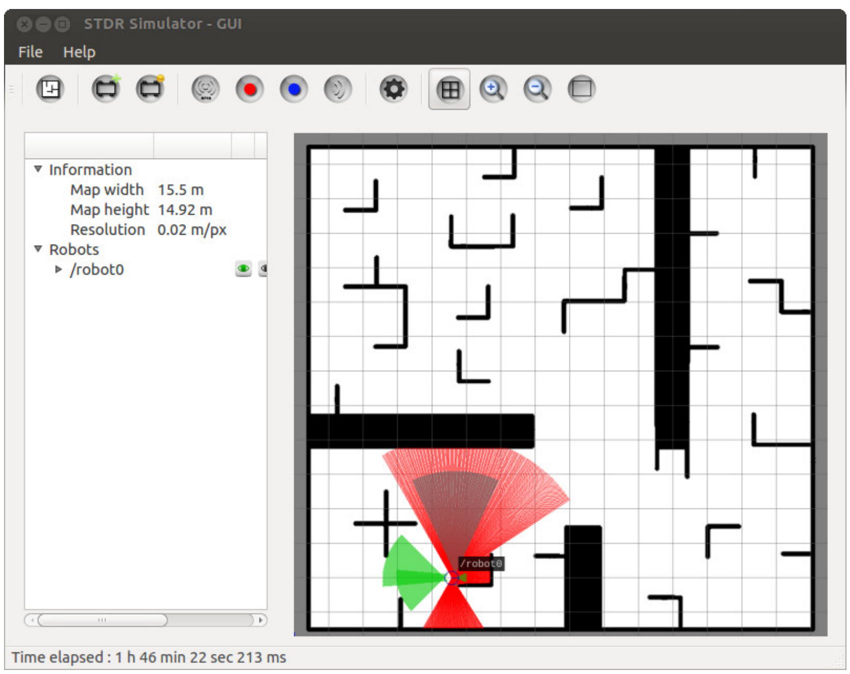


FIGURE 3.2: STDR stalls at collision after executing forward-motion command from start position

with the new relative positions of reflective surfaces in the robot's virtual world.

The manually-entered twist command is reiterated at 2Hz, via the option `-r 2`. However, if the rostopic pub command is halted (with control-C), the robot will continue to try to move with the last twist command received. We can return the twist command to zero with:

```
rostopic pub -r 2 /robot0/cmd_vel geometry_msgs/Twist  '{linear:  {x: 0.0, y: 0.0, z: ↩
     0.0}, angular: {x: 0.0,y: 0.0,z: 0.0}}'
```

To get the robot unstuck and allow it to move forward again, a pure rotation command may be issued, e.g. as:

```
rostopic pub -r 2 /robot0/cmd_vel geometry_msgs/Twist  '{linear:  {x: 0.0, y: 0.0, z: ↩
     0.0}, angular: {x: 0.0,y: 0.0,z: 0.1}}'
```

The only non-zero component of the above command is the z-component of the angular velocity. The robot's z axis points "up" (out of the plan view) and a positive angular-velocity command corresponds to counter-clockwise rotation. The commanded angular velocity is 0.1 rad/s. Allowing this command to run for approximately 15 seconds corresponds to a rotation of 1.5 radians, or approximately 90 degrees. The resulting pose of the robot is shown in Fig 3.3 The robot has not translated after this command, but it has rotated (as

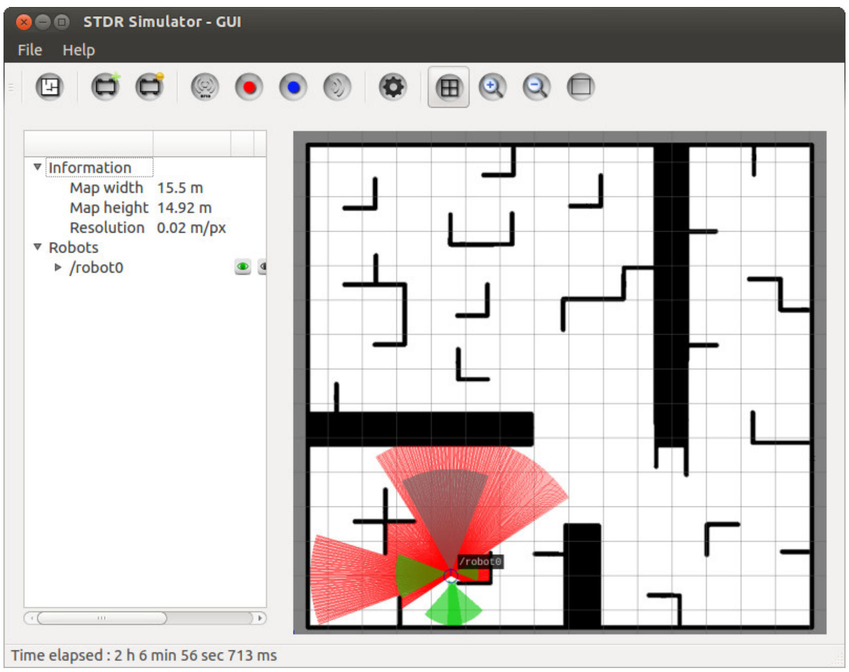


FIGURE 3.3: STDR after an approximate 90-deg counter-clockwise rotation

is apparent from the new sensor visualization) to point approximately towards the upper boundary of the map. If the command:

```
rostopic pub -r 2 /robot0/cmd_vel geometry_msgs/Twist  '{linear:  {x: 0.5, y: 0.0, z: ↩
     0.0}, angular: {x: 0.0,y: 0.0,z: 0.0}}'
```

is issued, the robot will again move forward until it encounters a new barrier, as shown in Fig 3.4.

While the manual command-line Twist publications illustrate how to control the robot, such commands should be issued programmatically (i.e. from a ROS node). An example node to control the STDR mobile robot appears in the accompanying repository within package `stdr_control` (under Part II of the repository). This package was created using:

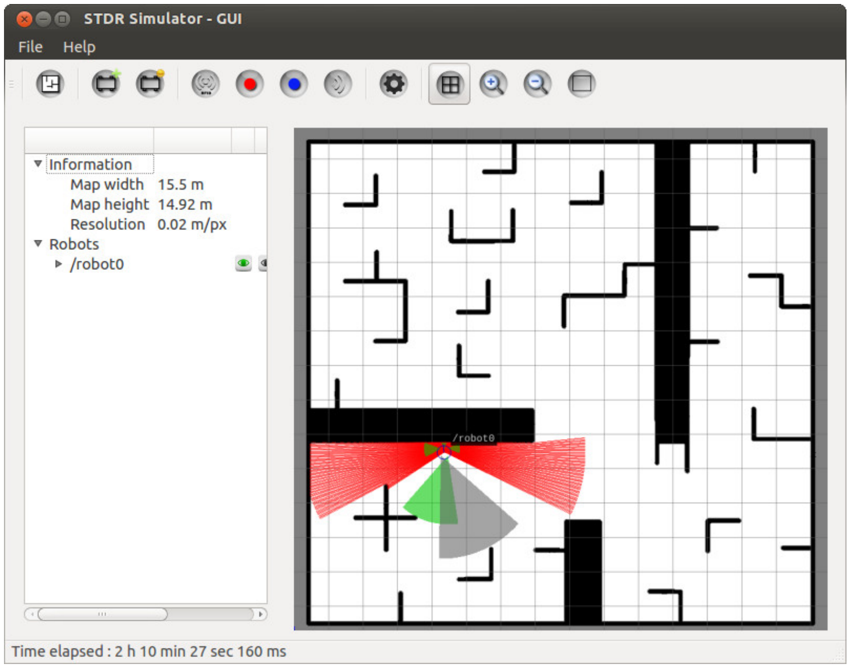


FIGURE 3.4: STDR stalls again at collision after executing another forward-motion command

```
cs_create_pkg stdr_control roscpp geometry_msgs/Twist
```

which specifies a dependency on `geometry_msgs/Twist`, which is needed to publish messages on topic `robot0/cmd_vel`.

A command publisher node, `stdr_open_loop_commander.cpp`, was written, starting from a copy of `minimal_publisher`. The contents of this program is listed in Listing 3.1.

Listing 3.1: C++ code to command velocities to the STDR simulator

```cpp
#include <ros/ros.h>
#include <geometry_msgs/Twist.h>
//node to send Twist commands to the Simple 2-Dimensional Robot Simulator via cmd_vel
int main(int argc, char **argv) {
    ros::init(argc, argv, "stdr_commander");
    ros::NodeHandle n; // two lines to create a publisher object that can talk to ROS
    ros::Publisher twist_commander = n.advertise<geometry_msgs::Twist>("/robot0/cmd_vel", 1);
    //some "magic numbers"
    double sample_dt = 0.01; //specify a sample period of 10ms
    double speed = 1.0; // 1m/s speed command
    double yaw_rate = 0.5; //0.5 rad/sec yaw rate command
    double time_3_sec = 3.0; // should move 3 meters or 1.5 rad in 3 seconds


    geometry_msgs::Twist twist_cmd; //this is the message type required to send twist commands to STDR
    // start with all zeros in the command message; should be the case by default, but just to be safe..
    twist_cmd.linear.x=0.0;
    twist_cmd.linear.y=0.0;
    twist_cmd.linear.z=0.0;
```

```
20     twist_cmd.angular.x=0.0;
21     twist_cmd.angular.y=0.0;
22     twist_cmd.angular.z=0.0;
23 
24     ros::Rate loop_timer(1/sample_dt); //create a ros object from the ros Rate class; ↩
           set 100Hz rate
25     double timer=0.0;
26     //start sending some zero-velocity commands, just to warm up communications with ↩
           STDR
27     for (int i=0;i<10;i++) {
28       twist_commander.publish(twist_cmd);
29       loop_timer.sleep();
30     }
31     twist_cmd.linear.x=speed; //command to move forward
32     while(timer<time_3_sec) {
33           twist_commander.publish(twist_cmd);
34           timer+=sample_dt;
35           loop_timer.sleep();
36           }
37     twist_cmd.linear.x=0.0; //stop moving forward
38     twist_cmd.angular.z=yaw_rate; //and start spinning in place
39     timer=0.0; //reset the timer
40     while(timer<time_3_sec) {
41           twist_commander.publish(twist_cmd);
42           timer+=sample_dt;
43           loop_timer.sleep();
44           }
45 
46     twist_cmd.angular.z=0.0; //and stop spinning in place
47     twist_cmd.linear.x=speed; //and move forward again
48     timer=0.0; //reset the timer
49     while(timer<time_3_sec) {
50           twist_commander.publish(twist_cmd);
51           timer+=sample_dt;
52           loop_timer.sleep();
53           }
54     //halt the motion
55     twist_cmd.angular.z=0.0;
56     twist_cmd.linear.x=0.0;
57     for (int i=0;i<10;i++) {
58       twist_commander.publish(twist_cmd);
59       loop_timer.sleep();
60     }
61     //done commanding the robot; node runs to completion
62 }
```

In Listing 3.1, line 2 includes the required header file to use the `geometry_msgs/Twist` message.

```
#include <geometry_msgs/Twist.h>
```

A publisher is instantiated on line 7, specifying the command topic and compatible message type:

```
ros::Publisher twist_commander = n.advertise<geometry_msgs::Twist>("/robot0/cmd_vel", ↩
     1);
```

A sample period is defined on line 9:

```
 double sample_dt = 0.01; //specify a sample period of 10ms
```

and this sample period is used to keep a timer consistent with "sleep()" calls of a ROS Rate object. A twist object is defined on line 15:

```
geometry_msgs::Twist twist_cmd; //this is the message type required to send twist ↩
    commands to STDR
```

and lines 17-22 show how at access all six components of this message object.

The robot is commanded to move forward 3m, spin counter-clockwise 90 deg, then move forward again 3m. This is done open-loop by commanding speeds for computed durations in lines 32-36, 39-44 and 48-53, thus:

```
timer=0.0; //reset the timer
while(timer<time_3_sec) {
      twist_commander.publish(twist_cmd);
      timer+=sample_dt;
      loop_timer.sleep();
      }
```

As a result, the robot moves as desired, ending at the final pose shown in Fig 3.5

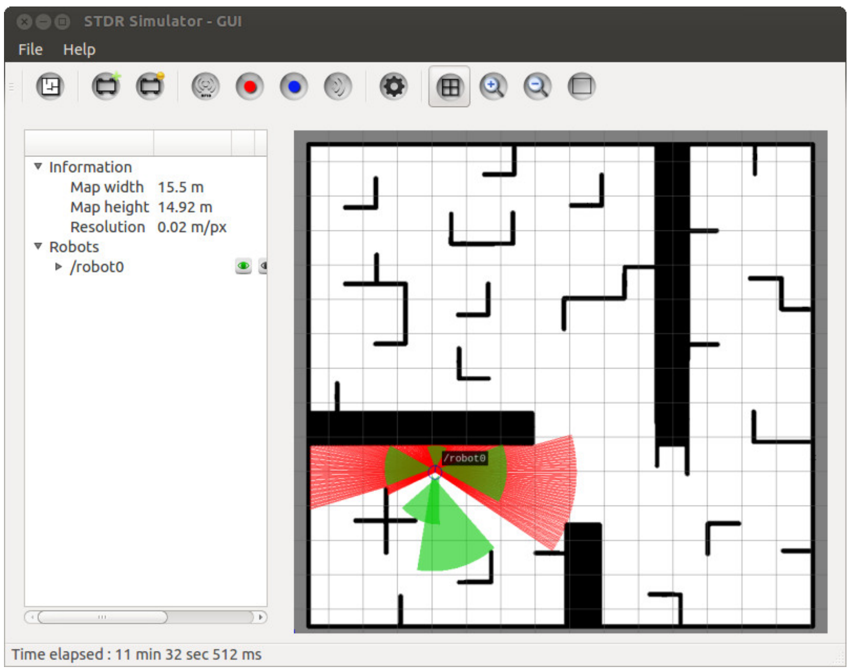


FIGURE 3.5: STDR final pose after executing programmed speed control

The Simple Two-Dimensional Robot Simulator may be used to develop more intelligent control code by interpreting sensor values and commanding computed twists that achieve effective reactive behaviors. Approaches to such control will be deferred until sensory-processing techniques have been introduced.

For designing sensor-based behaviors, this level of simulation abstraction may be appropriate, since it easy to operate, requires little computational power, and it allows the designer to focus on a limited context. This level of abstraction, however, does not include a variety of realistic effects, and thus code developed using this simulator may require additional work with a more general simulator before the code is suitable for testing on a real robot. In addition to the 2-D abstraction, the STDR simulator does not account for sensor noise, force/torque disturbances (including wheel slip), or realistic vehicle dynamics.

The STDR dynamic response can be viewed using `rqt_plot`. The topic `/robot0/odom` is useful for analysis, since messages on this topic (of type `nav_msgs/Odometry`) include the simulated robot's velocity and angular velocity. (By convention in ROS, a mobile robot should publish its "state" to an `odom` topic with message type `nav_msgs/Odometry`, where the state is based on sensor measurements. Details of the `/odom` topic will be covered later, in the context of mobile robots). Figure 3.6 shows the time history of the commanded and "actual" robot speed and yaw rate. Both the speed and spin are com-

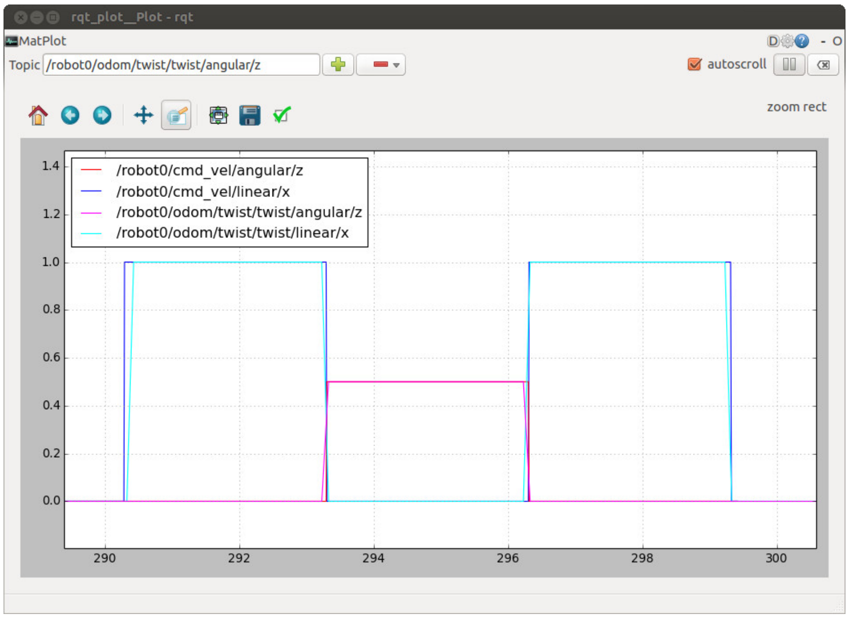


FIGURE 3.6: STDR commanded and actual speed and yaw rate vs time

manded with step changes, persisting for 3 seconds within each of the control blocks of the `stdr_open_loop_commander.cpp` code. The corresponding "actual" robot speed and yaw rate follow the commands very closely, with almost imperceptible errors due to jitter between when the command is issued and when the robot responds. In reality, the influences of inertia, angular inertia, actuator saturation, wheel friction and servo-controller response would result in quite different behavior. Expected non-idealities include ramp-up slew-rate limitations, overshoot, and wheel slip. The robot might even fall over due to commanding instantaneous braking. A more realistic controller would ramp up and ramp down the velocity commands to respect dynamic limitations. Appropriate ramping can be implemented to control the STDR simulation, but the necessity of such ramping will not be revealed by STDR, nor can we expect that ramping implemented on STDR will be appropriately tuned for a real robot.

More sophisticated robot simulation requires a sufficiently detailed robot model, including geometry, mass properties, surface contact properties and actuator dynamics. These can be specified using the "Unified Robot Description Format" introduced next.

3.2 MODELING FOR DYNAMIC SIMULATION

To perform physically-realistic dynamic simulations, objects must be specified with sufficient detail. Modeling details are organized within three broad categories: a dynamic model, a collision model and a visual model.

Dynamic objects must always include specification of inertial properties. This constitutes the necessary description for an abstract physics model. With an inertial model alone, one can already perform useful simulations. As an example, one could compute the dynamics of a satellite acted on by gravity and thrusters, producing a simulation that incorporates the influences of gravity, centrifugal and Coriolis effects acting on an arbitrary inertia tensor.

To compute the dynamics of interacting bodies, including colliding objects or a robot grabbing or pushing an object, it is also necessary to compute the forces and moments due to contacts. Simulation of contact dynamics is challenging, and different physics engines address this problem with different methods. The default physics engine used in ROS, "Open Dynamics Engine" (ODE) (see `http://ode-wiki.org/wiki`) uses energy and conservation of momentum to deduce the outcome of collisions, rather than attempt to detail the very fast dynamics of force profiles during impacts. (As a consequence, this physics engine does a good job of modeling brief collisions, but it suffers from artifacts in simulating sustained contact forces, including wheels on the ground and fingers grasping objects). To include the dynamic effects of collisions, the simulator must be able to deduce when and where (on each body) contact occurs. It is thus necessary to describe the envelope (boundary description) of 3-D objects in a manner suitable for efficient computation of contact sites. This part of the robot description is a "collision model." It can be as simple as a primitive 3D object description (e.g. a rectangular prism or a cylinder), or it can be high-fidelity model of a complex surface, typically originating from a CAD model. The surface descriptions are translated to faceted approximations (equivalent to a Stereolithography, or "STL" model, comprised of triangular facets), which is convenient for computing intersections (collisions) between model boundaries. A pragmatic concern with the collision model is that a large number of facets results in slower simulations.

The third categorty of model description is the visual model. The visual model is used for graphical display purposes. Having a graphical display of the computed dynamics can be very helpful in interpreting results and debugging software development. The visual model is often identical to the collision model, since the visual model also requires a boundary description of each object. It is common, however, for the visual model to be higher fidelity (e.g. incorporating more facets) than the collision model. Displaying a high-fidelity model is less demanding than computing collisions using a model with a large number of facets.

In the simplist case, the visual model can be identical to the collision model. However, these two categories can include their own options. The collision model, for example, can include specification of surface properties, including friction and resilience (which are irrelevant to visual appearance). The visual model may include specification of color, reflectivity and transparency (which are irrelevant to the collision model).

Some simple model descriptions are contained in the accompanying repository under the package (ROS folder) `exmpl_models`. Within the subfolder `rect_prism` of the `exmpl_models` package, the file `model-1_4.sdf` is shown in Listing 3.2.

Listing 3.2: Model description of a simple rectangular prism

```
<?xml version='1.0'?>
<sdf version='1.4'>
  <model name="rect_prism">
    <link name='link'>
      <inertial>
```

```
      <mass>2000</mass>
      <inertia>
          <ixx>3000</ixx>
          <ixy>0</ixy>
          <ixz>0</ixz>
          <iyy>3000</iyy>
          <iyz>0</iyz>
          <izz>1000</izz>
       </inertia>
      </inertial>
     <collision name='collision'>
        <geometry>
             <box>
               <size> 2 2 4 </size>
             </box>
       </geometry>
      </collision>

      <visual name='visual'>
        <geometry>
             <box>
               <size> 2 2 4 </size>
             </box>
       </geometry>
      </visual>
    </link>
  </model>
</sdf>
```

This file contains a model description using the Simulation Description Format ("SDF"–see `http://sdformat.org`). The model description is in XML format and contains three fields: inertial, collision, and visual. The inertial properties include a specification of both the mass (in kg) and the moments of inertia (in kg-m^2); explanation of inertial properties will be detailed later in this chapter. The collision and visual models are identical, in this simple case. They both specify a simple "box" (a rectangular prism) with x, y and z dimensions of 2, 2 and 4 meters, respectively. The rectangular prism has an associated coordinate frame defined to have its origin in the middle of the box, and with x, y, and z axes parallel to the respective specified dimensions.

The simple rectangular prism model is sufficient for performing interesting dynamic simulations. Modeling robots with joint controllers requires additional detail.

3.3 THE UNIFIED ROBOT DESCRIPTION FORMAT

A robot model can be specified using SDF. However, the older "Unified Robot Description Format" (URDF–see `http://wiki.ros.org/urdf`) is closely related, and most existing open-source robot models are expressed in URDF. Further, SDF models end up being translated into URDF for use with ROS. This section will thus describe robot modeling in URDF.

A minimal example is introduced here. A good tutorial for more detail can be found at `http://gazebosim.org/tutorials?tut=ros_urdf` for the "rrbot" (a 2-DOF robot arm). A URDF model can be used as input for dynamic simulation. The dynamic simulator used with ROS is "Gazebo" (see `http://gazebosim.org/`), which offers alternative choices for physics engines.

3.3.1 The Kinematic Model

A minimal kinematic model of a single-degree-of-freedom robot in the URDF style is given in Listing 3.3, which is file `links_and_joints.urdf` in the `minimal_robot_description` package of our accompanying repository. `minimal_robot_description.urdf`).

Listing 3.3: A 1-DOF Robot kinematic description

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

 <!-- Base Link -->
 <link name="link1" />
 <!--distal link -->
 <link name="link2" />

  <joint name="joint1" type="continuous">
    <parent link="link1"/>
    <child link="link2"/>
    <origin xyz="0 0 0.5" rpy="0 0 0"/>
    <axis xyz="0 1 0"/>
  </joint>
</robot>
```

In Listing 3.3, a robot called `one_DOF_robot` is defined, consisting of two “links” and one “joint.” Descriptions of links and joints are the minimal requirements of a robot definition. These elements are described in XML syntax, and they can be described in any order in the URDF. It is only necessary that the result is kinematically consistent.

A restriction on URDF files is that one cannot describe closed chains. A URDF will have a single, unique “base” link, and all other links form a tree relative to this base. This may be a single, open chain (like a conventional robot arm), or the tree may be more complex (such as a humanoid robot with legs, neck, and arms sprouting from a base link on the torso). Closed chains, such as a 4-bar linkage, can be “spoofed” in a URDF by making some links kinematically dependent and by introducing virtual actuators; this is adequate for visualization but inaccurate in terms of dynamics. The URDF robot description is not suitable for supporting dynamic simulation of closed-chain mechanisms.

An abstraction of a robot model requires only specifying spatial relationships among (solid) links. In the conventional “Denavit-Hartenberg” representation, this requires only one frame definition per link, and spatial relations between successive link frames are implied by only 4 values (3 fixed parameters and one joint variable) from parent link to child link. The URDF format departs from this compact description, using up to 10 values (9 fixed parameters and one joint variable) instead of the minimum 4 values. These parameters are described in on-line ROS tutorials; an alternative description of the URDF conventions is offered here.

Each link within a URDF (except for the base link) has a single parent link. However, a link may have multiple child links, e.g. such as multiple fingers extending from the base of a hand, or arms and legs extending from a torso.

Each link has a single “link” reference frame, which is rigidly associated with the link and moves with the link as the link moves through space. (The link frame may or may not be located within some part of the physical body of the link, but it nonetheless moves with the link conceptually, as though rigidly attached to some part of the physical body of the link). In addition to the link frame, there may be one or more “joint” frames that help to describe how children of the link are kinematically related. (Note: in Denavit-Hartenberg notation, there is no such thing as a “joint” frame; a joint frame is a URDF construct, which may be convenient in some instances, but also introduces confusing redundancy).

In Listing 3.3, link1 is the base frame, and link2 is a child of link1. The link frame for the base link is arbitrary–typically a consequence of whatever reference is chosen when describing this link in a CAD system. The respective link frames associated with all other (non-base) links, however, are constrained in their placements by URDF conventions.

For our base link, a “joint” frame is defined by the lines:

```
<joint name="joint1" type="continuous">
  <parent link="link1"/>
  <child link="link2"/>
  <origin xyz="0 0 0.5" rpy="0 0 0"/>
  <axis xyz="0 1 0"/>
</joint>
```

The "joint" frame is named "joint1." This frame is also associated with link1 (the parent link) and moves rigidly with link1. The position and location of this frame is specified by describing the displacement and rotation of the joint frame relative to the parent (link1) reference frame. This spatial relationship is specified by the line:

```
  <origin xyz="0 0 0.5" rpy="0 0 0"/>
```

This specification declares that the origin of the joint1 frame is offset from the link1 frame by the vector [0,0,1], i.e. aligned with the x and y coordinates of the link1 frame, but offset by 0.5m in the link1 z direction. (By default, all units in a URDF are MKS).

We must also specify the orientation of the joint1 frame. In this case, the joint1 frame is orientationally aligned with the link1 reference frame. (i.e., the respective x, y and z axes of these two frames are parallel). This is declared via the specification `rpy="0 0 0"`, which says that the joint1 frame orientation relative to the link1 reference frame is describable with zero roll, pitch and yaw angles.

The joint1 frame is useful as an intermediate frame for describing how link2 is related to link1. The line:

```
<joint name="joint1" type="continuous">
```

declares that the pose of link2 is related to link1 through a revolute joint. If unspecified, then the joint axis of this revolute joint lies colinear with the x-axis of the defined joint frame. (Note: this is in contrast to Denavit-Hartenberg convention, in which a joint axis is always defined as a "z" axis). In our example, the line:

```
 <axis xyz="0 1 0"/>
```

specifies that the joint axis is not aligned with the joint-frame x axis but is instead colinear with the joint-frame's y axis. The 3 components of the `<axis xyz="0 1 0"/>` specification allow for defining an arbitrary joint-axis orientation relative to the joint frame (where the three components specify a unit vector direction in space). Although the joint axis can be oriented arbitrarily, the URDF convention requires that the joint axis pass through the joint-frame's origin. (Note: orientation of the joint axis specifies more than a line in space; as a direction vector, it also implies a positive sense of rotation for the child link relative to the parent link about a joint axis colinear with this vector).

Having defined a joint, the reference frame for the child link follows implicitly. This is an important conceptual constraint (similar to the Denavit-Hartenberg convention for assigning link frames). In the URDF convention, a child link's reference frame must have its origin coincident with the joint frame that connects the child to the parent. In the simple example provided, the origin of the link2 frame is defined to be coincident with the origin of the specified joint1 frame.

The definition of the orientation of the link2 frame is also constrained by convention, based on the joint1 frame and definition of a "home" angle for joint1. Since a revolute joint joins link2 to link1, link2 can only move by rotating about the joint axis defined within the joint1 frame. When the robot moves with this single degree of freedom, the variable angle is known as the "joint angle" of joint1. The "home" angle (or zero angle) for joint1 is a definition, typically chosen to be something convenient (e.g., the 0 reading of the associated rotational sensor, or a convenient alignment that is easy to visualize). When a home angle is chosen, the child link's reference frame follows. It is defined to be coincident with its parent's joint frame. That is, if one puts link2 in the defined home position relative to link1, then the value of the joint1 angle will be defined to be zero at this pose, and the link2 reference frame will be identical to the joint1 reference frame at this pose. At any other (i.e. non-zero, non-periodic) values of joint1-angle, the joint1 and link2 frames will not be aligned (although their respective origins will remain coincident).

Our example URDF file thus defines a link frame for each link, and it defines how these link frames are constrained to move relative to each other. We can check if our URDF file is consistent by running:

```
check_urdf links_and_joints.urdf
```

In the above, either run this from the `minimal_robot_description` directory, or provide a path as part of the filename argument to `check_urdf`. The output produced is:

```
robot name is: one_DOF_robot
---------- Successfully Parsed XML ---------------
root Link: link1 has 1 child(ren)
    child(1):  link2
```

This confirms that our file has correct XML syntax and that the robot definition is kinematically consistent.

To this point, our URDF is consistent, but it only defines the constrained spatial relationship between two frames. For simulation purposes, we need to provide more information: a visual model, a collision model and a dynamic model.

3.3.2 The Visual Model

Our minimal kinematic model is augmented with visual information in Listing 3.4 (which also appears in the package `minimal_robot_description` as file `one_link_description.urdf` in our associated code repository).

Listing 3.4: A 1-Link Model with Visual Description

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

<!-- Used for fixing robot to the simulator's world frame -->
  <link name="world"/>

  <joint name="glue_robot_to_world" type="fixed">
    <parent link="world"/>
    <child link="link1"/>
  </joint>

 <!-- Base Link -->
  <link name="link1">
    <visual>
      <origin xyz="0 0 0.5" rpy="0 0 0"/>
      <geometry>
        <box size="0.2 0.2 1"/>
      </geometry>
    </visual>
  </link>
</robot>
```

For simulation purposes, a “world” frame with a “ground plane” will be defined, and we can place link1 fixed in the world by defining a “static” joint, lines 7-10:

```
<joint name="glue_robot_to_world" type="fixed">
  <parent link="world"/>
  <child link="link1"/>
</joint>
```

In defining joint `glue_robot_to_world`, the joint type is specified as “fixed”, which means our link1 will have a static relationship with respect to the world frame. Further, since we did not specify the x,y,z or r,p,y coordinates of the joint frame, the default values (all 0’s) are applied, and our link1-frame is thus defined to be identical to the world frame.

Within the link1 definition, in the sub-section inside the `<visual>` tags, lines 16-18

```
    <geometry>
      <box size="0.2 0.2 1"/>
    </geometry>
```

define a 3-D “box” entity with dimensions 0.2 by 0.2 by 1.0 meters. For this simple 3-D primitive (essentially identical to our earlier `rect_prism` model), a model frame is defined with origin at the center of the box, and x,y,z axes along the specified dimensions 0.2,0.2,1.0. This entity can be used to define the visual appearance of link1. (More commonly an entire CAD file is referenced to define each robot element, but simple 3-D primitives such as “box” are useful for quick models and for simple illustration of concepts).

Although we have defined a link1 frame (which is coincident with the world frame), we also need to define a “visual” frame for link1, such that our visual model (the box) appears in the correct location. Our link1 frame has its origin at the level of the ground plane, whereas our box entity has its origin in the center of the box. We wish to place the appearance of the box such that it appears to sit on one (square) face (i.e. with this face coplanar with the ground plane) and with the long dimension of the box (the visual-frame z axis) pointing up (i.e., normal to the ground plane and parallel to the z axis of the world frame). This is accomplished with line 15:

```
<origin xyz="0 0 0.5" rpy="0 0 0"/>
```

which specifies that the box-frame axes are parallel to the respective link1 frame axes, but that the box-frame origin is elevated (offset in the link1-frame z direction) by 0.5m.

We cannot test our visualization in our simulator yet until we augment the model with additional, dynamic information.

3.3.3 The Dynamic Model

For our robot model to be consistent with a physics engine, we must define mass properties for every link in the system. (Although it should not be necessary conceptually, a dynamic model is required even for link1, which we intend to fix rigidly to the ground plane). Our one-link model, augmented with mass-property information, is shown in Listing 3.5.

Listing 3.5: A 1-Link Model with Visual and Inertial Descriptions

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

<!-- Used for fixing robot to the simulator's world frame -->
```

```
5    <link name="world"/>
6
7    <joint name="glue_robot_to_world" type="fixed">
8      <parent link="world"/>
9      <child link="link1"/>
10   </joint>
11
12  <!-- Base Link -->
13   <link name="link1">
14     <visual>
15       <origin xyz="0 0 0.5" rpy="0 0 0"/>
16       <geometry>
17         <box size="0.2 0.2 1"/>
18       </geometry>
19     </visual>
20     <inertial>
21       <origin xyz="0 0 0.5" rpy="0 0 0"/>
22       <mass value="1"/>
23       <inertia
24         ixx="1.0" ixy="0.0" ixz="0.0"
25         iyy="1.0" iyz="0.0"
26         izz="1.0"/>
27     </inertial>
28   </link>
29   </link>
30 </robot>
```

Within the \link tags of the link1 description, the new additional lines 20-27 describe the mass properties:

```
    <inertial>
      <origin xyz="0 0 0.5" rpy="0 0 0"/>
      <mass value="1"/>
      <inertia
        ixx="1.0" ixy="0.0" ixz="0.0"
        iyy="1.0" iyz="0.0"
        izz="1.0"/>
    </inertial>
```

In the <inertial> field, one specifies the mass of the link (here, set to 1kg) and the coordinates of the center of mass (here, xyz="0 0 0.5"). The coordinates of the center of mass are specified in the link1 frame. For our box example, if the box has uniform density, then the center of mass will be in the center of the box, which is 0.5m above the link1 frame.

Specifying rotational inertia properties is more complex. Specification of rotational inertia requires defining a coordinate frame with respect to which the inertial properties are computed. With respect to this inertial frame, the components of a 3x3 matrix inertia tensor may be computed as:

$$\mathbf{I} = \int_V \rho(x,y,z) \begin{bmatrix} x^2+y^2 & -xy & -xz \\ -xy & z^2+x^2 & -yx \\ -xz & -yz & x^2+y^2 \end{bmatrix} dxdydz \tag{3.1}$$

where $\rho(x,y,z)$ is the density of material at location (x,y,z), and the integral over volume V is defined as the volume of all material comprising the rigid link of interest. The matrix $\mathbf{I}$ is always symmetric, and thus there are only 6 values to specify. For simple shapes, this moment of inertia tensor is easy to compute. Many common shapes have published tabulated values. The inertia tensor can be difficult to compute for complex link shapes. If the link is detailed in a CAD program, it is typical that the CAD program can compute the inertia tensor numerically.

Note that the matrix $\mathbf{I}$ will have different numerical components for different positions and/or orientations of the inertial reference frame. In a URDF, the origin of the inertial frame is always coincident with the center of mass (which is typically convenient in dynamics). It is sometimes convenient to define the orientation of this inertial frame to be aligned with some axes of symmetry, which simplifies specification of inertial components. In the present case (and often, in URDFs) the inertial frame is simply chosen to be parallel to the link frame (as specified by the rpy values in `<origin xyz="0 0 0.5" rpy="0 0 0"/>`).

With respect to the defined reference frame, the rotational inertial components can be specified in the URDF as in lines 23-27. The terms of $\mathbf{I}$ are labeled in the URDF as ixx=$\mathbf{I}_{1,1}$, ixy=$\mathbf{I}_{1,2} = \mathbf{I}_{2,1}$, ixz=$\mathbf{I}_{1,3} = \mathbf{I}_{3,1}$, iyy= $\mathbf{I}_{2,2}$, iyz=$\mathbf{I}_{2,3} = \mathbf{I}_{3,2}$ and izz= $\mathbf{I}_{3,3}$.

Ideally, these values will be a close approximation to the true inertial components of the actual links. One should attempt to at least estimate these values roughly. It is important, though, that neither the mass nor the diagonal components of inertia be assigned a value of 0. The physics engine will have divide-by-zero numerical problems in trying to simulate the dynamics of massless or inertialess objects.

For our single-link URDF, the mass properties have been assigned to unity (1kg mass, and 1kg-m^2 rotational inertia about each of the x, y and z principal axes). The inertial values are not realistic for a uniform rectangular prism. However, since this link will be fixed to the ground plane, the accurate mass values will not be a concern.

At this point, our model is boring (a static, rectangular prism). Nonetheless, it is a viable URDF that could be loaded into a simulator. This will be deferred, though, until our model becomes more interesting by adding a movable link. Listing 3.6 combines our initial kinematic model with visual and inertial properties.

Listing 3.6: A 1-DOF Robot URDF description

```
1  <?xml version="1.0"?>
2  <robot  name="one_DOF_robot">
3
4  <!-- Used for fixing robot to the simulator's world frame -->
5    <link name="world"/>
6
7    <joint name="glue_robot_to_world" type="fixed">
8      <parent link="world"/>
9      <child link="link1"/>
10   </joint>
11
12  <!-- Base Link -->
13   <link name="link1">
14     <visual>
15       <origin xyz="0 0 0.5" rpy="0 0 0"/>
16       <geometry>
17         <box size="0.2 0.2 1"/>
18       </geometry>
19     </visual>
20
21     <inertial>
22       <origin xyz="0 0 0.5" rpy="0 0 0"/>
23       <mass value="1"/>
24       <inertia
25         ixx="1.0" ixy="0.0" ixz="0.0"
26         iyy="1.0" iyz="0.0"
27         izz="1.0"/>
28     </inertial>
29   </link>
30
31 <!-- Moveable Link -->
32   <link name="link2">
33     <visual>
34       <origin xyz="0 0 0.5" rpy="0 0 0"/>
35       <geometry>
36         <cylinder length="1" radius="0.1"/>
37       </geometry>
38     </visual>
39
40     <inertial>
41       <origin xyz="0 0 0.5" rpy="0 0 0"/>
42       <mass value="1"/>
```

```
43       <inertia
44         ixx="0.1" ixy="0.0" ixz="0.0"
45         iyy="0.1" iyz="0.0"
46         izz="0.005"/>
47     </inertial>
48   </link>
49
50   <joint name="joint1" type="continuous">
51     <parent link="link1"/>
52     <child link="link2"/>
53     <origin xyz="0 0 1" rpy="0 0 0"/>
54     <axis xyz="0 1 0"/>
55   </joint>
56 </robot>
```

In Listing 3.6, visual and dynamic models are defined for two links. The joint between these links, joint1, could include additional properties, e.g. to model viscous or Coulomb friction. Further, joint limits and actuator torque limits could be specified. However, the minimal model that we have is adequate for simulating robot dynamics subject to specified joint (actuator) torques and the influence of gravity.

The visual model for link2 is defined on line 36:

```
<cylinder length="1" radius="0.1"/>
```

as a cylinder of length 1m and radius 0.1m. The inertial properties for link2 are specified as mass=1kg, ixx= iyy = 0.1 and izz=0.005. These values are reasonable approximations. For a thin rod of uniform density, mass m andlength l, the rotational inertia about its x or y axes is $I_{xx} = I_{yy} = (1/12)ml^2$. The assigned values have been rounded up to 0.1 m^2kg. The inertia of this rod spinning about its cylindrical axis is $I_{zz} = (1/2)mr^2$, which evaluates to the assignment of izz=0.005m^2kg.

As with the "box" entity, the cylinder entity's reference frame has its origin in the middle of the cylinder. The z-axis of this visualization frame points along the cylinder's major axis.

As introduced in Subsection 3.3.1, the link2 frame is defined to be coincident with the joint1 frame when the angle of joint1 is in its home (0) position. The reference frame of our visual model is specified relative to the link2 frame in line 34: <origin xyz="0 0 0.5" rpy="0 0 0"/>. That is, the visual frame is aligned with the link2 frame (rpy="0 0 0"), but the origin of the visual frame of link2 is offset from the link2 reference from by 0.5m along the link2 z axis (xyz="0 0 0.5"). As with our box visual description, this 0.5m offset locates the geometric (visual) model such that one endcap of the model coincides with a joint origin (in this case, joint1).

In the "home" position, the link2 frame is aligned with the joint1 frame. With our choices, the home position of our 1-DOF, 2-link robot corresponds to link2 pointing straight up.

3.3.4 The Collision Model

Our minimal robot description so far has included a kinematic model, an inertial model and a visual model. Motion of our robot will depend on forces and moments exerted on the robot. The dynamics engine of our simulator will enforce the kinematic constraints (e.g., how link2 can move with respect to link1) and will compute angular accelerations of link2 about joint1. These accelerations will be due to effects including gravity, joint torque exerted by an actuator (once we have defined an actuator), and possible collisions with other bodies. To include the influence of contact forces (e.g. due to collisions), we must include a "collision model" in our URDF.

A <collision> tag defines the region in which the collision model is defined in the URDF. Our 1-DOF URDF with collision properties appears in Listing 3.7. 3.7 (which also appears in the package **minimal_robot_description** as file **minimal_robot_description.urdf** in our associated code repository).

Listing 3.7: A 1-DOF Robot URDF description

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

<!-- Used for fixing robot to the simulator's world frame -->
  <link name="world"/>

  <joint name="glue_robot_to_world" type="fixed">
    <parent link="world"/>
    <child link="link1"/>
  </joint>

 <!-- Base Link -->
  <link name="link1">
    <collision>
      <origin xyz="0 0 0.5" rpy="0 0 0"/>
      <geometry>
       <box size="0.2 0.2 0.7"/>
      </geometry>
    </collision>

    <visual>
      <origin xyz="0 0 0.5" rpy="0 0 0"/>
      <geometry>
        <box size="0.2 0.2 1"/>
      </geometry>
    </visual>

    <inertial>
      <origin xyz="0 0 0.5" rpy="0 0 0"/>
      <mass value="1"/>
      <inertia
        ixx="1.0" ixy="0.0" ixz="0.0"
        iyy="1.0" iyz="0.0"
        izz="1.0"/>
    </inertial>
  </link>

<!-- Moveable Link -->
  <link name="link2">
    <collision>
      <origin xyz="0 0 0.5" rpy="0 0 0"/>
      <geometry>
        <cylinder length="1" radius="0.1"/>
        <!--box size="0.15 0.15 0.8"-->
      </geometry>
    </collision>

    <visual>
      <origin xyz="0 0 0.5" rpy="0 0 0"/>
      <geometry>
        <cylinder length="1" radius="0.1"/>
      </geometry>
    </visual>

    <inertial>
      <origin xyz="0 0 0.5" rpy="0 0 0"/>
      <mass value="1"/>
      <inertia
        ixx="0.1" ixy="0.0" ixz="0.0"
        iyy="0.1" iyz="0.0"
        izz="0.005"/>
    </inertial>
  </link>

  <joint name="joint1" type="continuous">
    <parent link="link1"/>
    <child link="link2"/>
```

```
    <origin xyz="0 0 1" rpy="0 0 0"/>
    <axis xyz="0 1 0"/>
  </joint>
</robot>
```

In Listing 3.7, the collision models define the geometric detail corresponding to a "skin" on the links. The collision model is used to compute intersections between solids within a world model; such collisions lead to interaction forces/torques at the points of contact.

Often, the collision model is identical to the visual model (and this is the case in Listing 3.7). However, collision-checking can be a computationally-intensive process, and therefore the collision model should be as sparse as possible. This can be done by reducing the number of triangles in a tessellated surface model, or by creating a primitive collision model based on geometric solids, e.g. rectangular prisms, cylinders or spheres.

Another concern for the collision model is that a collision model that does not offer adequate clearance between the links can result in simulation instability, as the simulator repeatedly determines that the links are colliding with each other. For our crude model, we will set the collision model of link2 to be identical to the visual model of link2–a simple cylinder. However, we will set the collision model of link1 to be a shorter box, thus providing clearance for link2.

Since link1 is stationary, fidelity of its collision model is not a concern for computing dynamics of this minimal robot. However, if this model were part of a finger, one would care about how this link contacts objects to be grasped. Alternatively, if there were additional robots in the virtual world, one would care about how these robots might collide with link1 of this minimal robot, as such collisions would affect the dynamics of the other robots.

Our one-DOF minimal robot URDF now contains kinematic, inertial, visual and collision information. We next introduce the "Gazebo" simulator, which can perform dynamic simulations of robots based on URDF specifications.

3.4 INTRODUCTION TO GAZEBO

"Gazebo" is the simulator used with ROS (see `http://gazebosim.org/`). Gazebo offers options for alternative physics engines, with a default of "ODE" (Open Dynamics Engine). The Gazebo simulator consists of two parts: a server (which runs as process "gzserver") and a client (which runs as process "gzclient"). The client process presents a graphical display and human interface. However, Gazebo can be run "headless" if a visual display is not needed.

An impressive and valuable capability of Gazebo is that it can simulate sensors as well as dynamics, including force sensors, accelerometers, sonar, LIDARs, color cameras, and 3-D point-cloud sensors. Description of camera simulation will be deferred to Part III.

Gazebo simulations require a world model in addition to one or more robot models. The world model may contain details of terrain, buildings, barriers, tables, graspable objects, and additional active entities, including swarms of robots. To start simple, however, we consider our minimal robot in a minimal world.

A common default world model consists only of a flat ground plane that is oriented perpendicular to the direction of gravity. To start up Gazebo with this empty world model, run:

```
roslaunch gazebo_ros empty_world.launch
```

This starts a launch file called `empty_world.launch` from the ROS package `gazebo_ros`. The result of this command is to start up both the gzserver and the gzclient, presenting a

Gazebo display of an empty world (except for a ground plane). The Gazebo window will appear as in Fig 3.7.

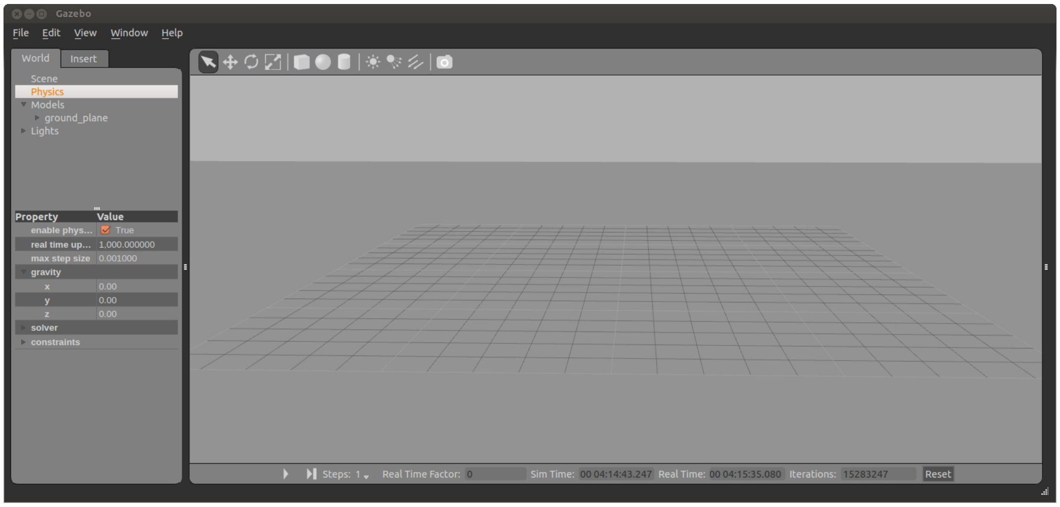


FIGURE 3.7: Gazebo display of an empty world, with gravity set to 0

Along the bottom bar of the Gazebo simulator are some controls and displays. A run/-pause button allows the user to suspend a simulation and resume it at will. The Real Time Factor display indicates the efficiency of the simulation. Gazebo will (by default) try to simulate the robot(s) in the virtual world in real time. If the host computer is not able to keep up with the required computations to achieve real time, the simulator will slow down, resulting in an apparent slow-motion output. If the Real Time Factor is unity, then the simulation is equivalent to real-time dynamics; if this factor is, e.g., 0.5, then the simulation will take twice as long as reality.

On the left side of the Gazebo display, as shown in Fig 3.7, are tabs labeled "world" and "insert." The "World" tab contains the option "Physics." This element has been expanded, which displays various properties, including the "gravity" item. The gravity item has also been expanded, revealing the numerical values for the x, y and z components of gravity. In this example, the z component was changed from its initial value of -9.8 to 0 (which is done by clicking on the the displayed value and editing it). With gravity set to zero, models can float in space indefinitely, whereas when gravity has a negative z component, models will fall to the ground plane.

Also within the Gazebo window's "World" tab is a "Models" menu, which allows one to inspect the spatial and dynamic properties of all models within the simulation. At this point, the only model in the simulation is a ground plane. We can add models to the simulation in multiple ways. Using the "insert" tab of Gazebo, one can select any of the pre-defined models displayed in the model list to be inserted in the simulation. The list of available models will include on-line models in Gazebo's database, as well as models defined locally that reside within the (hidden) directory `~/.gazebo`.

Alternatively, one can load a model into gazebo manually by invoking a gazebo node with a path to a model. For example, first navigate to the directory `~/ros_ws/src/learning_ros/exmpl_models/rect_prism`, then enter the following command:

```
rosrun gazebo_ros spawn_model -file model-1_4.sdf -sdf -model rect_prism
```

Our simple, rectangular prism model will be loaded into Gazebo, and the display will as in Fig 3.8. We can remove this model by clicking on it, then pressing the “delete” key.

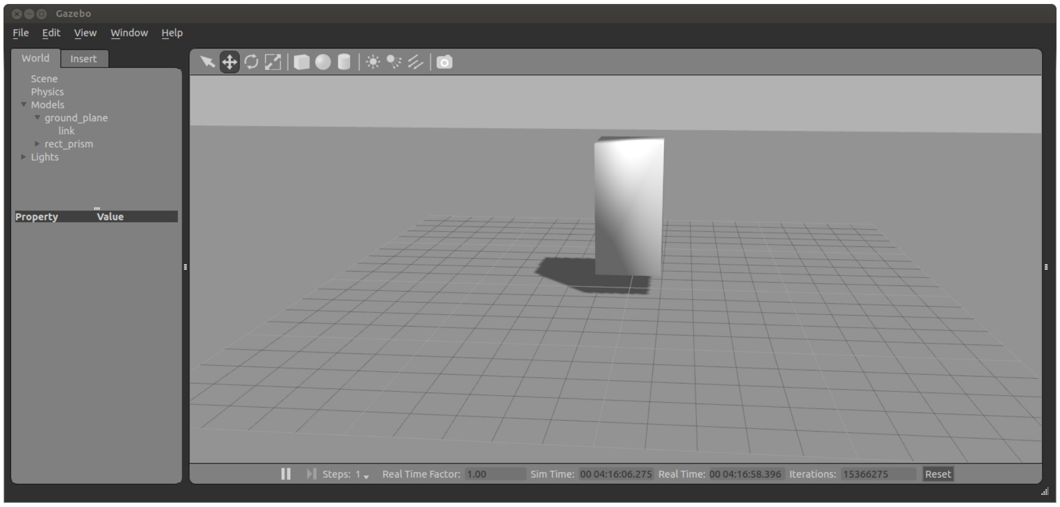


FIGURE 3.8: Gazebo display after loading the rectangular-prism model

Alternatively, the prism can be loaded with specified world coordinates. For example, the command:

```
rosrun gazebo_ros spawn_model -file model-1_4.sdf -sdf -model rect_prism -x 0 -y 0 -z ↩
    4
```

loads the prism with its coordinate frame located 4 meters above the ground plane.

Instead of navigating to the model directory to load models, one can include the full path to the model location. This is made somewhat more general with use of the environment variable `ROS_WORKSPACE`. The following command loads the model using an explicit path:

```
rosrun gazebo_ros spawn_model -file $ROS_WORKSPACE/src/learning_ros/exmpl_models/↩
    rect_prism/model-1_4.sdf -sdf -model rect_prism
```

It can me more convenient to ask ROS to find the package in which the model resides, which can be done with `$(rospack find package_name)`, e.g. as with the following command:

```
rosrun gazebo_ros spawn_model -file $(rospack find exmpl_models)/rect_prism/model-1_4.↩
    sdf -sdf -model rect_prism
```

Since direct commands can be tedious to type in, it is typically more convenient to enter the commands in a launch file, then invoke the launch file to load models. An example is in `exmpl_models/launch/add_rect_prism.launch`. The contents of this file are:

```
<launch>
<node name="spawn_sdf" pkg="gazebo_ros" type="spawn_model" args="-file $(find ↩
    exmpl_models)/rect_prism/model-1_4.sdf -sdf -model rect_prism -x 0 -y 0 -z 5" />
</launch>
```

The syntax is similar to the command line, though with variations. Within a launch file, one finds ROS packages using `$(find package_name)` instead of `$(rospack find package_name)`. This launch file can be invoked from any directory with:

```
roslaunch exmpl_models add_rect_prism.launch
```

This command finds the package `exmpl_models`, implicitly looks in the sub-directory `launch`, and invokes the launch file `add_rect_prism.launch`. This launch file loads the rectangular-prism model into Gazebo with initial coordinates of (x,y,z) = (0,0,5). With gravity off, this model remains stationary, floating above the ground plane.

A second simple model (a cylinder) can be added with a similar launch file:

```
roslaunch exmpl_models add_cylinder.launch
```

which results in the Gazebo display in Fig 3.9. Note that the `Models` menu now includes

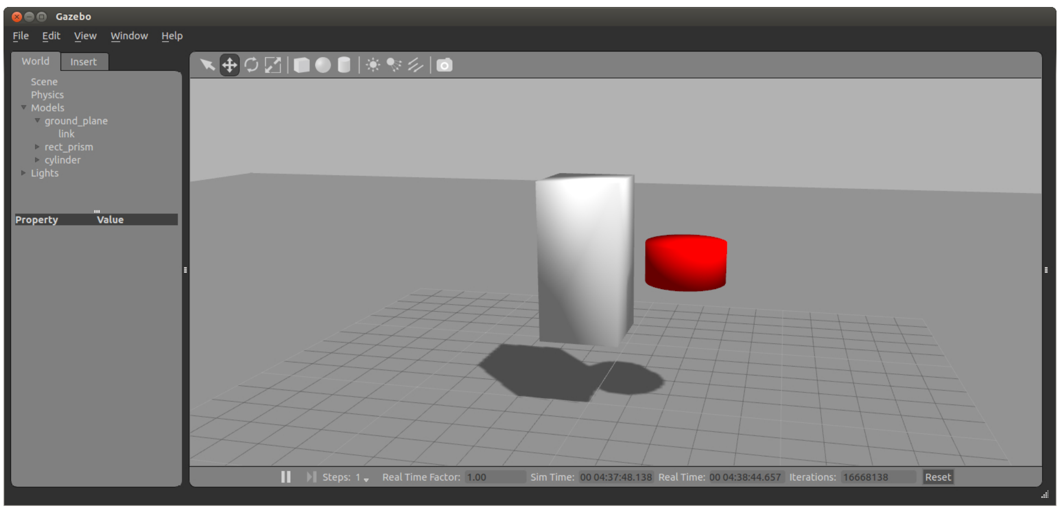


FIGURE 3.9: Gazebo display after loading a rectangular prism and a cylinder

'`rect_prism` and `cylinder`.

At this point, we have confirmed that our models are syntactically correct, that they present the expected visual displays, and that they contain the minimum requirements to be compatible with Gazebo simulation. More interestingly, dynamic simulation can be observed by giving these models an initial velocity that results in collision, from which we can observe behavior of the physics engine. To initialize our models with a non-zero velocity and angular velocity, we can use a service of gazebo. Running:

```
roservice list
```

shows 30 services running, one of which is `/gazebo/set_model_state`. Examining this service with

```
rosservice info gazebo/set_model_state
```

shows that the service message expects an argument, `model_state`, of type `gazebo_msgs/SetModelState`.

Looking in the `gazebo_msgs/srv` directory, one finds the service-message description `SetModelState.srv`, the contents of which is:

```
gazebo_msgs/ModelState model_state
---
bool success                  # return true if setting state successful
string status_message         # comments if available
```

which confirms that the "request" field of this service message contains a field called `model_state` of type `gazebo_msgs/ModelState`.

The message type `gazebo_msgs/ModelState` can be examined with:

```
rosmsg show gazebo_msgs/ModelState
```

which displays details of this message type to be:

```
string model_name
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
geometry_msgs/Twist twist
  geometry_msgs/Vector3 linear
    float64 x
    float64 y
    float64 z
  geometry_msgs/Vector3 angular
    float64 x
    float64 y
    float64 z
string reference_frame
```

A model state can be set using a manual command. An example is:

```
rosservice call /gazebo/set_model_state '{model_state: {model_name: rect_prism, twist:↩
     {angular:{z: 1.0}}}}'
```

This command specifies the model name to be our rectangular prism, and it commands the z-component of the angular velocity to be 1.0 rad/sec. Implicitly, all other components of position, orientation, linear velocity and angular velocity are set to 0. The syntax for specifying components of a message type is YAML. See `http://wiki.ros.org/ROS/YAMLCommandLine` for details on using YAML within a ROS command line.

Since the YAML syntax can get tedious (and error prone for direct typing), it can be more convenient to set the model state programmatically. An example of how to do this is contained in the package `example_gazebo_set_state` in the source code (node) `example_gazebo_set_prism_state`.

Key lines of this node include:

```
ros::ServiceClient set_model_state_client =
       nh.serviceClient<gazebo_msgs::SetModelState>("/gazebo/set_model_state");
```

A compatible service message is instantiated with the line:

```
gazebo_msgs::SetModelState model_state_srv_msg;
```

Components of this message are filled in, e.g. as:

```
model_state_srv_msg.request.model_state.model_name = "rect_prism";
```

which specifies the model for which the state is to be set, and:

```
model_state_srv_msg.request.model_state.twist.angular.z= 1.0;
```

which specifies a z-component of angular velocity to be 1.0 rad/sec. Similarly, all components of position, orientation, translational velocity and angular velocity can be specified. After the service message is populated, it is sent to the gazebo service to set the specified model state with:

```
set_model_state_client.call(model_state_srv_msg);
```

This program can be run by entering:

```
rosrun example_gazebo_set_state example_gazebo_set_prism_state
```

which causes the prism to rotate about its z axis and translate slowly in the x direction. After a single service call, this program concludes. The objects the evolve in time according to their twist vectors.

Using the initial conditions within the launch files for loading a prism and a cylinder, then invoking `example_gazebo_set_prism_state`, the prism will begin spinning and translating, eventually colliding with the cylinder. This collision results in a change in momentum (including angular momentum) of both objects. The states of the models can be observed with:

```
rostopic echo gazebo/model_states
```

Before the models collide, this will show that the cylinder has zero twist, but the prism has an angular velocity about its z axis of 1 rad/sec, as well as a translational velocity in the x direction of 0.02 m/sec. After the models collide, they both have altered translational and rotational velocities. However, it can be shown that the total system linear momentum and system angular momentum is preserved. (Momentum before collision equals momentum after collision). This illustrates that the physics engine behaves as expected. Although the objects may be translating and tumbling in a complex way after collision, the system momentum is conserved.

Robot models can be inserted and dynamically modeled in a similar fashion, although the model and its stimulus are more complex. To import our minimal robot model, in a separate terminal, navigate to the `minimal_robot_description` package and enter:

```
rosrun gazebo_ros spawn_model -urdf -file minimal_robot_description.urdf -model ↩
    one_DOF
```

This invokes the `spawn_model` node from the `gazebo_ros` package with three arguments: declaration that the input file is in URDF format, specification of the urdf filename to be loaded, and a model name to assign to the loaded file in Gazebo. This node will run to completion, resulting in inserting the named URDF file into the simulator. The result appears as

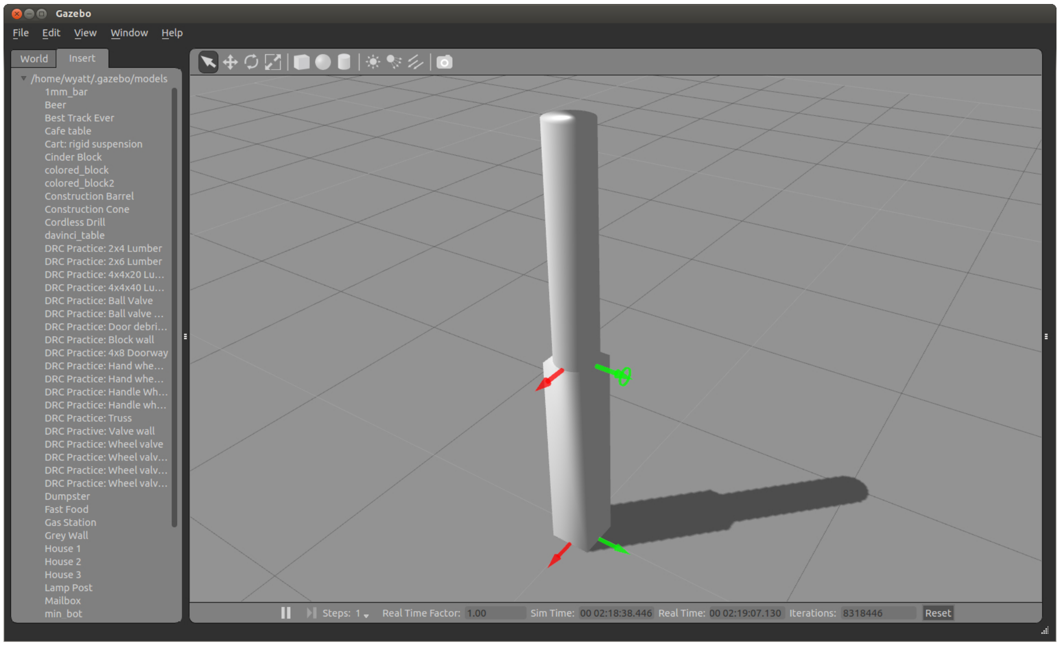


FIGURE 3.10: Gazebo display of a two-link, 1-DOF robot URDF model

in Fig 3.10. From Fig 3.10, it is verified that we have an upright, rectangular prism for link1, and we have a cylindrical link2. The coordinate frames for link1 and link2 are illustrated as well, where the red axes are x axes and the green axes are y axes. The green circle about the link2 y axis indicates that this vector is also a joint axis (the axis of joint1). This display was enabled via the top menu bar of the Gazebo display by enableing `view->Joints`. The Gazebo menus offer a variety of additional options that can be convenient for visualizing simulations, including centers of mass and contact forces.

Use of the "Insert" tab in Gazebo can be convenient for constructing variations on the virtual world for experiments with the robot. A variety of models are available on-line to import. Additionally, Gazebo will look in the (hidden) user directory called `.gazebo`, which typically resides in the user's Home directory. Alternatively, one can explicitly load a file from a named directory, as described above for our simple prism and

With Gazebo running, entering `rostopic list` reveals that the active topics include:

```
/clock
/gazebo/link_states
/gazebo/model_states
/gazebo/parameter_descriptions
/gazebo/parameter_updates
/gazebo/set_link_state
/gazebo/set_model_state
```

Entering `rosservice list` shows that the following gazebo services are active:

```
/gazebo/apply_body_wrench
/gazebo/apply_joint_effort
/gazebo/clear_body_wrenches
/gazebo/clear_joint_forces
/gazebo/delete_model
```

```
/gazebo/get_joint_properties
/gazebo/get_link_properties
/gazebo/get_link_state
/gazebo/get_loggers
/gazebo/get_model_properties
/gazebo/get_model_state
/gazebo/get_physics_properties
/gazebo/get_world_properties
/gazebo/pause_physics
/gazebo/reset_simulation
/gazebo/reset_world
/gazebo/set_joint_properties
/gazebo/set_link_properties
/gazebo/set_link_state
/gazebo/set_logger_level
/gazebo/set_model_configuration
/gazebo/set_model_state
/gazebo/set_parameters
/gazebo/set_physics_properties
/gazebo/spawn_gazebo_model
/gazebo/spawn_sdf_model
/gazebo/spawn_urdf_model
/gazebo/unpause_physics
```

Running `rostopic echo gazebo/link_states` displays updates of the 6-D pose and 6-D velocity of each link in the system. The intial part of the display looks like:

```
name: ['ground_plane::link', 'one_DOF::link1', 'one_DOF::link2']
pose:
  -
    position:
      x: 0.0
      y: 0.0
      z: 0.0
    orientation:
      x: 0.0
      y: 0.0
      z: 0.0
      w: 1.0
```

The output declares that there are three links in the system (including the ground plane). The position and orientation of every link is given, as well as the linear and angular velocity vectors. For the simulation at present, these values are boring. All velocities are zero, and all links are aligned with the world frame (except for link2 being elevated by 1.0m).

The initial pose in this scene is the home pose, corresponding to link2 straight up. In Fig 3.10, link2 is precariously balanced at an unstable equilibrium point. There is no joint controller keeping it upright, and it may unexpectedly tip over, tilting about the joint1 axis.

To make our robot more interesting, a joint controller is needed. The joint controller can integrate with Gazebo, obtainint joint angles and inducing joint torques, simulating a servoed actuator.

3.5 A MINIMAL JOINT CONTROLLER

An important connection between Gazebo and ROS is how controls are exerted using joint actuator commands and joint displacement sensors. For purposes of illustrating this interaction, a minimal joint controller ROS node is presented. The source code `minimal_joint_controller.cpp` (in the `minimal_joint_controller` package) is described here, which interacts with Gazebo and creates a ROS interface (similar to constructing the bridges necessary to interact with a real robot).

Please note that the example controller described here normally would not be used. For a physical robot, the PD controller would be contained within dedicated control hardware. Similarly, for Gazebo, there are pre-defined Gazebo "plug-ins" that perform the equivalent of the present joint-controller example. (See `http://gazebosim.org/tutorials?tut=ros_plugins` for a tutorial on how to write Gazebo plug-ins). Thus, you would not need to use the `minimal_joint_controller` package; this is presented only to illustrate the equivalent of what a Gazebo controller plug-in performs, using concepts already introduced. A disadvantage of using the present controller code relative to a Gazebo plug-in is that the example code incurs additional computational and bandwidth loads of serialization/de-serialization and corresponding latency in message passing, which can be critical in high-performance control implementations.

The contents of `minimal_joint_controller.cpp` appears in Listing 3.8.

Listing 3.8: Minimal joint controller via Gazebo services

```
#include <ros/ros.h> //ALWAYS need to include this
#include <gazebo_msgs/GetModelState.h>
#include <gazebo_msgs/ApplyJointEffort.h>
#include <gazebo_msgs/GetJointProperties.h>
#include <sensor_msgs/JointState.h>
#include <string.h>
#include <stdio.h>
#include <std_msgs/Float64.h>
#include <math.h>

//a simple saturation function; provide saturation threshold, sat_val, and arg to be ↩
    saturated, val
double sat(double val, double sat_val) {
    if (val>sat_val)
        return (sat_val);
    if (val< -sat_val)
        return (-sat_val);
    return val;

}

double g_pos_cmd=0.0; //position command input-- global var
void posCmdCB(const std_msgs::Float64& pos_cmd_msg)
{
  ROS_INFO("received value of pos_cmd is: %f",pos_cmd_msg.data);
  g_pos_cmd = pos_cmd_msg.data;
}



int main(int argc, char **argv) {
    ros::init(argc, argv, "minimal_joint_controller");
    ros::NodeHandle nh;
    ros::Duration half_sec(0.5);

    // make sure service is available before attempting to proceed, else node will ↩
        crash
    bool service_ready = false;
    while (!service_ready) {
      service_ready = ros::service::exists("/gazebo/apply_joint_effort",true);
      ROS_INFO("waiting for apply_joint_effort service");
      half_sec.sleep();
    }
    ROS_INFO("apply_joint_effort service exists");

    ros::ServiceClient set_trq_client =
       nh.serviceClient<gazebo_msgs::ApplyJointEffort>("/gazebo/apply_joint_effort");

```

```
    service_ready = false;
    while (!service_ready) {
      service_ready = ros::service::exists("/gazebo/get_joint_properties",true);
      ROS_INFO("waiting for /gazebo/get_joint_properties service");
      half_sec.sleep();
    }
    ROS_INFO("/gazebo/get_joint_properties service exists");

    ros::ServiceClient get_jnt_state_client =
       nh.serviceClient<gazebo_msgs::GetJointProperties>("/gazebo/get_joint_properties");

    gazebo_msgs::ApplyJointEffort effort_cmd_srv_msg;
    gazebo_msgs::GetJointProperties get_joint_state_srv_msg;

    ros::Publisher trq_publisher = nh.advertise<std_msgs::Float64>("jnt_trq", 1);
    ros::Publisher vel_publisher = nh.advertise<std_msgs::Float64>("jnt_vel", 1);
    ros::Publisher pos_publisher = nh.advertise<std_msgs::Float64>("jnt_pos", 1);
    ros::Publisher joint_state_publisher = nh.advertise<sensor_msgs::JointState>("joint_states", 1);

    ros::Subscriber pos_cmd_subscriber = nh.subscribe("pos_cmd",1,posCmdCB);

    std_msgs::Float64 trq_msg;
    std_msgs::Float64 q1_msg,q1dot_msg;
    sensor_msgs::JointState joint_state_msg;

    double q1, q1dot;
    double dt = 0.01;
    ros::Duration duration(dt);
    ros::Rate rate_timer(1/dt);

    effort_cmd_srv_msg.request.joint_name = "joint1";
    effort_cmd_srv_msg.request.effort = 0.0;
    effort_cmd_srv_msg.request.duration= duration;

    get_joint_state_srv_msg.request.joint_name = "joint1";
    //double q1_des = 1.0;
    double q1_err;
    double Kp = 10.0;
    double Kv = 3;
    double trq_cmd;

    // set up the joint_state_msg fields to define a single joint,
    // called joint1, and initial position and vel values of 0
    joint_state_msg.header.stamp = ros::Time::now();
    joint_state_msg.name.push_back("joint1");
        joint_state_msg.position.push_back(0.0);
        joint_state_msg.velocity.push_back(0.0);
    while(ros::ok()) {
        get_jnt_state_client.call(get_joint_state_srv_msg);
        q1 = get_joint_state_srv_msg.response.position[0];
        q1_msg.data = q1;
        pos_publisher.publish(q1_msg);

        q1dot = get_joint_state_srv_msg.response.rate[0];
        q1dot_msg.data = q1dot;
        vel_publisher.publish(q1dot_msg);

    joint_state_msg.header.stamp = ros::Time::now();
        joint_state_msg.position[0] = q1;
        joint_state_msg.velocity[0] = q1dot;

    joint_state_publisher.publish(joint_state_msg);

        //ROS_INFO("q1 = %f;  q1dot = %f",q1,q1dot);
        //watch for periodicity
        q1_err= g_pos_cmd-q1;
        if (q1_err>M_PI) {
            q1_err -= 2*M_PI;
        }
        if (q1_err< -M_PI) {
            q1_err += 2*M_PI;
        }

        trq_cmd = Kp*(q1_err)-Kv*q1dot;
```

```
121         //trq_cmd = sat(trq_cmd, 10.0); //saturate at 1 N-m
122         trq_msg.data = trq_cmd;
123         trq_publisher.publish(trq_msg);
124         // send torque command to Gazebo
125         effort_cmd_srv_msg.request.effort = trq_cmd;
126         set_trq_client.call(effort_cmd_srv_msg);
127         //make sure service call was successful
128         bool result = effort_cmd_srv_msg.response.success;
129         if (!result)
130             ROS_WARN("service call to apply_joint_effort failed!");
131         ros::spinOnce();
132     rate_timer.sleep();
133   }
134 }
```

The example controller in Listing 3.8 interacts with Gazebo via service clients of the services `/gazebo/get_joint_properties` and `/gazebo/apply_joint_effort`. In lines 36-41, the nodes tests if the service `/gazebo/apply_joint_effort` is ready, after which a corresponding service client is instantiated (lines 44-45). Similarly, lines 47-52 test if the `/gazebo/get_joint_properties` service is ready, after which a corresponding client is instantiated (lines 55-56). Compatible service messages are instantiated in lines 58-59.

The service `/gazebo/get_joint_properties` can be examined manually by entering:

```
rosservice call /gazebo/get_joint_properties "joint1"
```

which results in the following example output:

```
type: 0
damping: []
position: [0.0]
rate: [0.0]
success: True
status_message: GetJointProperties: got properties
```

From this service, we can obtain the "state" of joint1, which includes the joint position and the joint (angular) velocity. The service client of this service, `get_jnt_state_client`, makes such calls repeatedly in the control loop to get the joint position and velocity from the dynamic simulator.

The service client of `/gazebo/apply_joint_effort`, `set_trq_client`, sets fields in the corresponding service message for `joint_name` to "joint1" and `effort` to a desired joint torque, enabling the controller node to impose joint torques on joint1 of the simulated robot.

A subscriber to the topic `pos_cmd` is also set up (line 66), ready to accept user input for desired joint1 position values via callback function "posCmdCB" (lines 22-26).

The main loop of our controller node does the following:

- get the current joint position and velocity from Gazebo and republish these on topic `joint_states` (lines 95-108)
- compare the (virtual) joint-sensor value to the commanded joint angle (from the `pos_cmd` callback), accounting for periodicity (lines 112-118).
- compute a torque P-D torque response (line 120)
- send this effort to Gazebo via the `apply_joint_effort` service (lines 125-126)

The minimal controller node can be started with the command:

```
rosrun minimal_joint_controller minimal_joint_controller
```

Initially, there is no noticeable effect on Gazebo, since the start-up desired angle is 0, and the robot is already at zero angle. However, we can command a new desired angle manually from the command line (and later, under program control) with the command:

```
rostopic pub pos_cmd std_msgs/Float64 1.0
```

which commands a new joint angle of 1.0 radians. The robot then moves to the position shown in Fig 3.11. We can record the dynamic response of an input command by plotting out

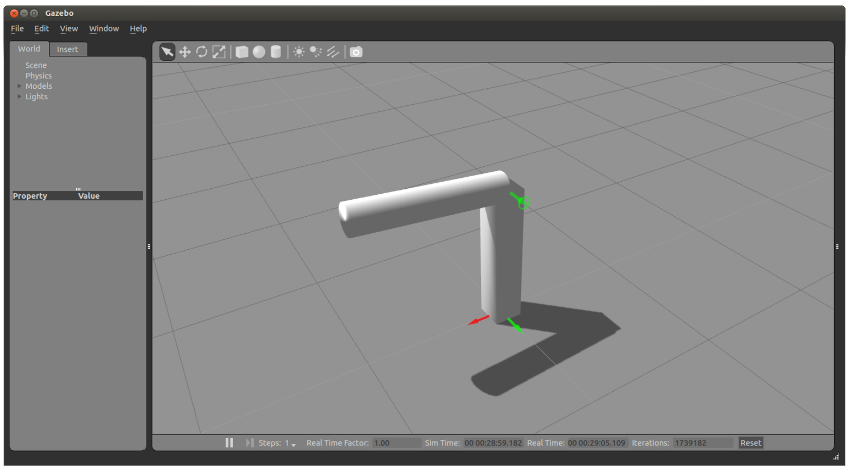


FIGURE 3.11: Gazebo display of minimal robot with minimal controller

the published values of joint torque, velocity and position, using `rqt_plot`. Enter the command `rqt_plot`, then add topics of `/jnt_pos/data`, `/jnt_trq/data` and `/jnt_vel/data`. The plot in Fig 3.12 shows the transient response, starting from a position command of 1.0, then responding to a new command of 2.0. As shown, the initial joint angle is larger than the commanded 1.0. This is due to a low feedback proportional gain and the influence of gravity, causing droop relative to the desired angle. At approximately t=52.8, the input position command is changed to 2.0 rad, resulting in a transient in joint torque as the link accelerates towards the new goal. The goal of 2.0 is overshot, again due to gravity load. As the link settles, a sustained torque of approximately -3.5N-m is required to hold the link against gravity.

To illustrate the influence of contact dynamics, an additional model is added to Gazebo, as shown in Fig 3.13. Here, a "cafe table" (from a list of pre-defined models) is inserted using the "insert" menu of Gazebo. Gazebo offers the user the capability of moving the table to a desired location, which was chosen to be within reach of the robot.

Next, the robot was commanded to position 0 (straight up), then commanded to position 2.5–which is unreachable with the table in the way. The transient dynamics from this command are shown in the Fig 3.14. The collision-model of the link and the collision model of the table are used by Gazebo to detect that contact has occurred. This results in a reaction force from the table to the robot (and from the robot to the table). As a result, the robot does not reach its goal angle, and equilibrium occurs with the robot's joint actuator exerting an effort downwards on the table, while struggling to reach the desired angle of 2.5 rad.

The ability to move models in Gazebo under program control can have useful applications

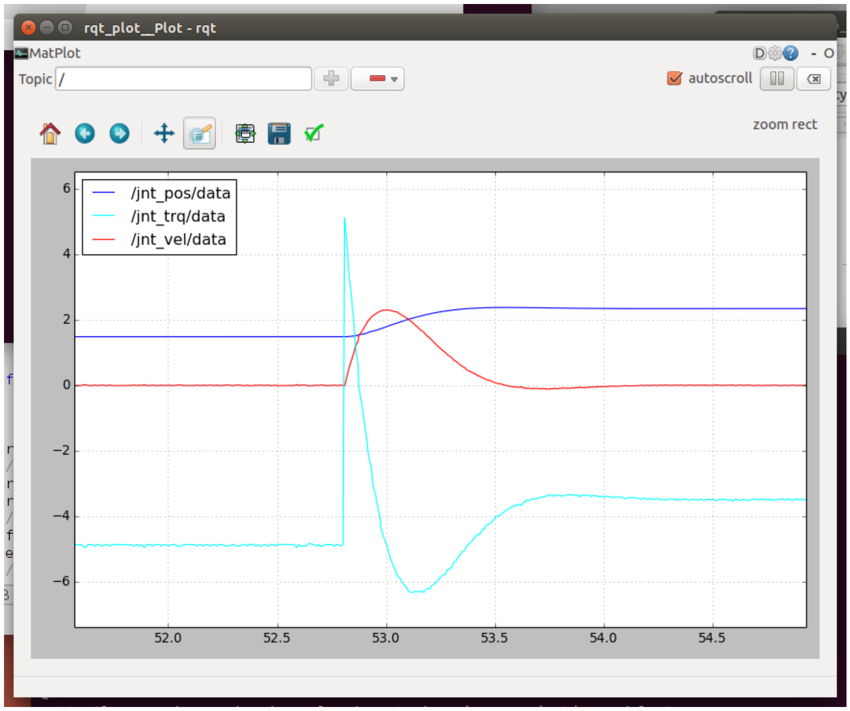


FIGURE 3.12: Gazebo display of minimal robot with minimal controller

in performing simulations in dynamic virtual worlds. As noted, though, simulating joint controllers in Gazebo is better performed using “plug-ins,” as described next.

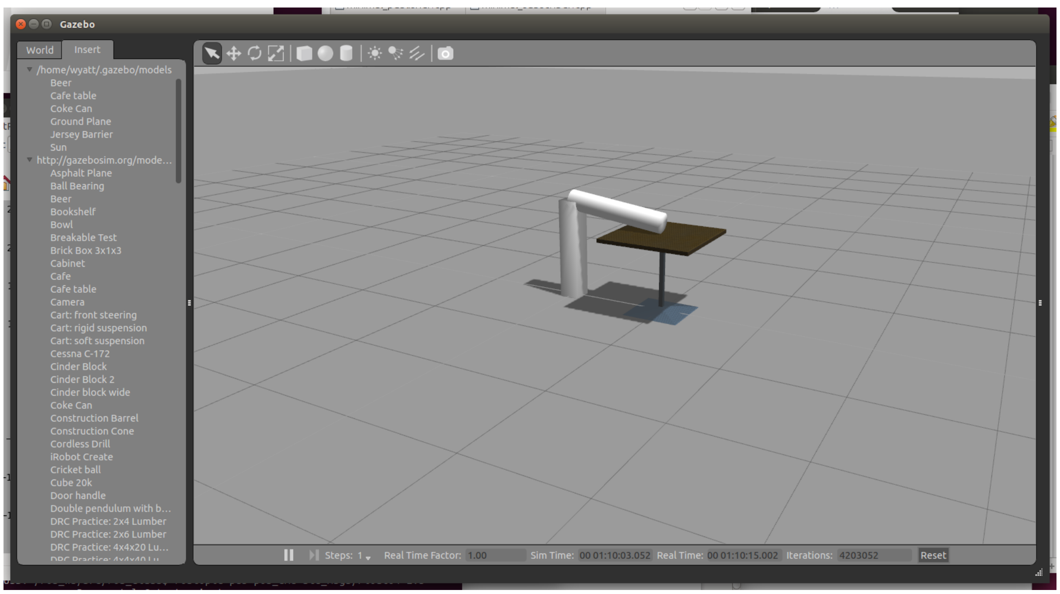


FIGURE 3.13: Gazebo display of minimal robot contacting a rigid object

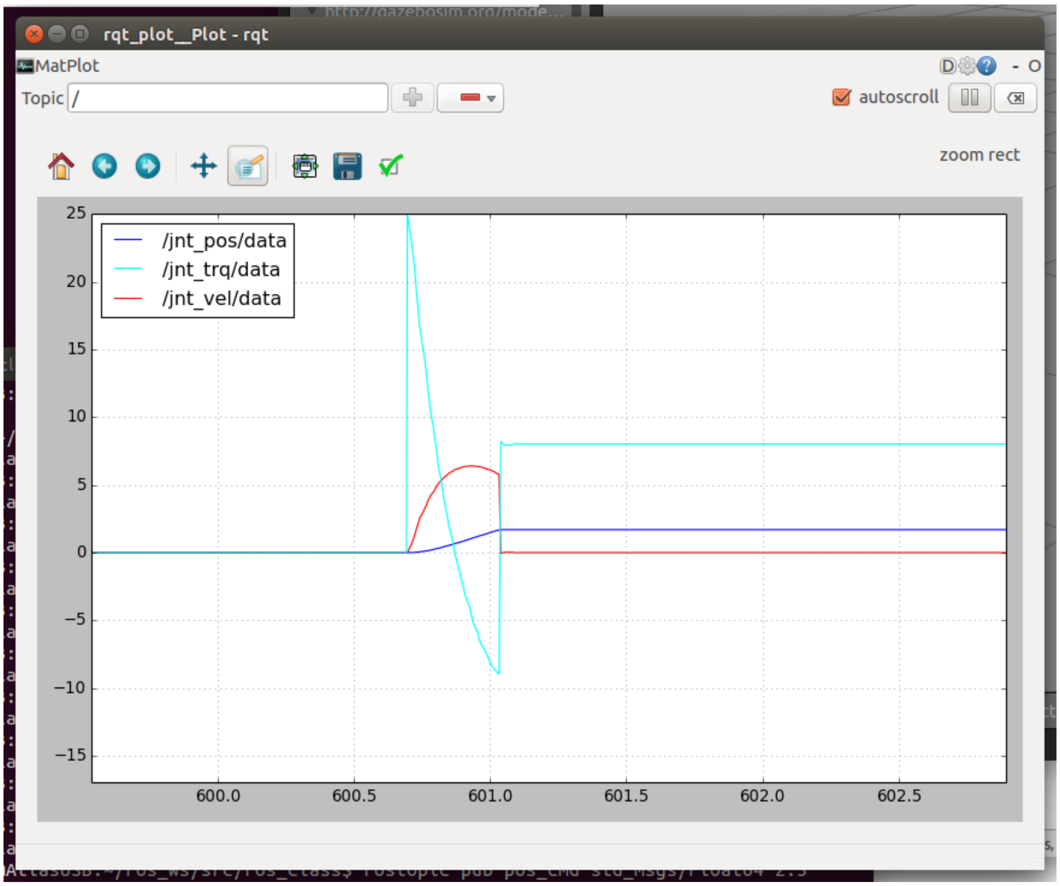


FIGURE 3.14: Contact transient when colliding with table

3.6 USING A GAZEBO PLUG-IN FOR JOINT SERVO CONTROL

A Gazebo plug-in can run at the full rate of the Gazebo simulator (1kHz, by default), without the overhead and associated latency of message passing. ROS controllers in Gazebo are intended to be constructed with interfaces identical to actual hardware–as well as dynamic behavior in simulation that emulates dynamic behavior of actual hardware. Further, the ROS packages and example controllers were constructed with anticipation of growth and generalization. Unfortunately, this has resulted in a fair amount of complexity in creating and simulating models of joint controllers. Further, multiple additional packages are required (including, for example with the "indigo" ROS release: ros-indigo-controller-interface, ros-indigo-gazebo-ros-control, ros-indigo-joint-state-controller, and ros-indigo-effort-controllers), and these packages are not installed automatically, even with the "desktop-full" ROS installation. These can be added with the (admin) command:

```
sudo apt-get install ros-indigo-controller-interface ros-indigo-gazebo-ros-control ros-indigo-joint-state-controller ros-indigo-effort-controllers
```

A good Tutorial example to follow regarding ROS control is the "rrbot" tutorial, which can be found here: `http://gazebosim.org/tutorials/?tut=ros_control`.

The process for incorporating ROS joint controllers as a Gazebo plug-in requires the following steps:

- edit the "joint" fields to specify torque and joint limits in the URDF file
- add a "transmission" field to the URDF, one for each joint to be controlled
- add a "gazebo" block in the URDF to bring in the `libgazebo_ros_control.so` controller plugin library
- create a controller parameter YAML file that declares the controller gains
- modify the launch file put the control parameters on the parameter server and start the controllers running

These steps are illustrated here, revisiting our minimal robot specification. Within the package `minimal_robot_description`, a modified URDF file '`minimal_robot_description_w_jnt_ctl.urdf` contains the additions necessary for ROS joint control. The contents of this file appear in Listing 3.9.

Listing 3.9: Modified minimal robot URDF including ROS control specifications

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

<!-- Used for fixing robot to Gazebo 'base_link' -->
  <link name="world"/>

  <joint name="glue_robot_to_world" type="fixed">
    <parent link="world"/>
    <child link="link1"/>
  </joint>

 <!-- Base Link -->
  <link name="link1">
    <collision>
      <origin xyz="0 0 0.5" rpy="0 0 0"/>
      <geometry>
       <box size="0.2 0.2 0.7"/>
      </geometry>
```

```
    </collision>

    <visual>
      <origin xyz="0 0 0.5" rpy="0 0 0"/>
      <geometry>
        <box size="0.2 0.2 1"/>
      </geometry>
    </visual>

    <inertial>
      <origin xyz="0 0 0.5" rpy="0 0 0"/>
      <mass value="1"/>
      <inertia
        ixx="1.0" ixy="0.0" ixz="0.0"
        iyy="1.0" iyz="0.0"
        izz="1.0"/>
    </inertial>
  </link>

<!-- Moveable Link -->
  <link name="link2">
    <collision>
      <origin xyz="0 0 0.5" rpy="0 0 0"/>
      <geometry>
        <cylinder length="1" radius="0.1"/>
        <!--box size="0.15 0.15 0.8"-->
      </geometry>
    </collision>

    <visual>
      <origin xyz="0 0 0.5" rpy="0 0 0"/>
      <geometry>
        <cylinder length="1" radius="0.1"/>
      </geometry>
    </visual>

    <inertial>
      <origin xyz="0 0 0.5" rpy="0 0 0"/>
      <mass value="1"/>
      <inertia
        ixx="0.1" ixy="0.0" ixz="0.0"
        iyy="0.1" iyz="0.0"
        izz="0.005"/>
    </inertial>
  </link>

  <joint name="joint1" type="revolute">
    <parent link="link1"/>
    <child link="link2"/>
    <origin xyz="0 0 1" rpy="0 0 0"/>
    <axis xyz="0 1 0"/>
    <limit effort="10.0" lower="0.0" upper="2.0" velocity="0.5"/>
    <dynamics damping="1.0"/>
  </joint>

  <transmission name="tran1">
    <type>transmission_interface/SimpleTransmission</type>
    <joint name="joint1">
      <hardwareInterface>EffortJointInterface</hardwareInterface>
    </joint>
    <actuator name="motor1">
      <hardwareInterface>EffortJointInterface</hardwareInterface>
      <mechanicalReduction>1</mechanicalReduction>
    </actuator>
  </transmission>
  <gazebo>
     <plugin name="gazebo_ros_control" filename="libgazebo_ros_control.so">
       <robotNamespace>/one_DOF_robot</robotNamespace>
     </plugin>
  </gazebo>
</robot>
```

Listing 3.9 is largely identical to our previous minimal robot URDF, Listing 3.7, except for three blocks. First, the “joint1” block:

```
<joint name="joint1" type="revolute">
  <parent link="link1"/>
  <child link="link2"/>
  <origin xyz="0 0 1" rpy="0 0 0"/>
  <axis xyz="0 1 0"/>
  <limit effort="10.0" lower="0.0" upper="2.0" velocity="0.5"/>
  <dynamics damping="1.0"/>
</joint>
```

has a new “type” of joint and additional parameters. The joint type “revolute” is more appropriate than “continuous” for a robot arm, since robot joints typically have limited range of motion (in contrast to joints for wheels). Another common joint type is “prismatic”, for joints that extend and retract. (For more detail on joint types and specifications, see `http://wiki.ros.org/urdf/XML/joint`). Joint limit values are expressed in the “limit” tag, including upper and lower joint range of motion limits (constrained to the range 0 to 2.0, for this example). Additionally, one can express actuator dynamic limits, including a velocity limit (here, set to 0.5 rad/s) and a torque limit (set to 10.0 N-m). The expression “effort” is used instead of “torque”, since an “effort” can be either a force or a torque, depending on whether the joint is prismatic or revolute. The joint might also have inherent damping (linear friction). This is included in our example with the expression `<dynamics damping="1.0"/>`, which imposes joint friction of 1.0 N-m/(rad/s).

A second edit to the URDF file is a “transmission” block (see `http://wiki.ros.org/urdf/XML/Transmission`). In the present example, this declares a transmission associated with joint1:

```
<transmission name="tran1">
  <type>transmission_interface/SimpleTransmission</type>
  <joint name="joint1">
    <hardwareInterface>EffortJointInterface</hardwareInterface>
  </joint>
  <actuator name="motor1">
    <hardwareInterface>EffortJointInterface</hardwareInterface>
    <mechanicalReduction>1</mechanicalReduction>
  </actuator>
</transmission>
```

The above block is required, and the joint name must be associated with a corresponding joint name detailed in the URDF–“joint” in this case. (For multiple joints to be controlled, a corresponding transmission block must be inserted for each controlled joint). While future options are anticipated, the lines

```
<type>transmission_interface/SimpleTransmission</type>
```

and

```
<hardwareInterface>EffortJointInterface</hardwareInterface>
```

are, at the time of this writing, the only available “options” for these required elements.

In the example transmission block, the transmission ratio is set to unity. For realistic robot dynamics, transmission ratios on the order of 100 are common, and the reflected inertia of the motor can dominate the influence of the link inertia. At present, incorporating this in the Gazebo model would require that the modeler incorporate an estimated reflected motor inertia augmentation of the corresponding link inertia. Correspondingly, the actuator parameters defining velocity saturation, torque saturation and control gains must be represented consistently. If the default unity transmission ratio is used, then the gains must be expressed in joint-space (transmission output) values, as though the motor were a low-speed, high-torque, direct-drive actuator.

A second block to be inserted in the URDF file for use of Gazebo plug-in controllers follows:

```
<gazebo>
   <plugin name="gazebo_ros_control" filename="libgazebo_ros_control.so">
     <robotNamespace>/one_DOF_robot</robotNamespace>
   </plugin>
</gazebo>
```

This block is an instruction to the Gazebo simulator. It brings in the the plug-in library `libgazebo_ros_control.so`. The line

```
<robotNamespace>/one_DOF_robot</robotNamespace>
```

sets a "namespace" for the controller. Since multiple robots may be present in the simulator, it is useful to separate their interfaces into separate namespaces. The name chosen, `one_DOF_robot`, must be consistent verbatim with naming in two other files: the control-parameter YAML file and the launch file (described below).

A file that must be created for use with Gazebo plug-in controllers is a controller parameter file, in YAML syntax. Such files typically reside within a subdirectory for configuration files. In the the package `minimal_robot_description`, a subdirectory `control_config` was created, which contains the file `one_dof_ctl_params.yaml`. The contents of this file are:

```
one_DOF_robot:
  # Publish all joint states -----------------------------------
  joint_state_controller:
    type: joint_state_controller/JointStateController
    publish_rate: 50

  # Position Controllers ---------------------------------------
  joint1_position_controller:
    type: effort_controllers/JointPositionController
    joint: joint1
    pid: {p: 10.0, i: 10.0, d: 10.0, i_clamp_min: -10.0, i_clamp_max: 10.0}
```

The control parameter file starts with a robot name, in this case: `one_DOF_robot:`. This name must agree with the namespace name in the "gazebo" tag within the URDF file.

The control-parameter file associates a controller name with each controlled joint. In the present case, a controller named `joint1_position_controller` is associated with joint "joint1." For additional controlled joints, this block should be replicated, assigning a unique controller name associated with each controlled joint name in the corresponding URDF file.

The type of controller specified for this example is a PID joint position controller, which is a servo controller with proportional, derivative and integral-error gains. The gain values are specified by the line:

```
pid: {p: 10.0, i: 10.0, d: 10.0, i_clamp_min: -10.0, i_clamp_max: 10.0}
```

which assigns values for the proportional gain (10.0 N-m/rad), the derivative gain (10.0 N-m/(rad/s)) and the integral-error gain (10.0 N-m/(rad-s)). The suggested values are not well tuned, but they are valid.

Finding good gains for joint controllers can be challenging. This can be done interactively with graphical assistance using the command:

```
rosrun rqt_gui rqt_gui
```

Use of this graphical tool for joint-control parameter tuning is described here `http://gazebosim.org/tutorials?tut=ros_control`.

Tuning the integral-error gain can be particularly challenging. This control gain can easily lead to instability. Using a gain of 0 is a good place to start. If this gain is non-zero, then anti-windup constraints should be imposed on the integral-error computation. These are specified by the values of `i_clamp_min` and `i_clamp_max`. If integral-error feedback is not used (i.e., by setting the i: term to zero), then it is not necessary to specify these values.

For the control-parameter file to get associated with the corresponding real-time control code, the YAML file is first loaded onto the parameter server. This is conveniently done within a launch file.

A launch file that performs the necessary start-up functions for our example is `minimal_robot_w_jnt_ctl.launch`, contained within the `minimal_robot_description` package. The contents of this launch file appears in Listing 3.10.

Listing 3.10: Launch file for minimal robot using ROS control plug-ins

```
1  <launch>
2    <!-- Load joint controller configurations from YAML file to parameter server -->
3    <rosparam file="$(find minimal_robot_description)/control_config/one_dof_ctl_params.yaml" command="load"/>
4    <param name="robot_description"
5       textfile="$(find minimal_robot_description)/minimal_robot_description_w_jnt_ctl.urdf"/>
6
7    <!-- Spawn a robot into Gazebo -->
8    <node name="spawn_urdf" pkg="gazebo_ros" type="spawn_model"
9       args="-param robot_description -urdf -model one_DOF_robot" />
10
11   <!--start up the controller plug-ins via the controller manager -->
12   <node name="controller_spawner" pkg="controller_manager" type="spawner" respawn="false"
13     output="screen" ns="/one_DOF_robot" args="joint_state_controller joint1_position_controller"/>
14
15 </launch>
```

Line 3 of Listing 3.10 loads the control parameter file on the parameter server, where it will be accessed when the joint controllers are started. Lines 4-5 loads the (modified) robot model URDF file onto the parameter server. Lines 8-9 load the robot model into the Gazebo simulator. This task was formerly done with a separate terminal command, but this operation is now automated by including it in the launch file. Also, the robot model in this instance is spawned into Gazebo by accessing it from the parameter server instead of from a file. Lines 12-13 use the “spawner” node from the `controller_manager` package to bring in the PID controller(s) and start them up. In this command, the argument `joint1_position_controller` refers to the controller name specified in the control-parameter YAML file. If there are more joint controllers to be used, each controller’s name should be listed in the list of arguments in this command.

Note that the controller launch command specifies `ns="/one_DOF_robot"`. This namespace assignment must be identical to the name assigned in the control-parameter YAML file as well as the namespace specified in the “gazebo” tag of the URDF file.

A roslaunch option used in the controller launch command is: `output="screen"`. With this option specified, printed output from the node being launched will appear within the terminal from which roslaunch is invoked. This is the case even when there are multiple nodes launched from the same launch file. (Formerly, we used `rqt_console` to view such messages when launching `minimal_nodes`, since launching multiple nodes from a single terminal suppressed their `ROS_INFO` displays within that terminal).

With the preceding changes, our controlled robot can be launched as follows. First, bring up Gazebo with an empty world from a terminal with the command:

```
roslaunch gazebo_ros empty_world.launch
```

In a second terminal, launch our robot model, complete with controllers:

```
roslaunch minimal_robot_description minimal_robot_w_jnt_ctl.launch
```

Terminal output from this launch ends with:

```
Loading controller: joint_state_controller
Loading controller: joint1_position_controller
Controller Spawner: Loaded controllers: joint_state_controller,
    joint1_position_controller
Started controllers: joint_state_controller, joint1_position_controller
```

Also, the terminal from which "gazebo" was launched displays:

```
Loading gazebo_ros_control plugin
Starting gazebo_ros_control plugin in namespace: /one_DOF_robot
gazebo_ros_control plugin is waiting for model URDF in
   parameter [/robot_description] on the ROS param server.
Loaded gazebo_ros_control.
```

Our 1-DOF robot appears in the Gazebo graphical display, identical to the initial case presented in Sec 3.4. However, we now have additional topics. Running `rostopic list` shows that the following additional topics:

```
/one_DOF_robot/joint1_position_controller/command
/one_DOF_robot/joint1_position_controller/pid/parameter_descriptions
/one_DOF_robot/joint1_position_controller/pid/parameter_updates
/one_DOF_robot/joint1_position_controller/state
/one_DOF_robot/joint_states
```

These topics all appear under the namespace `one_DOF_robot`. The topic `/one_DOF_robot/joint1_position_contr` is subscribed to by Gazebo, and the message type is `std_msgs/Float64`. This topic is used by the joint position controller as a desired setpoint, equivalent to `pos_command` in our previous example minimal controller. We can use this topic to command our robot manually by entering, e.g.:

```
rostopic pub -r 10 /one_DOF_robot/joint1_position_controller/command std_msgs/Float64 ↩
    1.0
```

which commands the joint to a desired angle of 1.0 radians. The resulting response is shown in Fig 3.15. The resulting response is slow and has overshoot, calling for tuning of the control parameters. With the integral-error term active, however, the link ultimately converges virtually perfectly on the commanded setpoint of 1.0 rad.

With the preceding introductory material, we next consider a slightly more complex robot model in the context of a mobile robot.

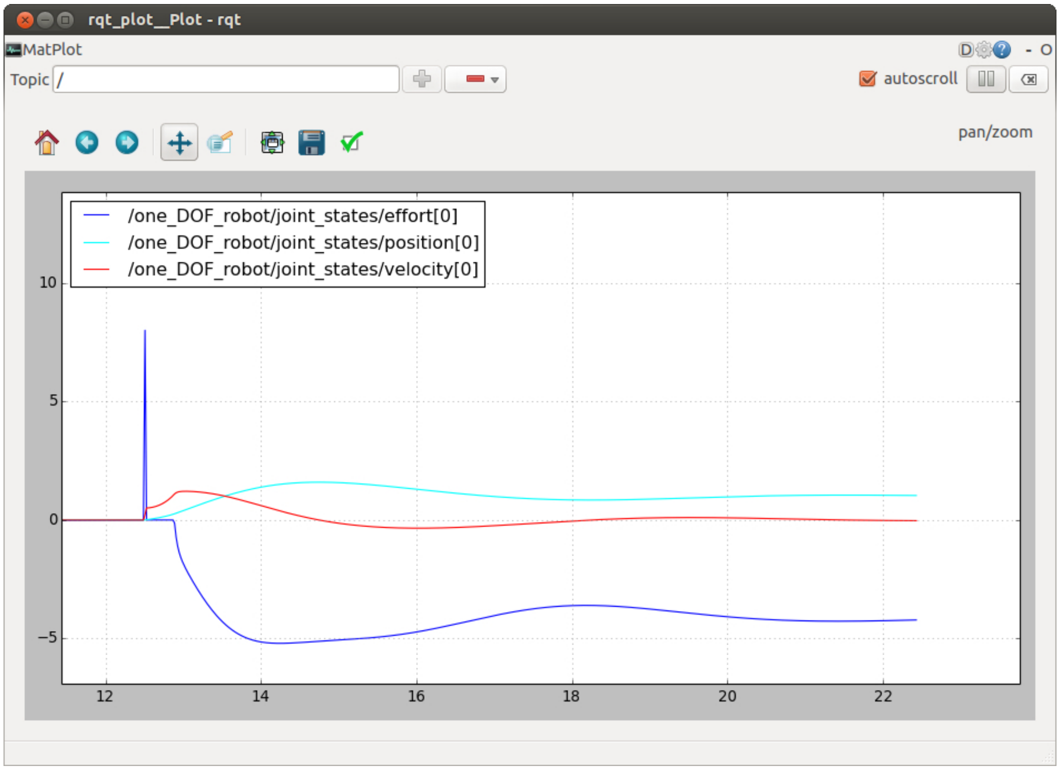


FIGURE 3.15: Transient response to step position command with ROS PID controller

3.7 BUILDING A MOBILE ROBOT MODEL

Extending the URDF modeling introduced in Section 3.3, modelling of a simple mobile robot is presented here, introducing some additional modeling and Gazebo capabilities. On-line tutorials for building mobile robots, including `http://wiki.ros.org/urdf/Tutorials/Building%20a%20Visual%20Robot%20Model%20with%20URDF%20from%20Scratch`. (See `http://gazebosim.org/tutorials?tut=build_robot` for a mobile-robot modeling tutorial using SDF, which is similar, but richer modeling format that is becoming more popular).

Extensions of URDF modeling to be introduced here include the use of "xacro" to help simplify URDF models, and how to include a Gazebo "plug-in," in this case for a differential-drive controller. The model, called "mobot.xacro," is contained in the accompanying repository within the package "`mobot_urdf`." The contents of `mobot.xacro` is given in Listing 3.11, which will be discussed in detail.

Listing 3.11 is long, even though the model is relatively simple. Since URDF files can get very detailed, a technique that helps is to define macros for common operations and to name parameters rather than embed magic numbers throughout the file. Defining and using macros within a URDF is performed with the "xacro" package (see `http://wiki.ros.org/xacro`). A model file that uses xacro macros is saved with the file suffix `.xacro`. Xacro files are converted to URDF files using the "xacro" executable within the "xacro" package. For example, to convert the file `mobot.xacro` to a URDF with filename `mobot.urdf`, run the following command:

```
rosrun xacro xacro mobot.xacro > mobot.urdf
```

This action creates a URDF file with all of the substitutions defined by the xacro macros. The URDF file produced can be checked for consistency with the command:

```
check_urdf mobot.urdf
```

which produces the output:

```
robot name is: mobot
---------- Successfully Parsed XML ---------------
root Link: base_link has 5 child(ren)
    child(1):  batterybox
    child(2):  castdrop_left
        child(1):  brackettop_left
            child(1):  bracketside1_left
                child(1):  left_casterwheel
            child(2):  bracketside2_left
    child(3):  castdrop_right
        child(1):  brackettop_right
            child(1):  bracketside1_right
                child(1):  right_casterwheel
            child(2):  bracketside2_right
    child(4):  left_wheel
    child(5):  right_wheel
```

This output shows that the URDF file can be parsed logically, and it displays that there are 14 links in the model. Relationships among the links are shown in an outline style. The tree of links also can be visualized graphically using the command:

```
urdf_to_graphiz mobot.urdf
```

which produces the file `mobot_graphiz.pdf`, which is displayed in Fig 3.16.

Figure 3.16 illustrates the connectivities and spatial relationships among the links.

The mobot model can be loaded into Gazebo to visualize and simulate. As before, this is done by first loading the robot-description file into the parameter server, then executing the `spawn_model` node within the `gazebo_ros` package, referencing the robot model on the parameter server. This is accomplished via the launch file `mobot.launch` in the `urdf` subdirectory of the package `mobot_urdf`. First, Gazebo is started with the command:

```
roslaunch gazebo_ros empty_world.launch
```

Then the mobot model is inserted into the Gazebo simulation by running the corresponding launch file:

```
roslaunch mobot_urdf mobot.launch
```

The resulting Gazebo display is shown in Fig 3.17. In Fig 3.17, the Gazebo option `view->joints` has been enabled. It can be seen that there are 6 movable joints: 2 joints corresponding to the 2 large drive wheels, and 4 joints associated with the passive casters.

An interpretation of the Listing 3.11 follows. Line 3:

```
xmlns:xacro="http://www.ros.org/wiki/xacro" name="mobot">
```

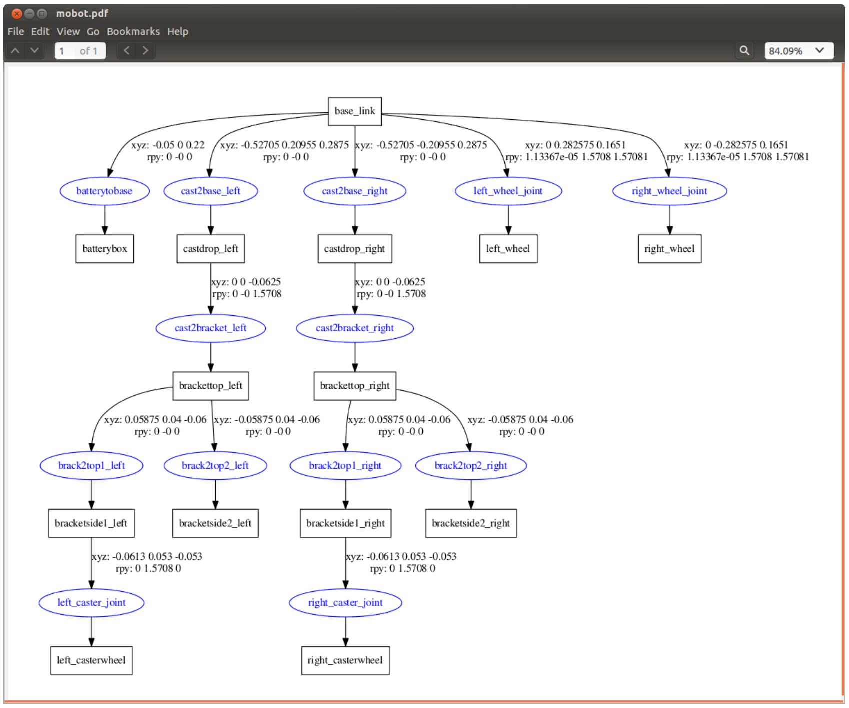


FIGURE 3.16: Graphical display of URDF tree for mobot URDF

---

establishes that this file will use the “xacro” package for defining/using macros.

Lines 6-43 use xacro properties to define mnemonic names to represent numerical values. This technique allows for defining URFD elements with symbolic parameters. Use of xacro properties helps to keep numerical values consistent throughout the file (importantly, if these values are used in multiple instances). It also makes the file easier to modify to change dimensions to tune a model to a physical system. Such assignments can be used as in line 79:

```
<origin xyz="0 ${reflect*track/2} ${tirerad}" rpy="0 ${M_PI/2} ${M_PI/2}"/>
```

which uses symbols to define the origin of a wheel frame.

In addition to defining parameters, xacro allows for defining macros. For example, lines 47-54 define the macro `default_inertia`:

```
    <xacro:macro name="default_inertial" params="mass">
        <inertial>
            <mass value="${mass}" />
            <inertia ixx="0.01" ixy="0.0" ixz="0.0"
         iyy="0.01" iyz="0.0"
         izz="0.01" />
        </inertial>
    </xacro:macro>
```

This macro is used 6 times in the model listing, e.g. as in line 98:

```
<origin xyz="0 ${reflect*track/2} ${tirerad}" rpy="0 ${M_PI/2} ${M_PI/2}"/>
```

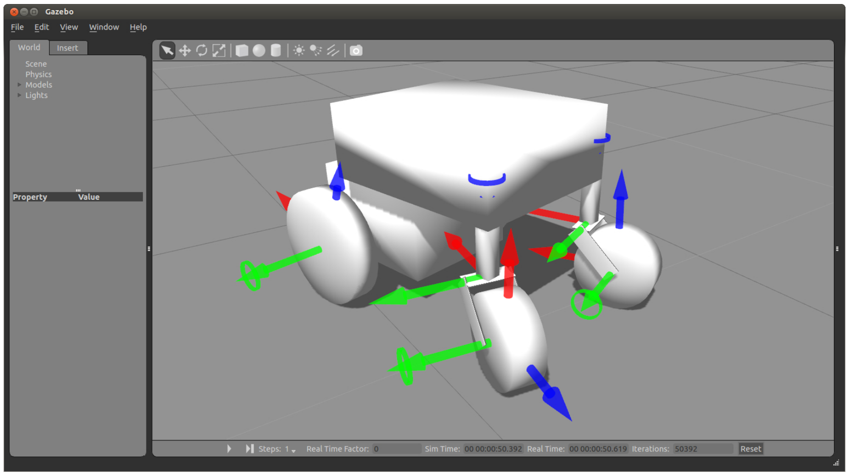


FIGURE 3.17: Gazebo view of mobot in empty world

In fact, the use of this macro in line 98 is embedded within another macro definition, the “caster” macro, lines 84-189. The first two lines of this macro are:

```
<xacro:macro name="caster" params="prefix reflect">
<link name="castdrop_${prefix}">
```

The `caster` macro is fairly detailed. It is used twice in the code to define symmetric left and right caster elements through use of the “prefix” parameter. This allows defining links with names `castdrop_left` and `castdrop_right`, which are declared in the model on lines 191-192:

```
    <xacro:caster prefix="left" reflect="1"/>
    <xacro:caster prefix="right" reflect="-1"/>
```

Through use of this macro, it is assured that the left and right casters will be identical, except for their placement within the model. Changes to parameter values used within this macro will still result in left and right casters that are identical.

This same macro approach is used to define the drive wheels on lines 55-83, starting with the lines:

```
<xacro:macro name="wheel" params="prefix reflect">
        <link name="${prefix}_wheel">
```

The “wheel” macro is invoked twice, on lines 217-218:

```
    <xacro:wheel prefix="left" reflect="1"/>
    <xacro:wheel prefix="right" reflect="-1"/>
```

Again, the macro assures that the left and right drive wheels will be identical, even when dimensional changes are made to the parameters.

The “box” and “cylinder” geometric objects are used to define the 14 links in this system, both for their visual representation and for their collision boundaries. Inertial components are defined for each link in the system. All fields within the link and joint definitions are of the same style as introduced in the `minimal_robot` URDF, except for the use of named parameters in place of numerical values.

As with the minimal robot arm example, it is useful to bring in a Gazebo plug-in for joint control. For the "mobot" model, this is done in lines 239-255:

```
<gazebo>
  <plugin name="differential_drive_controller" filename="libgazebo_ros_diff_drive.so">
    <alwaysOn>true</alwaysOn>
    <updateRate>100</updateRate>
    <leftJoint>right_wheel_joint</leftJoint>
    <rightJoint>left_wheel_joint</rightJoint>
    <wheelSeparation>${track}</wheelSeparation>
    <wheelDiameter>${tirediam}</wheelDiameter>
    <torque>200</torque>
    <commandTopic>cmd_vel</commandTopic>
    <odometryTopic>odom</odometryTopic>
    <odometryFrame>odom</odometryFrame>
    <robotBaseFrame>base_link</robotBaseFrame>
    <publishWheelTF>true</publishWheelTF>
    <publishWheelJointState>true</publishWheelJointState>
  </plugin>
 </gazebo>
```

The XML code between the `<gazebo>` tags brings in the library `libgazebo_ros_diff_drive.so`, which is useful for differential-drive control similar to that of the previous Simple Two-Dimensional Robot simulator. To use the differential-drive plug-in, several parameters must be defined, including

- the names of the drive-wheel joints, `right_wheel_joint` and `left_wheel_joint`
- the wheel separation, `track`
- the name of the root of the URDF tree, which is `base_link`
- the topic name to be used to command speed/spin, set (per convention) to `cmd_vel`

Further details can be found at `http://gazebosim.org/tutorials?tut=ros_gzplugins` and `http://www.theconstructsim.com/?p=3332`.

Listing 3.11: A simple mobile robot xacro model file

```
1  <?xml version="1.0"?>
2  <robot
3       xmlns:xacro="http://www.ros.org/wiki/xacro" name="mobot">
4       <!-- define the base-link origin to lie at floor level, between the drive wheels -->
5       <!--main body is a simple box; origin is a center of box-->
6      <xacro:property name="bodylen" value="0.5461" />
7      <xacro:property name="bodywidth" value="0.4572" />
8      <xacro:property name="bodyheight" value="0.2" />
9      <xacro:property name="bodyclearance" value="0.4" />  <!--clearance from bottom of box to ground-->
10     <!-- derived values-->
11     <xacro:property name="half_bodylen" value="${bodylen/2.0}" />
12     <xacro:property name="half_bodyheight" value="${bodyheight/2.0}" />
13     <!-- placement of main body relative to base link frame -->
14     <xacro:property name="bodyOX" value="${-half_bodylen}" />
15     <xacro:property name="bodyOY" value="0" />
16     <xacro:property name="bodyOZ" value="0.45" />
17
18     <!-- define the drive-wheel dimensions-->
19     <xacro:property name="tirediam" value="0.3302" />
20     <xacro:property name="tirerad" value="${tirediam/2.0}" />
21     <xacro:property name="tirewidth" value="0.06985" />
22     <!-- "track" is the distance between the drive wheels -->
```

```
    <xacro:property name="track" value=".56515" />

    <!-- battery box dimensions -->
    <xacro:property name="batterylen" value="0.381" />
    <xacro:property name="batterywidth" value="0.3556" />
    <xacro:property name="batteryheight" value="0.254" />
    <!-- placement of battery box relative to base frame -->
    <xacro:property name="batOX" value="-0.05" />
    <xacro:property name="batOY" value="0" />
    <xacro:property name="batOZ" value="0.22" />


    <xacro:property name="M_PI" value="3.1415926535897931" />
    <xacro:property name="boschwidth" value="0.0381" />
    <xacro:property name="casterdrop" value="0.125" />
    <xacro:property name="bracketwidth" value="0.1175" />
    <xacro:property name="bracketheight" value="0.16" />
    <xacro:property name="bracketthick" value="0.0508" />
    <xacro:property name="bracketangle" value="0.7854" />
    <xacro:property name="casterwidth" value="0.0826" />
    <xacro:property name="casterdiam" value="0.2286" />

    <!--here is a default inertia matrix with small, but legal values; use this when don't need accuracy for I -->
    <!--model will assign inertia matrix dominated by main body box -->
    <xacro:macro name="default_inertial" params="mass">
        <inertial>
            <mass value="${mass}" />
            <inertia ixx="0.01" ixy="0.0" ixz="0.0"
         iyy="0.01" iyz="0.0"
         izz="0.01" />
        </inertial>
    </xacro:macro>
    <xacro:macro name="wheel" params="prefix reflect">
        <link name="${prefix}_wheel">
            <visual>
                <geometry>
                    <cylinder radius="${tirerad}" length="${tirewidth}"/>
                </geometry>
            </visual>
            <collision>
                <geometry>
                    <cylinder radius="${tirerad}" length="${tirewidth}"/>
                </geometry>
            </collision>
            <inertial>
            <!--assign inertial properties to drive wheels -->
            <mass value="1" />
            <inertia ixx="0.1" ixy="0" ixz="0"
              iyy="0.1" iyz="0"
              izz="0.1" />
        </inertial>
        </link>
        <joint name="${prefix}_wheel_joint" type="continuous">
            <axis xyz="0 0 1"/>
            <parent link="base_link"/>
            <child link="${prefix}_wheel"/>
            <origin xyz="0 ${reflect*track/2} ${tirerad}" rpy="0 ${M_PI/2} ${M_PI/2}"/>
            <limit effort="100" velocity="15" />
            <joint_properties damping="0.0" friction="0.0" />
        </joint>
    </xacro:macro>
    <xacro:macro name="caster" params="prefix reflect">
        <link name="castdrop_${prefix}">
            <visual>
                <geometry>
                    <box size="${boschwidth} ${boschwidth} ${casterdrop}"/>
                </geometry>
                <origin xyz="0 0 0" rpy="0 0 0"/>
            </visual>
            <collision>
                <geometry>
                    <box size="${boschwidth} ${boschwidth} ${casterdrop}"/>
                </geometry>
                <origin xyz="0 0 0" rpy="0 0 0"/>
```

```
            </collision>
            <xacro:default_inertial mass="0.2"/>
        </link>
        <joint name="cast2base_${prefix}" type="fixed">
            <parent link="base_link"/>
            <child link="castdrop_${prefix}"/>
            <origin xyz="${-bodylen/2+bodyOX+boschwidth/2} ${reflect*bodywidth/2-reflect*boschwidth/2} ${-casterdrop/2-bodyheight/2+bodyOZ}" />
        </joint>
        <link name="brackettop_${prefix}">
            <visual>
                <geometry>
                    <box size="${bracketwidth} ${bracketthick} .005"/>
                </geometry>
                <origin xyz="0 0 0" rpy="0 0 0"/>
            </visual>
            <collision>
                <geometry>
                    <box size="${bracketwidth} ${bracketthick} .005"/>
                </geometry>
                <origin xyz="0 0 0" rpy="0 0 0"/>
            </collision>
            <xacro:default_inertial mass="0.2"/>
        </link>
        <joint name="cast2bracket_${prefix}" type="continuous">
            <axis xyz="0 0 1"/>
            <parent link="castdrop_${prefix}"/>
            <child link="brackettop_${prefix}"/>
            <origin xyz="0 0 ${-casterdrop/2}" rpy="0 0 ${M_PI/2}"/>
            <joint_properties damping="0.0" friction="0.0" />
        </joint>
        <link name="bracketside1_${prefix}">
            <visual>
                <geometry>
                    <box size="${bracketthick} ${bracketheight} .005"/>
                </geometry>
                <origin xyz="0 0 0" rpy="${M_PI/2} ${-bracketangle} ${M_PI/2}"/>
            </visual>
            <collision>
                <geometry>
                    <box size="${bracketthick} ${bracketheight} .005"/>
                </geometry>
                <origin xyz="0 0 0" rpy="${M_PI/2} ${-bracketangle} ${M_PI/2}"/>
            </collision>
            <xacro:default_inertial mass="0.2"/>
        </link>
        <joint name="brack2top1_${prefix}" type="fixed">
            <parent link="brackettop_${prefix}"/>
            <child link="bracketside1_${prefix}"/>
            <origin xyz="${bracketwidth/2} .04 -${bracketheight/2-.02}" rpy="0 0 0" />
        </joint>
        <link name="bracketside2_${prefix}">
            <visual>
                <geometry>
                    <box size="${bracketthick} ${bracketheight} .005"/>
                </geometry>
                <origin xyz="0 0 0" rpy="${M_PI/2} ${-bracketangle} ${M_PI/2}"/>
            </visual>
            <collision>
                <geometry>
                    <box size="${bracketthick} ${bracketheight} .005"/>
                </geometry>
                <origin xyz="0 0 0" rpy="${M_PI/2} ${-bracketangle} ${M_PI/2}"/>
            </collision>
            <xacro:default_inertial mass="0.2"/>
        </link>
        <joint name="brack2top2_${prefix}" type="fixed">
            <parent link="brackettop_${prefix}"/>
            <child link="bracketside2_${prefix}"/>
            <origin xyz="${-bracketwidth/2} .04 -${bracketheight/2-.02}" rpy="0 0 0" />
        </joint>
        <link name="${prefix}_casterwheel">
            <visual>
                <geometry>
                    <cylinder radius="${casterdiam/2}" length="${casterwidth}"/>
```

```
               </geometry>
           </visual>
           <collision>
               <geometry>
                   <cylinder radius="${casterdiam/2}" length="${casterwidth}"/>
               </geometry>
           </collision>
           <!-- accept default inertial properties for caster wheels-->
           <xacro:default_inertial mass="0.5"/>
       </link>
       <joint name="${prefix}_caster_joint" type="continuous">
           <axis xyz="0 0 1"/>
           <parent link="bracketside1_${prefix}"/>
           <child link="${prefix}_casterwheel"/>
           <origin xyz="${-casterwidth/2-.02} .053 -.053" rpy="0 ${M_PI/2} 0"/>
           <limit effort="100" velocity="15" />
           <joint_properties damping="0.0" friction="0.0" />
       </joint>
   </xacro:macro>

   <xacro:caster prefix="left" reflect="1"/>
   <xacro:caster prefix="right" reflect="-1"/>

   <link name="base_link">
       <visual>
           <geometry>
               <box size="${bodylen} ${bodywidth} ${bodyheight}"/>
           </geometry>
           <origin xyz="${bodyOX} ${bodyOY} ${bodyOZ}" rpy="0 0 0"/>
       </visual>
       <collision>
           <geometry>
               <box size="${bodylen} ${bodywidth} ${bodyheight}"/>
           </geometry>
           <origin xyz="${bodyOX} ${bodyOY} ${bodyOZ}" rpy="0 0 0"/>
       </collision>
       <inertial>
           <!--assign almost all the mass to the main body box; set m= 100kg; treat I as approx m*r^2 -->
           <mass value="100" />
           <inertia ixx="10" ixy="0" ixz="0"
             iyy="10" iyz="0"
             izz="10" />
       </inertial>
   </link>


   <xacro:wheel prefix="left" reflect="1"/>
   <xacro:wheel prefix="right" reflect="-1"/>
   <link name="batterybox">
       <visual>
           <geometry>
               <box size="${batterylen} ${batterywidth} ${batteryheight}"/>
           </geometry>
           <origin xyz="0 0 0" rpy="0 0 0"/>
       </visual>
       <collision>
           <geometry>
               <box size="${batterylen} ${batterywidth} ${batteryheight}"/>
           </geometry>
           <origin xyz="0 0 0" rpy="0 0 0"/>
       </collision>
       <xacro:default_inertial mass="1"/>
   </link>
   <joint name="batterytobase" type="fixed">
       <parent link="base_link"/>
       <child link="batterybox"/>
       <origin xyz="${batOX} ${batOY} ${batOZ}" rpy="0 0 0"/>
   </joint>
<gazebo>
 <plugin name="differential_drive_controller" filename="libgazebo_ros_diff_drive.so">
   <alwaysOn>true</alwaysOn>
   <updateRate>100</updateRate>
   <leftJoint>right_wheel_joint</leftJoint>
   <rightJoint>left_wheel_joint</rightJoint>
   <wheelSeparation>${track}</wheelSeparation>
```

```
    <wheelDiameter>${tirediam}</wheelDiameter>
    <torque>200</torque>
    <commandTopic>cmd_vel</commandTopic>
    <odometryTopic>odom</odometryTopic>
    <odometryFrame>odom</odometryFrame>
    <robotBaseFrame>base_link</robotBaseFrame>
    <publishWheelTF>true</publishWheelTF>
    <publishWheelJointState>true</publishWheelJointState>
  </plugin>
 </gazebo>

</robot>
```

3.8 SIMULATING THE MOBILE ROBOT MODEL

As noted earlier, the robot model can be introduced in the Gazebo simulator by

```
roslaunch gazebo_ros empty_world.launch
```

after which the mobot model is inserted into the Gazebo simulation by running the corresponding launch file:

```
roslaunch mobot_urdf mobot.launch
```

At this point, the mobile robot can be controlled via the `cmd_vel` topic, identical to what was done with the simple two-dimensional robot simulator. For example, the command:

```
rostopic pub cmd_vel geometry_msgs/Twist  '{linear:  {x: 0.5, y: 0.0, z: 0.0}, angular↩
    : {x: 0.0,y: 0.0,z: 0.3}}'
```

causes the robot to move in a counter-clockwise circle.

An important difference relative to the STDR simulator is that the Gazebo simulation includes physics. Inertias, friction, controller dynamics and actuator saturation are taken into account. This can be seen by re-using the `stdr_open_loop_commander` node from the package `stdr_ctl` introduced in Sec 3.1. Note, though, that the `stdr_open_loop_commander.cpp` code publishes to topic file publishes to the topic `/robot0/cmd_vel`, whereas our differential-drive controller of the mobot expects commands on topic `cmd_vel`. We can re-use the `stdr_open_loop_commander` node nonetheless if we “remap” the output topic name. Roslaunch has the capability to remap topics, as illustrated in the launch file `open_loop_squarewave_commander.launch` within the “launch” subdirectory of package `mobot_urdf`.

```
<launch>
<!-- original node publishes to /robot0/cmd_vel; direct this instead to topic /cmd_vel↩
     -->
<node pkg="stdr_control" type="stdr_open_loop_commander" name="commander">
 <remap from="/robot0/cmd_vel" to="cmd_vel" />
</node>
</launch>
```

Invoking this launch file with:

```
roslaunch mobot_urdf open_loop_squarewave_commander.launch
```

the resulting response can be plotted with `rqt_plot`. The mobot simulation in Gazebo publishes its state to the “odom” topic, from which we can plot the robot forward velocity

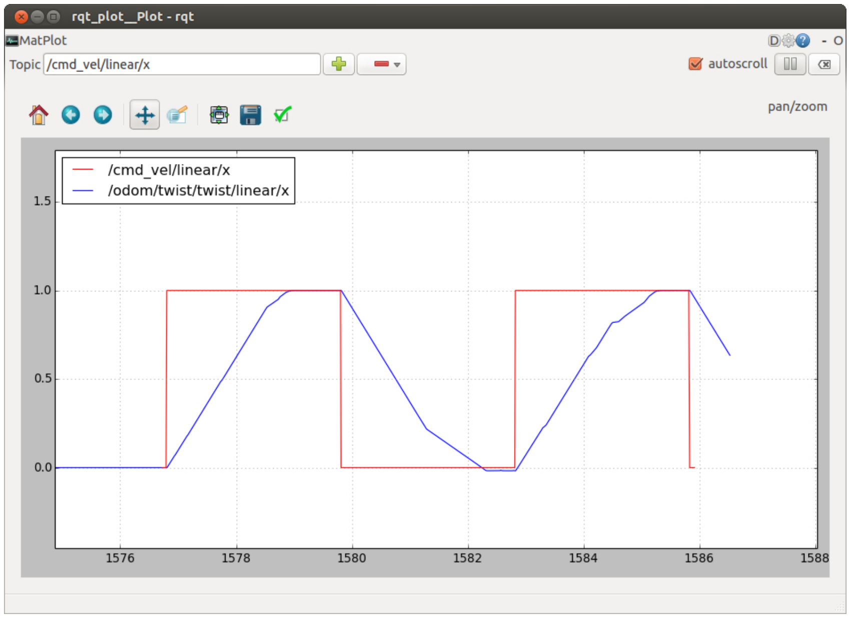


FIGURE 3.18: Response of mobot to step velocity commands in Gazebo simulation

in response to step velocity commands. As seen in Fig 3.18, the robot does not change velocities instantaneously.

One of the idiosyncracies of the ODE physics engine is that it does not handle well modeling sustained stiff contact between separate models. This is noticeable with wheels or feet on the ground, as well as gripper fingers grasping objects. This can be observed with the simple mobile robot model. After starting up Gazebo and inserting the mobot model, the robot will slowly slip to its left. Actually, the robot wheels are "chattering" against the ground. This can be seen in the Gazebo viewer by enabling `view->contacts` from the top menu bar. An example screenshot with contacts displayed appears in Fig 3.19. In Fig 3.19, the blue markers are points of contact and the green lines show the direction and (by length) the magnitude of the contact forces. At this instant, the contact forces on wheels are strong on its left side than on its right. However, if the display is viewed dynamically, the contact forces can be seen to toggle on/off and to shift around due to numerical instability of modelling stiff contact. This problem is exacerbated by the fact that the robot has 4 wheels, and simultaneous contact of four points with a flat plane is numerically challenging. As a result, the robot seems to constantly vibrate on the ground.

A few adjustments can be made to try to improve the numerical stability. Within the URDF (or xacro file), insert the lines:

```
<gazebo reference="left_wheel">
  <mu1>100000.0</mu1>
  <mu2>100000.0</mu2>
  <material>Gazebo/Black</material>
</gazebo>
```

and repeat the block for `right_wheel` as well. This gazebo tag passes parameters to the physics engine, declaring a custom friction property. In fact, the "mu" coefficients are not realistically physical. They do have an influence on the simulation, but these values cannot be treated like genuine dimensionless Coulomb friction components. Some experimentation

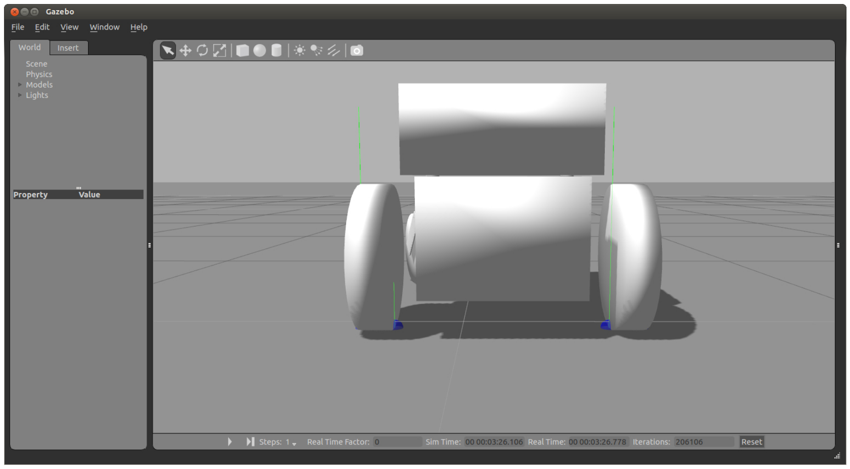


FIGURE 3.19: Gazebo simulation of mobot with contacts display enabled

may be required to get acceptable simulation results for sustained contact simulations, including wheels, feet and fingers.

The line `<material>Gazebo/Black</material>` was also introduced in the above to show how to set colors for links in Gazebo. This property must also be included within `<gazebo>` delimiters. The result of the example insertion (repeated for left and right wheels) is that the model shows up with black wheels.

Another variation that can be introduced is to tell the simulator to run more iterations of internal computations each time step (where the time step is, by default, 1ms). This can result in better fidelity of simulation, including less chatter with sustained, stiff contacts. But this is done at the price of more demanding computation, potentially resulting in a lower real-time factor. The "iterations" parameter can be changed with a gazebo tag in the URDF, or it can be done interactively from the Gazebo window. For the latter, on the left pane, choose the "World" tab, then under `physics->solver->iterations`, edit the numerical value from its default value (50) to something larger (e.g. 200). This change can reduce the side-slip of the mobot model to be imperceptibly slow.

The mobile-robot simulation can be made more interesting by introducing elements in the world model. This can be done by manually inserting existing models or by inserting custom-designed models. An interesting world model that can be inserted from the Gazebo "insert" tab is `Starting Pen`. After selecting this model from the "Insert" tab, the user has the opportunity to move the corresponding model around in the simulated world to a desired location (via the mouse), then click to complete the model placement. Figure 3.20 shows the mobot model within a virtual world that includes the Starting Pen model.

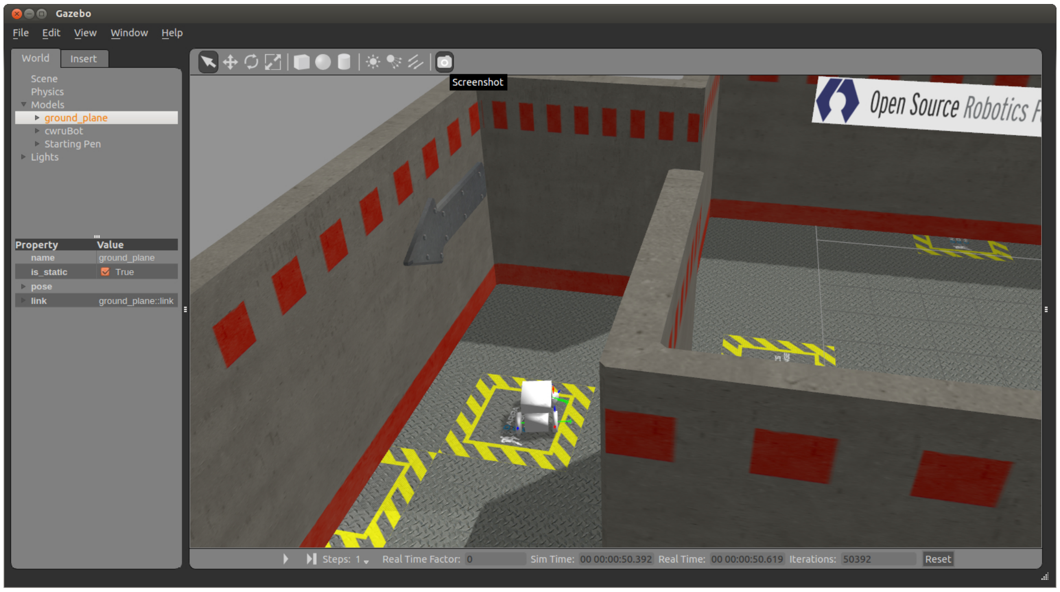


FIGURE 3.20: Gazebo display of a mobot in Starting Pen

3.9 COMBINING ROBOT MODELS

Since robot URDF file can get quite long, it is useful to be able to include sub-systems within a single model file. Similarly, it is convenient to include launch files within launch files.

To illustrate, we will mount our minimal robot arm model onto our simple mobile base. Revisiting the minimal arm model in package `minimal_robot_description`, we make a simple change to the URDF to create the file `minimal_robot_description_unglued.urdf`. Importantly, we delete the joint `glue_robot_to_world`, so our arm can be attached to the mobile base instead. To make the arm a bit more attractive, the following material tags are added to colorize the links:

```
<gazebo reference="link1">
  <material>Gazebo/Blue</material>
</gazebo>
<gazebo reference="link2">
  <material>Gazebo/Red</material>
</gazebo>
```

We can combine the mobile-platform and minimal arm models as illustrated by the file `mobot_w_arm.xacro`, the contents of which are shown in Listing 3.12.

Listing 3.12: Model file combining mobot and minimal arm

```
<?xml version="1.0"?>
<robot
     xmlns:xacro="http://www.ros.org/wiki/xacro" name="mobot">
  <xacro:include filename="$(find mobot_urdf)/urdf/mobot2.xacro" />
  <xacro:include filename="$(find minimal_robot_description)/↩
      minimal_robot_description_unglued.urdf" />

  <!-- attach the simple arm to the mobile robot -->
```

```
8    <joint name="arm_base_joint" type="fixed">
9      <parent link="base_link" />
10     <child link="link1" />
11     <origin rpy="0 0 0 " xyz="${-bodylen/2} 0 ${bodyOZ+bodyheight/2}"/>
12   </joint>
13 </robot>
```

The xacro file of Listing 3.12 resides in the `mobot_urdf` package. Lines 4-5:

```
  <xacro:include filename="$(find mobot_urdf)/urdf/mobot2.xacro" />
  <xacro:include filename="$(find minimal_robot_description)/↩
      minimal_robot_description_unglued.urdf" />
```

use the "include" feature of xacro to bring in two files verbatim. The named file paths start with, e.g., `$(find mobot_urdf)`. This syntax informs the launcher to search for the named files by stated package name (`mobot_urdf`, in this example). Specifying directories in this manner simplifies specifying the search path, and it also makes the launcher more robust. Packages that are installed in a different system under different directories can still be found using this syntax.

After bringing in both the mobile-base and the simple arm models, the arm is attached to the base by declaring a new (fixed) joint, `arm_base_joint`:

```
  <joint name="arm_base_joint" type="fixed">
    <parent link="base_link" />
    <child link="link1" />
    <origin rpy="0 0 0 " xyz="${-bodylen/2} 0 ${bodyOZ+bodyheight/2}"/>
  </joint>
```

This new joint specifies how `link1` (the first link of the arm model) is to be attached to its parent, `base_link` (the base link of the mobile-platform model). The arm model is offset in y and z, using xacro parameters, to center the base of link1 on the top of the main link of the mobile robot.

In addition to combining URDF (or xacro) models, it is also convenient to combine launch files. Our launch file for the robot arm, `minimal_robot_w_jnt_ctl.launch` performed three functions: it loaded the arm URDF onto the parameter server, it spawned the robot model from the parameter server into Gazebo, and it started up the ROS joint position controller(s). Since our integrated mobile base with arm is a new model, we wish to separate the arm-specific launch commands from the model loading and spawning. For this purpose, a smaller launch file is created in the `minimal_robot_description` package, called `minimal_robot_ctl.launch`. The contents of this file appears in Listing 3.13.

Listing 3.13: Launch file for starting arm controller

```
<launch>
  <!-- Load joint controller configurations from YAML file to parameter server -->
  <rosparam file="$(find minimal_robot_description)/control_config/one_dof_ctl_params.↩
      yaml" command="load"/>

  <!--start up the controller plug-ins via the controller manager -->
  <node name="controller_spawner" pkg="controller_manager" type="spawner" respawn="↩
      false"
    output="screen" ns="/one_DOF_robot" args="joint_state_controller ↩
        joint1_position_controller"/>
</launch>
```

A launch file for the integrated robot model resides in the package `mobot_urdf` within the `launch` sub-directory, named `mobot_w_arm.launch`. The contents of this launch file appears in Listing 3.14.

Listing 3.14: Launch file combining mobot and minimal arm

```
<launch>
<!-- Convert xacro model file and put on parameter server -->
<param name="robot_description" command="$(find xacro)/xacro.py '$(find mobot_urdf)/urdf/mobot_w_arm.xacro'" />

<!-- Spawn the robot from parameter server into Gazebo -->
<node name="spawn_urdf" pkg="gazebo_ros" type="spawn_model" args="-param robot_description -urdf -model mobot" />

<!-- load the controller parameter yaml file and start the ROS controllers for the arm -->
<include file="$(find minimal_robot_description)/minimal_robot_ctl.launch">
</include>

</launch>
```

This launch file loads the combined xacro file onto the parameter server then spawns this model into Gazebo. The line:

```
<include file="$(find minimal_robot_description)/minimal_robot_ctl.launch">
```

searches for the file `minimal_robot_ctl.launch` in package `minimal_robot_description` and includes this launch file verbatim.

After launching Gazebo, the integrated launch file can be invoked with:

```
roslaunch mobot_urdf mobot_w_arm.launch
```

The resulting Gazebo display appears as shown in Fig 3.21.

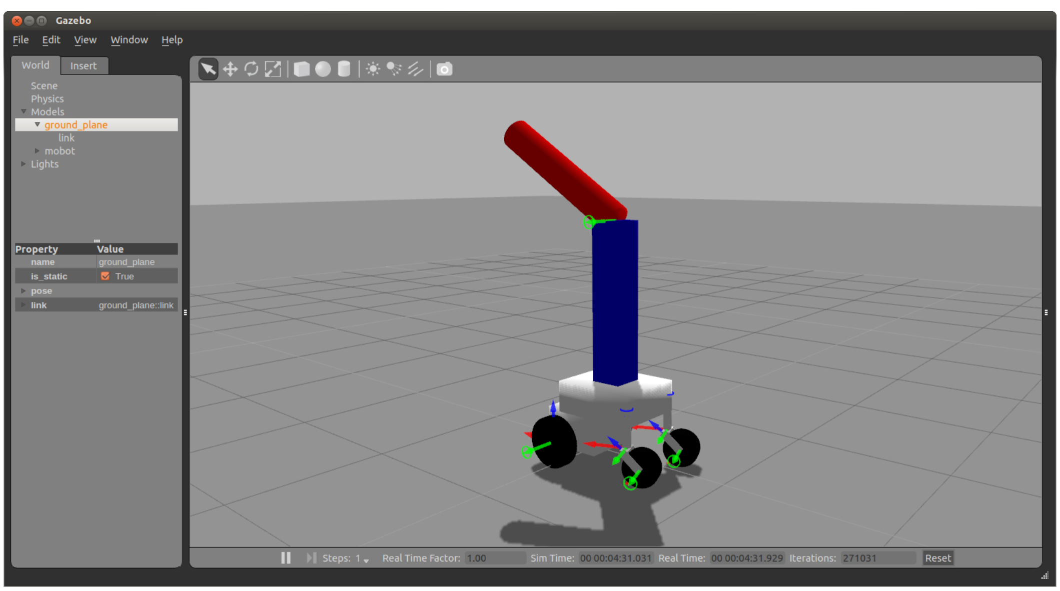


FIGURE 3.21: Gazebo display of combined mobile base and minimal arm models

Running `rostopic list` shows that the following topics are active:

```
/joint_states
/odom
```

```
/cmd_vel
/one_DOF_robot/joint1_position_controller/command
/one_DOF_robot/joint1_position_controller/pid/parameter_descriptions
/one_DOF_robot/joint1_position_controller/pid/parameter_updates
/one_DOF_robot/joint1_position_controller/state
/one_DOF_robot/joint_states
```

We thus see that command-interface topics are available for both the mobile platform and the arm. Feedback values of joint states for both the arm and mobile base are available as well.

3.10 WRAP-UP

Our robot modelling so far has been quite crude–using only primitive boxes and cylinders. Nonetheless, we have seen the basic elements of modeling, including kinematic, dynamic, visual and collision properties, as well as use of plug-ins for physically-realistic controllers. The simple examples illustrate modeling both of articulated mechanisms and mobile vehicles.

Making more sophisticated models involves extending the same techniques to more joints as well as bringing in CAD-file descriptions that offer more detailed and realistic modeling of visual and collision properties. A few examples examples of more realistic robot models are shown in Figures 3.22 through 3.24.

The Baxter [**?**] robot model in Fig 3.22 includes control over 15 joints (7 per arm plus a neck pan motion) plus grippers (which can be substituted). Sensor outputs include the dynamic state of every joint, streaming images from 3 color cameras, distance sensors from the wrists, and sonar sensors around the head. This model is realistic in terms of its kinematics (including joint limits), actuator behavior (including torque saturations), visual appearance and collision model. The robot is highly dexterous and well instrumented. Further, the model is publicly available [**?**]. Consequently, this model will be used in this text for further examples in use of ROS for sensing and manipulation.

The robot model in 3.23 is of a dual-arm “DaVinci” surgical robot [**?**]. The CAD descriptions have been posted by Johns Hopkins University [**?**]. Inertial parameters and ROS controller plug-ins were added to make this model Gazebo compliant. The resulting system is useful for testing extensions to computer-assisted robotic surgery and potentially development of a surgical training system.

Figure 3.24 shows a model of the Boston-Dynamics, Inc. “Atlas” robot [**?**] developed by the Open Source Robotics Foundation (OSRF) [**?**] for the DARPA Robotics Challenge [**?**]. This model was used by teams to develop code for the DARPA competition tasks, including both bipedal motion control and object manipulations (e.g., valve turning). Given the risk, difficulty and crew size required to perform physical experiments with Atlas, this model helped teams develop code more quickly, enabling individuals to make progress in parallel and to perform extensive software debugging and testing in simulation before moving to physical trials.

It should be appreciated that robot models in Gazebo include realistic physical interactions, including collisions, effects of gravity and inertial effects. Simulators that lack these properties can be used to evaluate viability of kinematic trajetories in terms of reachability, but simulators that lack a physics engine cannot evaluate dynamic effects, including walking or object manipulation.

An even stronger motivation for using a physics-based simulator is incorporation of sensor simulation. Gazebo is capable of simulating physical sensors acting in a virtual environment. Plug-ins for sensors include: color cameras, LIDARs, depth cameras (including the Microsoft Kinect [**?**]), stereo cameras, accelerometers, force sensors, and sonar sensors. With emulation of sensors, one can develop sensor-based behaviors that are testable in Gazebo.

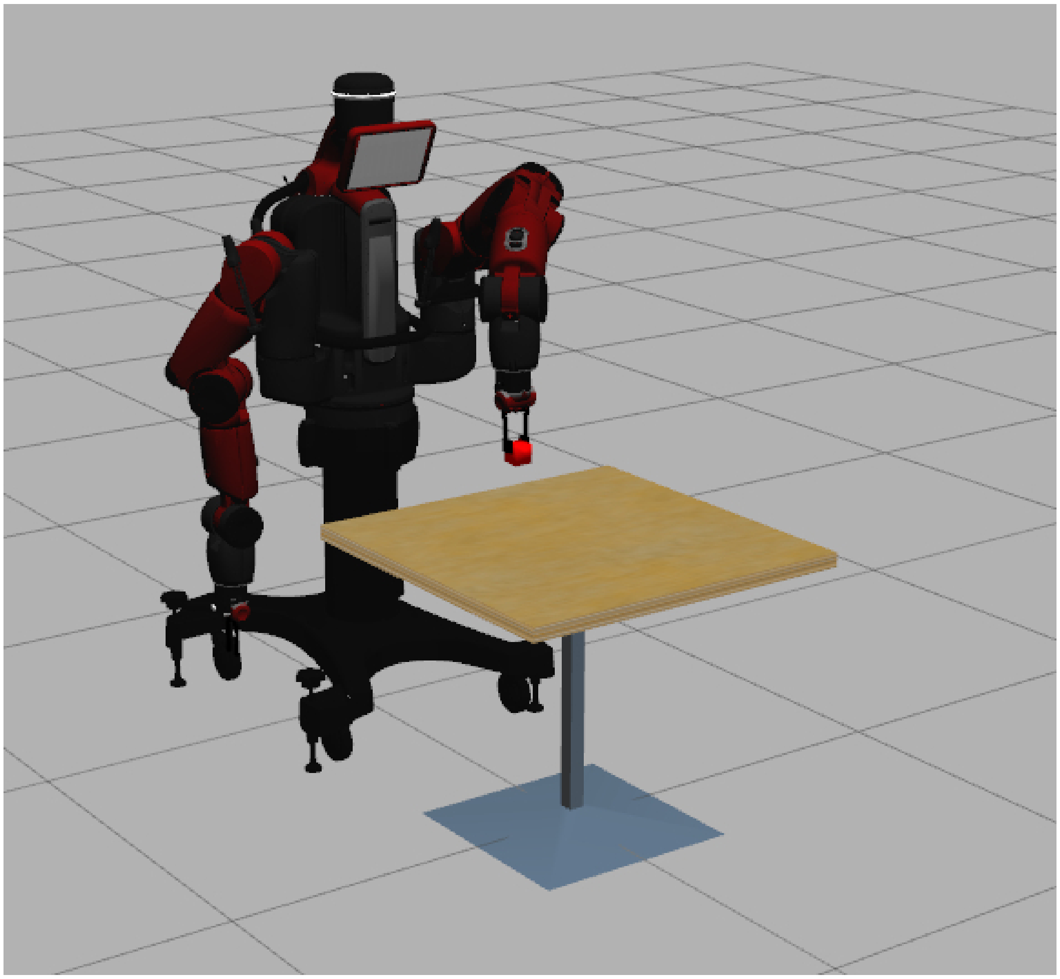


FIGURE 3.22: Gazebo display of Baxter robot model

Since such development can be time consuming, having a suitable simulator is a valuable productivity tool.

Emulation and visualization of sensor data will be covered next in Chapter 5.

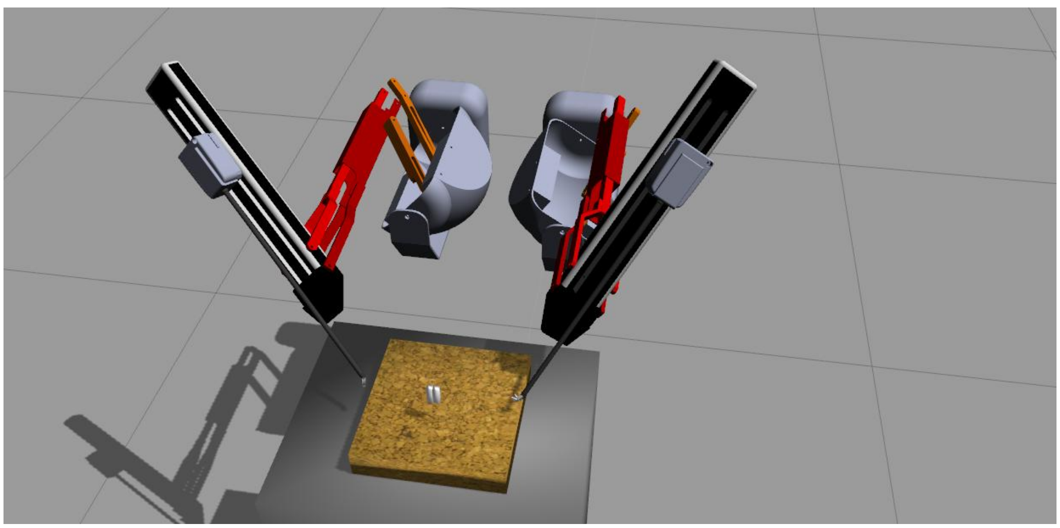


FIGURE 3.23: Gazebo display of DaVinci robot model

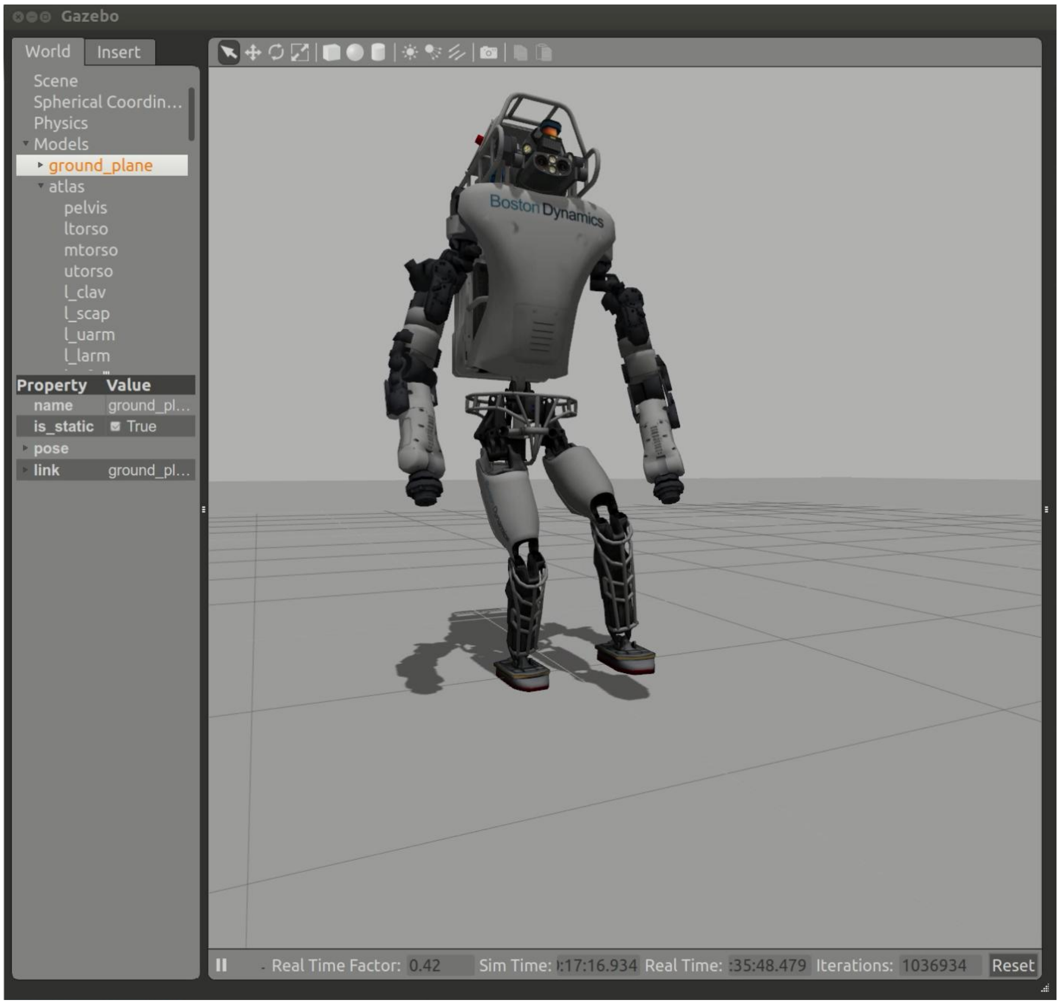


FIGURE 3.24: Gazebo display of Atlas robot model

CHAPTER 4

Coordinate Transforms in ROS

CONTENTS

INTRODUCTION Reconciliation of coordinates is necessary to take advantage of sensor-driven behaviors, whether in simulation or in physical control. Coordinating data from multiple models and sensors is accomplished through coordinate transformations. Fortunately, ROS has a extensive support for coordinate transformations. Since transforms are ubiquitous in ROS and in robotics in general, it is important to have an understanding of how coordinate transforms are handled in ROS.

4.1 INTRODUCTION TO COORDINATE TRANSFORMS IN ROS

Coordinate-frame transformations are fundamental to robotics. For articulated robot arms, full 6-DOF transforms are required to compute gripper poses as a function of joint angles. Multiplication of sequential transforms, link by link, performs such computations. Sensor data, e.g. from cameras, or LIDARs, is acquired in terms of the sensor's own frame, and this data must be interpreted in terms of alternative frames (e.g. world frame or robot frame).

Any introductory robotics text will cover coordinate-frame assignments and transformations (e.g. [?]). A brief introduction to coordinate-frame transformations is given here, leading to ROS's treatment of coordinate transforms.

A "frame" is defined by a point in 3-D space, $\mathbf{p}$, that is the frame's origin, and three vectors: $\mathbf{n}$, $\mathbf{t}$ and $\mathbf{b}$ (which define the local $\mathbf{x}$, $\mathbf{y}$ and $\mathbf{z}$ axes, respectively). The axis vectors are normalized (have unit length), and they form a right-hand triad, such that $\mathbf{n}$ crossed into $\mathbf{t}$ equals $\mathbf{b}$: $\mathbf{b} = \mathbf{n} \times \mathbf{t}$.

These three directional axes can be stacked side-by-side as column vectors, comprising a 3x3 matrix, $\mathbf{R}$.

$$\mathbf{R} = \begin{bmatrix} \mathbf{n} & \mathbf{t} & \mathbf{b} \end{bmatrix} = \begin{bmatrix} n_x & t_x & b_x \\ n_y & t_y & b_y \\ n_z & t_z & b_z \end{bmatrix} \tag{4.1}$$

We can include the origin vector as well to define a 3x4 matrix as:

$$\begin{bmatrix} \mathbf{n} & \mathbf{t} & \mathbf{b} & \mathbf{p} \end{bmatrix} = \begin{bmatrix} n_x & t_x & b_x & p_x \\ n_y & t_y & b_y & p_y \\ n_z & t_z & b_z & p_z \end{bmatrix} \tag{4.2}$$

A useful trick to simplify mathematical operations is to define an augmented matrix, converting the above 3x4 matrix in to a square 4x4 matrix by adding a fourth row consisting of [0 0 0 1]. This augmented matrix is a 4x4, which we will refer to as a **T** matrix:

$$\mathbf{T} = \begin{bmatrix} n_x & t_x & b_x & p_x \\ n_y & t_y & b_y & p_y \\ n_z & t_z & b_z & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.3}$$

Conveniently, matrices constructed as above (consistent with valid frame specifications) are always invertible. Further, computation of the inverse of a **T** matrix is efficient.

Abstractly, one can refer to an origin and a set of orientation axes (vectors) without having to specify numerical values. However, to perform computations, numerical values are required–and this requires further definitions. Specifically, when numerical values are given, one must define the coordinate system in which the values are measured. For example, to specify the origin of frame "B" (i.e. point **p**) with respect to frame "A", we can measure from the origin of frame "A" to the origin of frame "B" along three directions: the **x** axis of frame "A", the **y** axis of frame "A", and the **z** axis of frame "A". These measurements can be referred to explicitly as $p_{x/A}$, $p_{y/A}$, and $p_{z/A}$, respectively. If we had provided coordinates for the origin of frame B from any other viewpoint, the numerical values of the components of **p** would be different.

Similarly, we can describe the components of frame-B's **n** axis (correspondingly, **t** and **b** axes) with respect to frame A by measuring the x, y and z components along the respective axes of frame A.

Equivalently, this representation can be interpreted as the following operation. First, translate frame B such that its origin coincides with frame A–but do not change the orientation of frame B. The three axis vectors of frame B have tips that are each unit length from the (common) origin. These define three distinct points in space, and each of these axis tips of frame B can be expressed as 3-D points, e.g. as measured in frame A. Equivalently, the tip of the **n** axis of frame B is simply $\begin{bmatrix}1 & 0 & 0\end{bmatrix}^T$ in the B frame, but it is $\begin{bmatrix}n_{x/A} & n_{y/A} & n_{z/A}\end{bmatrix}^T$ with respect to the A frame.

We can then state the position and orientation of frame B with respect to frame A as:

$$\mathbf{{}^{A}T_{B}} = \mathbf{T_{B/A}} = \begin{bmatrix} n_{x/A} & t_{x/A} & b_{x/A} & p_{x/A} \\ n_{y/A} & t_{y/A} & b_{y/A} & p_{y/A} \\ n_{z/A} & t_{z/A} & b_{z/A} & p_{z/A} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.4}$$

We can refer to the above matrix as "frame B with respect to frame A."

Having labeled frames A and B, providing values for the elements of $\mathbf{{}^{A}T_{B}}$ fully specifies the position and orientation of frame B with respect to frame A.

In addition to providing a means to explicitly declare the position and orientation of a frame (with respect to some named frame, e.g. frame B with respect to frame A), **T** matrices can also be interpreted as operators. For example, if we know the position and orientation of frame C with respect to frame B, i.e. $\mathbf{{}^{B}T_{C}}$, and if we also know the position and orientation of frame B with respect to frame A, $\mathbf{{}^{A}T_{B}}$, then we can compute the position and orientation of frame C with respect to frame A as follows:

$$\mathbf{{}^{A}T_{C}} = \mathbf{{}^{A}T_{B}\,{}^{B}T_{C}} \tag{4.5}$$

That is, a simple matrix multiplication yields the desired transform. This process can be extended, e.g.:

$$\mathbf{{}^{A}T_{F}} = \mathbf{{}^{A}T_{B}\,{}^{B}T_{C}\,{}^{C}T_{D}\,{}^{D}T_{E}\,{}^{E}T_{F}} \tag{4.6}$$

In the above, the 4x4 on the left-hand side can be interpreted column by column. For example, the fourth column (rows 1 through 3), are the coordinates of the origin of frame "F" as measured with respect to frame "A."

By using the notation of prefix superscripts and post subscripts, there is a visual mnemonic to aid logical compatibility. The super and sub-scripts act like Lego™blocks, such that a subscript of a leading **T** matrix must match the pre-superscript of the trailing **T** matrix. Following this convention helps to keep transform operations consistent.

Since coordinate transforms are so common in robotics, ROS has provided a powerful package–the "tf" package (see `http://wiki.ros.org/tf`)–for handling transforms. Use of this package can be confusing, because "tf" performs considerable work.

To examine the `tf` topic, we can bring up gazebo with an empty world:

```
roslaunch gazebo_ros empty_world.launch
```

and then bring up our mobile robot with 1-DOF arm:

```
roslaunch mobot_urdf mobot_w_arm.launch
```

This results in 18 topics being published, including the topic `tf`. Issuing:

```
rostopic info tf
```

shows that the topic "tf" carries messages of type `tf2_msgs/TFMessage`.

Running:

```
    rosmsg show tf2_msgs/TFMessage
```

reveals that the `tf2_msgs/TFMessage` is organized as follows:

```
geometry_msgs/TransformStamped[] transforms
  std_msgs/Header header
    uint32 seq
    time stamp
    string frame_id
  string child_frame_id
  geometry_msgs/Transform transform
    geometry_msgs/Vector3 translation
      float64 x
      float64 y
      float64 z
    geometry_msgs/Quaternion rotation
      float64 x
      float64 y
      float64 z
      float64 w
```

That is, this message contains a vector (variable-length array) of messages of type `geometry_msgs/TransformStamped`.

The ROS transform datatype is not identical to a 4x4 homogeneous transformation matrix, but it carries equivalent information (and more). The ROS transform datatype contains a 3-D vector (equivalent to the 4th column of a 4x4 transform) and a quaternion (an alternative representation of orientation). In addition, transform messages have time stamps, and they explicitly name the child frame and the parent frame (as text strings).

The `tf` topic is being published to by gazebo. There can be (and typically are) many publishers to the `tf` topic. Each publisher expresses a transform relationship, describing a named "child" frame with respect to a named "parent" frame. Gazebo is publishing to `tf` because, within the differential-drive plug-in, this was requested with the "publishWheelTF" option:

```
<gazebo>
  <plugin name="differential_drive_controller" filename="libgazebo_ros_diff_drive.so">
     <publishWheelTF>true</publishWheelTF>
     <publishWheelJointState>true</publishWheelJointState>
```

(Note that `publishWheelJointState` was also enabled for this plug-in).

Running `rostopic hz tf` shows that the `tf` topic is being updated at 300Hz. Examining the output of `tf` with:

```
rostopic echo tf
```

shows that the "transforms" component (a variable-length array) of this message includes multiple, individual transform relationships. An (abbreviated) excerpt from the tf echo output is:

```
    frame_id: base_link
    child_frame_id: left_wheel
    transform:
      translation:
        x: -9.1739781936e-08
        y: 0.282574370084
        z: 0.165117569901
      rotation:
        x: -0.497142106739
        y: 0.502848355739
        z: 0.502843417149
        w: 0.497133538064

    frame_id: base_link
    child_frame_id: right_wheel
    transform:
      ...
```

The link `left_wheel` has a defined reference frame, as does the `base_link`. From the `tf` output, we see that a transform is expressed between the base-link frame and the left-wheel frame (and similarly between the base-link frame and the right-wheel frame). The transform from the base-link frame to the left-wheel frame has a translational part and a rotational part.

The translation from the origin of the base frame to the origin of the left-wheel frame is approximately `[0,  0.283, 0.165]`. These values are interpretable in terms of our robot model. Our base frame has its x axis pointing forward, its y axis pointing to the left, and its z axis pointing up. The origin of the base frame is at ground level, immediately below the point between the two wheels. Consequently, the x value of the left-wheel origin is 0 (neither in front of nor behind the base-frame origin). The y-value of the left-wheel origin is positive 0.283 (half the track width of the vehicle). The z-value of the left-wheel origin is equal to the wheel radius (i.e., the axle is above ground level by one wheel radius).

The left wheel frame is also rotated relative to the base frame. It's z axis points to the left of the base frame, parallel to the base-frame's y axis. However, the wheel's x and y axes will change their orientations as the wheel rotates (i.e., they are a function of the `left_wheel_joint` rotation value).

In ROS, orientations are commonly expressed as unit quaternions. The quaternion representation of orientation is an alternative to rotation matrices. Quaternions can be converted to 3x3 rotation matrices (and used within 4x4 coordinate transform matrices), or quaternions can be used directly with mathematically-defined quaternion operations for coordinate transforms. Quaternions are more compact than 3x3 rotation matrices and they have attractive mathematical properties. However, they are not as intuitive as 3x3 rotation matrices for visualizing orientations in terms of **n**, **t** and **b** axes.

Details of representations and mathematical operations with quaternions will not be covered here but can be found in many robotics textbooks (e.g. [**?**]). At this point, it is sufficient to know that there is a correspondence betweeen quaterions and rotation matrices (and ROS functions to perform such conversions will be introduced later), and that there are corresponding mathematical operations for coordinate transformations with quaternions.

Another topic to which gazebo is publishing is `joint_states`, which carries messages of type `sensor_msgs/JointState`. Running:

```
rostopic echo joint_states
```

shows output such as the (abbreviated) following:

```
name: ['left_wheel_joint', 'right_wheel_joint']
position: [0.005513943571622271, 0.007790399443280194]
```

These messages list the names of joints (e.g. `left_wheel_joint`), then list states of these joints (including positions) in the same order as the list of joint names (e.g. the left wheel joint angle is 0.0055 rad, per the example message). This information is needed as part of the computation of transforms. The transform of the left-wheel frame, e.g., cannot be known without knowing the value of the wheel rotation.

The wheel rotation values are published on `joint_states` because the model URDF contains the gazebo option: `<publishWheelJointState>true</publishWheelJointState>`. However, the `joint_states` topic does not contain other angles, including the caster wheel angles nor the arm joint angle (joint1).

The caster-wheel angles are known to gazebo, and these joint angles can be published to a ROS topic by including another gazebo plug-in in the model. The model file `mobot_w_jnt_pub.xacro` (in the `mobot_urdf` package) is the same as `mobot2.xacro`, except for the addition of the following lines:

```
<gazebo>
   <plugin name="joint_state_publisher" filename="↩
       libgazebo_ros_joint_state_publisher.so">
     <jointName>cast2bracket_right, cast2bracket_left, right_caster_joint, ↩
         left_caster_joint </jointName>
   </plugin>
</gazebo>
```

This gazebo plug-in accesses the internal gazebo dynamic model to obtain the joint values for the named joints, and it publishes these values to the topic `joint_states`.

Similarly, the model file `minimal_robot_w_jnt_pub.urdf` in the `minimal_robot_description` package is identical to `minimal_robot_description_unglued.urdf`, except for the addition of the following lines:

```
<gazebo>
```

```
    <plugin name="joint_state_publisher" filename="↩
        libgazebo_ros_joint_state_publisher.so">
      <jointName>joint1</jointName>
    </plugin>
  </gazebo>
```

which invokes the `libgazebo_ros_joint_state_publisher.so` gazebo plug-in to publish the values of `joint1` angles to the `joint_states` topic.

These two modified files are included in a common model file, `mobot_w_arm_and_jnt_pub.xacro` (in the `mobot_urdf` package), as follows:

```
<?xml version="1.0"?>
<robot
     xmlns:xacro="http://www.ros.org/wiki/xacro" name="mobot">
  <xacro:include filename="$(find mobot_urdf)/urdf/mobot_w_jnt_pub.xacro" />
  <xacro:include filename="$(find minimal_robot_description)/minimal_robot_w_jnt_pub.↩
      urdf" />

  <!-- attach the simple arm to the mobile robot -->
  <joint name="arm_base_joint" type="fixed">
    <parent link="base_link" />
    <child link="link1" />
    <origin rpy="0 0 0 " xyz="${-bodylen/2} 0 ${bodyOZ+bodyheight/2}"/>
  </joint>
</robot>
```

The `mobot_w_arm_and_jnt_pub.xacro` model file is identical to `mobot_w_arm.xacro`, except that it includes the modified mobot and arm model files. The `mobot_w_arm_and_jnt_pub.xacro` model file is referenced in a launch file, `mobot_w_arm_and_jnt_pub.launch`:

```
<launch>
<!-- Convert xacro model file and put on parameter server -->
<param name="robot_description" command="$(find xacro)/xacro.py '$(find mobot_urdf)/↩
    urdf/mobot_w_arm_and_jnt_pub.xacro'" />

<!-- Spawn the robot from parameter server into Gazebo -->
<node name="spawn_urdf" pkg="gazebo_ros" type="spawn_model" args="-param ↩
    robot_description -urdf -model mobot" />

<!-- load the controller parameter yaml file and start the ROS controllers for the arm↩
    -->
<include file="$(find minimal_robot_description)/minimal_robot_ctl.launch">
</include>

</launch>
```

To observe the effects of the joint-state publisher gazebo plug-in, kill and restart gazebo:

```
roslaunch gazebo_ros empty_world.launch
```

and then invoke the new launch file:

```
roslaunch mobot_urdf mobot_w_arm_and_jnt_pub.launch
```

This again brings up the mobile robot with attached 1-DOF arm. Now, however, the `joint_states` topic is richer. Example output of `rostopic echo joint_states` (excerpted) is:

```
name: ['left_wheel_joint', 'right_wheel_joint']
position: [-0.0010584164794300577, -0.0007484677653710747]

name: ['cast2bracket_right', 'cast2bracket_left', 'right_caster_joint',
       'left_caster_joint']
```

```
position: [0.12480710950733798, 0.090932225345286, 0.21368044937114838,
          0.2107250800969398]

name: ['joint1']
position: [0.17308318317281302]
```

We see that now there are 7 joint values being published. These values are needed to compute transforms for link poses. Given the model description, which prescribes how connected links relate to each other through joint displacements, it is possible to compute all individual transforms for pair-wise connected links. This could be done manually from the information available. However, this is such a common need that ROS has a package designed to do this: the `robot_state_publisher` (see `http://wiki.ros.org/robot_state_publisher`).

With the simulation running, invoking:

```
rosrun robot_state_publisher robot_state_publisher
```

results in the `tf` topic being much richer as well. Excerpts from the output of `rostopic echo tf` follow:

```
      frame_id: base_link
    child_frame_id: castdrop_right
      ...(static)

      frame_id: castdrop_right
    child_frame_id: brackettop_right
        transform:
      translation:
        x: 0.0
        y: 0.0
        z: -0.0625
      rotation:
        x: 0.0
        y: 0.0
        z: 0.947030895892
        w: 0.321142464065

      frame_id: brackettop_right
    child_frame_id: bracketside1_right
      ...(static)

      frame_id: bracketside1_right
    child_frame_id: right_casterwheel
    transform:
      translation:
        x: -0.0613
        y: 0.053
        z: -0.053
      rotation:
        x: 0.673542175535
        y: -0.215269453877
        z: 0.673542175539
        w: -0.215269453878

     frame_id: base_link
    child_frame_id: link1
```

```
transform:
  translation:
    x: -0.27305
    y: 0.0
    z: 0.55
  rotation:
    x: 0.0
    y: 0.0
    z: 0.0
    w: 1.0

   frame_id: link1
child_frame_id: link2
transform:
  translation:
    x: 0.0
    y: 0.0
    z: 1.0
  rotation:
    x: 0.0
    y: 0.0864419753897
    z: 0.0
    w: 0.996256886998
```

From the `tf` output, one can follow chains of transforms. One kinematic branch starts at `base_link` and has successive parent/child connections through `castdrop_right` (a static transform) to `brackettop_right` (which swivels about its z axis, as evidenced by the quaternion values), to `bracketside1_right` (a static transform) to `right_casterwheel` (which spins about its axle). The individual transforms can be multiplied together (an operation defined in the transform class, equivalent to multiplying **T** matrices) to find the pose of any link frame with respect to any desired reference frame (as long as there is a complete chain between the two frames).

Note also the transform between `base_frame` and `link1` (a static transform) and the transform between `link1` and `link2` (which depends on the `joint1` angle). From these transforms, one can obtain the frame of link2 of the one-DOF robot with respect to the base link. ROS also provides facilities to compute transforms between any 2 (connected) frames within a user's program with a `transform_listener`, details of which are introduced in Section 4.2.

With an understanding of transforms in ROS, we are ready to introduce the valuable ROS visualization tool "rviz."

4.2 THE TRANSFORM LISTENER

Performing transforms from one frame to another is enabled by the "tf" library in ROS (see: `http://wiki.ros.org/tf/Tutorials`). A very useful capability in the tf library is the `tf_listener`. We have seen that transforms are published to the topic "tf," where each such message contains a detailed description of how a child frame is spatially related to its parent frame. (In ROS, a parent can have multiple children, but a child must have a unique parent, thus guaranteeing a "tree" of geometric relationships). The `tf_listener` is typically started as an independent thread (thus not dependent on spin() or spinOnce() invokations from a main program). This thread subscribes to the tf topic and assembles a kinematic tree from individual parent/child transform messages. Since the `tf_listener` incorporates all transform publications, it is able to address specific queries, such as "where is my right-hand palm frame relative to my left-camera optical frame." The response to such a query is a transform message that can be used to reconcile different frames (e.g., for hand/eye coordination). As long as a complete tree is published connecting frames of interest, the transform listener can be used to transform all sensor data into a common reference frame, thus allowing for display of sensory data from multiple sources in a common view.

Use of the tf listener is illustrated with example code in the package `example_tf_listener` of the accompanying code repository. This illustrative package specifically assumes reference to our mobot model, notably with respect to frames `base_link`, `link1` and `link2`. The example code is comprised of three files: `example_tf_listener.h`, which defines a class `DemoTfListener`; the file `example_tf_listener_fncs.cpp`, which contains the implementation of the class methods of `DemoTfListener`; and the file `example_tf_listener.cpp`, which contains a main() program illustrating operations using a transform listener. The contents of the main program are shown in Listing 4.1.

Listing 4.1: Example code illustrating use of a transform listener

```cpp
//example_tf_listener.cpp:
//wsn, March 2016
//illustrative node to show use of tf listener, with reference to the simple mobile-robot model
// specifically, frames: odom, base_frame, link1 and link2

// this header incorporates all the necessary #include files and defines the class "DemoTfListener"
#include "example_tf_listener.h"
using namespace std;

//main pgm to illustrate transform operations

int main(int argc, char** argv) {
    // ROS set-ups:
    ros::init(argc, argv, "demoTfListener"); //node name
    ros::NodeHandle nh; // create a node handle; need to pass this to the class constructor
    ROS_INFO("main: instantiating an object of type DemoTfListener");
    DemoTfListener demoTfListener(&nh); //instantiate an ExampleRosClass object and pass in pointer to nodehandle for constructor to use

    tf::StampedTransform stfBaseToLink2, stfBaseToLink1, stfLink1ToLink2;
    tf::StampedTransform testStfBaseToLink2;

    tf::Transform tfBaseToLink1, tfLink1ToLink2, tfBaseToLink2, altTfBaseToLink2;

    demoTfListener.tfListener_->lookupTransform("base_link", "link1", ros::Time(0), stfBaseToLink1);
    cout << endl << "base to link1: " << endl;
    demoTfListener.printStampedTf(stfBaseToLink1);
    tfBaseToLink1 = demoTfListener.get_tf_from_stamped_tf(stfBaseToLink1);

```

```
29      demoTfListener.tfListener_->lookupTransform("link1", "link2", ros::Time(0), stfLink1ToLink2);
30      cout << endl << "link1 to link2: " << endl;
31      demoTfListener.printStampedTf(stfLink1ToLink2);
32      tfLink1ToLink2 = demoTfListener.get_tf_from_stamped_tf(stfLink1ToLink2);
33
34      demoTfListener.tfListener_->lookupTransform("base_link", "link2", ros::Time(0), stfBaseToLink2);
35      cout << endl << "base to link2: " << endl;
36      demoTfListener.printStampedTf(stfBaseToLink2);
37      tfBaseToLink2 = demoTfListener.get_tf_from_stamped_tf(stfBaseToLink2);
38      cout << endl << "extracted tf: " << endl;
39      demoTfListener.printTf(tfBaseToLink2);
40
41      altTfBaseToLink2 = tfBaseToLink1*tfLink1ToLink2;
42      cout << endl << "result of multiply tfBaseToLink1*tfLink1ToLink2: " << endl;
43      demoTfListener.printTf(altTfBaseToLink2);
44
45      if (demoTfListener.multiply_stamped_tfs(stfBaseToLink1, stfLink1ToLink2, testStfBaseToLink2)) {
46          cout << endl << "testStfBaseToLink2:" << endl;
47          demoTfListener.printStampedTf(testStfBaseToLink2);
48      }
49      cout << endl << "attempt multiply of stamped transforms in wrong order:" << endl;
50      demoTfListener.multiply_stamped_tfs(stfLink1ToLink2, stfBaseToLink1, testStfBaseToLink2);
51
52      geometry_msgs::PoseStamped stPose, stPose_wrt_base;
53      stPose = demoTfListener.get_pose_from_transform(stfLink1ToLink2);
54      cout << endl << "pose link2 w/rt link1, from stfLink1ToLink2" << endl;
55      demoTfListener.printStampedPose(stPose);
56
57      demoTfListener.tfListener_->transformPose("base_link", stPose, stPose_wrt_base);
58      cout << endl << "pose of link2 transformed to base frame:" << endl;
59      demoTfListener.printStampedPose(stPose_wrt_base);
60
61      return 0;
62  }
```

In Listing 4.1, an object of class DemoTfListener is instantiated (line 17). This object has a transform listener, a pointer to which is `tf::TransformListener* tfListener_;`. The transform listener is used in four places in the main program: lines 24, 29, 34 and 57.

The first instance, line 24, is:

```
demoTfListener.tfListener_->lookupTransform("base_link", "link1", ros::Time(0), stfBaseToLink1);
```

The transform listener subscribes to the topic "tf," and it constantly attempts to assemble the most current chain of transforms possible from all published parent/child spatial relationships. The transform listener, once it has been instantiated and given a brief time to start collecting published transform information, offers a variety of useful methods. In the above case, the method "lookupTransform" finds a transformation between the named frames (`base_link` and `link1`). Equivalently, this transform tells us where is the link1-frame with respect to the `base_link` frame. The "lookupTransform" method fills in "stfBaseToLink1," which is an object of type `tf::StampedTransform`. A stamped transform contains both an origin (a 3-D vector) and an orientation(a 3x3 matrix). The example code prints components of this transform, using accessor functions of transform objects.

The `lookupTransform()` function is called with arguments to define the frame of interest (`link1`) and the desired reference frame (`base_link`). The argument `ros::Time(0)` specifies that the current transform is desired. (Optionally, one can request a transform a historical transform corresponding to some specified time of interest in the past).

The object `stfBaseToLink1` has a time-stamp, labels for the reference frame (`frame_id`) and for the child frame (`child_frame_id`), and a `tf::Transform` object. Objects of type `tf::Transform` have a variety of member methods and defined operators. Extracting the `tf::Transform` from a `tf::StampedTransform` object is not as simple as would be expected. The function `get_tf_from_stamped_tf()` is defined within the class DemoTfListener to assist. On line 27,

```
tfBaseToLink1 = demoTfListener.get_tf_from_stamped_tf(stfBaseToLink1);
```

the Transform "tfBaseToLink1" is extracted from the StampedTransform "stfBaseToLink1." This transform describes the position and orientation of frame `link1` with respect to frame `base_frame`.

On line 29, the stamped-transform "stfLink1ToLink2" is obtained using the transform listener, but with specified frame of interest "link2" with respect to frame "link1." The transform is extracted from the stamped transform into the object "tfLink1ToLink2."

Lines 34 and 37 perform this operation again, this time to populate the object "tfBaseToLink2," which is the transform of `link2` with respect to `base_frame`.

The operator "*" is defined for `tf::Transform` objects. Thus, the objects `tfBaseToLink1` and `tfLink1ToLink2` can be multiplied together, as is done in line 41:

```
altTfBaseToLink2 = tfBaseToLink1*tfLink1ToLink2;
```

The meaning of this operation is to cascade these transforms, equivalent to multiplying 4x4 transforms:

$$ {}^{\mathbf{A}}\mathbf{T}_{\mathbf{C}} = {}^{\mathbf{A}}\mathbf{T}_{\mathbf{B}}{}^{\mathbf{B}}\mathbf{T}_{\mathbf{C}} \tag{4.7} $$

The result in "altTfBaseToLink2" is the same as the result of

```
demoTfListener.tfListener_->lookupTransform("base_link", "link2", ros::Time(0), ↩
    stfBaseToLink2);
```

and extracting the transform from `stfBaseToLink2`.

The example code displays the various transforms using the member functions `printTf()` and `printStampedTf()` of class DemoTfListener.

Line 57 shows use of another member method of the transform listener:

```
demoTfListener.tfListener_->transformPose("base_link", stPose, stPose_wrt_base);
```

With this function, one can transform an object of type `geometry_msgs::PoseStamped`. The position and orientation of this pose is expressed with respect to the named frame id. With the transformPose() function, the input PoseStamped object, `stPose`, is re-expressed as an output pose, `stPose_wrt_base`, expressed with respect to the named, desired frame (`base_link`, in this example).

To see the results of the example code, run the following: Start up gazebo with:

```
roslaunch gazebo_ros empty_world.launch
```

and invoke the mobot launch file:

```
roslaunch mobot_urdf mobot_w_arm_and_jnt_pub.launch
```

Running `rostopic echo tf`, one can see transforms published that include relationships among `odom`, `base_link`, `left_wheel` and `right_wheel`, which are published courtesy of the differential-drive gazebo plug-in. To get more transforms, including the minimal robot arm, run:

```
rosrun robot_state_publisher robot_state_publisher
```

The "tf" topic then carries many additional transform messages, including `link2` to `link1` (of the minimal robot arm) and `link1` to `base_link`.

With these nodes running, start up the example transform listener:

```
rosrun example_tf_listener example_tf_listener
```

The example transform listener output begins with:

```
[ INFO] [1457913167.079553126]: main: instantiating an object of type DemoTfListener
[ INFO] [1457913167.079639652]: in class constructor of DemoTfListener
[ INFO] [1457913167.097079718]: waiting for tf between link2 and base_link...
[ WARN] [1457913167.097393435]: "base_link" passed to lookupTransform argument
     target_frame does not exist. ; retrying...
[ WARN] [1457913167.843127914, 414.927000000]: Lookup would require extrapolation into the past.
            Requested time 414.914000000 but the earliest data is at time 414.934000000,
            when looking up transform from frame [link2] to frame [base_link]; retrying...
[ INFO] [1457913168.344223832, 415.427000000]: tf is good
```

Initially, the transform listener does not have knowledge of all of the incremental transforms of the kinematic tree. The transform listener call thus fails. This failure is trapped, and the call is re-attempted. By the time of the next try, the connecting transforms are all known, and the call is successful. Ordinarily, successive attempts to find transforms between any two named frames will be successful. However, it is still good practice to use "try/catch," in case of future missing transforms.

The first result displayed is:

```
base to link1:
frame_id: base_link
child_frame_id: link1
vector from reference frame to to child frame: -0.27305,0,0.55
orientation of child frame w/rt reference frame:
1,0,0
0,1,0
0,0,1
quaternion: 0, 0, 0, 1
```

This shows that the transform betwee frame `base_link` and frame `link1` is relatively simple. The orientation is merely the identity, indicating that the link1-frame is aligned with the base-link frame. The vector from the base-link frame origin to the link1-frame origin has a negative x component, a 0 y component and a positive z component. This makes intuitive sense, as the link1-frame origin is above the base-link origin (and thus positive z component) and "behind" the base-link frame origin (and thus the negative x component). The quaternion corresponding to the identity matrix orientation is (0,0,0,1).

The next part of the display output is:

```
link1 to link2:
frame_id: link1
child_frame_id: link2
vector from reference frame to to child frame: 0,0,1
orientation of child frame w/rt reference frame:
 0.978314,  0, 0.207128
 0,         1, 0
-0.207128,  0, 0.978314
quaternion: 0, 0.10413, 0, 0.994564
```

The vector from the link1-frame origin to the link2-frame origin, (0, 0, 1), is simply a 1-meter displacement in the z direction. The link2 frame is nearly aligned with the link1 frame. The y axis (2nd column of the rotation matrix) is (0,1,0), which implies the link2 y axis is identically parallel to the link1 y axis. However, the x and z axes of the link2 frame are not identical to the corresponding link1 axes. Since link2 is leaning slightly forward, the link2 z-axis, (0.207, 0, 0.078), has a slight postive x component (as expressed with respect to the link1 frame), and the link2 x axis, (0.978, 0, -0.207), points slightly down, and thus has a negative z component (with respect to the link1 frame).

The next display output corresponds to the transform between link2 and the base link.

```
base to link2:
frame_id: base_link
child_frame_id: link2
vector from reference frame to to child frame: -0.27305,0,1.55
orientation of child frame w/rt reference frame:
0.978314,0,0.207128
0,1,0
-0.207128,0,0.978314
quaternion: 0, 0.10413, 0, 0.994564
```

For comparison, the output of the product `tfBaseToLink1` and `tfLink1ToLink2` is:

```
result of multiply tfBaseToLink1*tfLink1ToLink2:
vector from reference frame to to child frame: -0.27305,0,1.55
orientation of child frame w/rt reference frame:
0.978314,0,0.207128
0,1,0
-0.207128,0,0.978314
quaternion: 0, 0.10413, 0, 0.994564
```

The components of this result are identical to those of the transform lookup directly from the base link to link2, demonstrating that products of transforms behave equivalent to 4x4 matrix transform multiplies (although the result of `tf::Transform` multiplies are objects of type `tf::Transform`, not merely matrices).

Multiplication of `tf::StampedTransform` objects is not defined. However, a member function of class DemoTfListener performs the equivalent operation. Line 45:

```
if (demoTfListener.multiply_stamped_tfs(stfBaseToLink1, stfLink1ToLink2, ↩
    testStfBaseToLink2))
```

performs the expected operation. The transform components of the stamped transforms are extracted, these are multiplied together, and they are used to populate the transform component of a resulting stamped transform, `testStfBaseToLink2`. The example function gets assigned the frame-id of the first stamped transform and the child-id of the second stamped transform. However, for the multiplication to make sense, the child-id of the first stamped transform must be identical to the frame-id of the second stamped transform. If this condition is not satisfied, the multiplication function returns "false" to indicate a logic error. (See line 50).

Finally, lines 52-59 illustrate how to transform a pose to a new frame. The output display is:

```
pose link2 w/rt link1, from stfLink1ToLink2
frame id = link1
origin: 0, 0, 1
quaternion: 0, 0.10413, 0, 0.994564
```

```
pose of link2 transformed to base frame:
frame id = base_link
origin: -0.27305, 0, 1.55
quaternion: 0, 0.10413, 0, 0.994564
```

Note that the transformed link2 pose has a translation and a rotation that are identical to the corresponding components of the stamped transform `stfBaseToLink2` obtained using `tfListener_->lookupTransform`.

Another check on this transform can be obtained with use of a command-line tool within the “tf” package. Running:

```
rosrun tf tf_echo base_link link2
```

results in output to the screen displaying the transform between the named frames. The order of naming matters– the above command displays the frame of `link2` with respect to frame `base_link`. Example output from this command is:

```
At time 23.585
- Translation: [-0.273, 0.000, 1.550]
- Rotation: in Quaternion [0.000, 0.103, 0.000, 0.995]
            in RPY (radian) [0.000, 0.207, 0.000]
            in RPY (degree) [0.000, 11.850, 0.000]
```

which agrees with the transform-listener result.

Converting between `geometry_msgs` types and `tf` types can be tedious. The code in Listing 4.2, extracted from `example_tf_listener_fncs.cpp`, illustrates how to extract a `geometry_msgs::PoseStamped` from a `tf::StampedTransform`.

Listing 4.2: Example conversion from tf to geometry-msgs type

```
geometry_msgs::PoseStamped DemoTfListener::get_pose_from_transform(tf::StampedTransform tf) {
  //clumsy conversions--points, vectors and quaternions are different data types in tf vs geometry_msgs
  geometry_msgs::PoseStamped stPose;
  geometry_msgs::Quaternion quat;  //geometry_msgs object for quaternion
  tf::Quaternion tfQuat; // tf library object for quaternion
  tfQuat = tf.getRotation(); // member fnc to extract the quaternion from a transform
  quat.x = tfQuat.x(); // copy the data from tf-style quaternion to geometry_msgs-style quaternion
  quat.y = tfQuat.y();
  quat.z = tfQuat.z();
  quat.w = tfQuat.w();
  stPose.pose.orientation = quat; //set the orientation of our PoseStamped object from result

  // now do the same for the origin--equivalently, vector from parent to child frame
  tf::Vector3 tfVec;  //tf-library type
  geometry_msgs::Point pt; //equivalent geometry_msgs type
  tfVec = tf.getOrigin(); // extract the vector from parent to child from transform
  pt.x = tfVec.getX(); //copy the components into geometry_msgs type
  pt.y = tfVec.getY();
  pt.z = tfVec.getZ();
  stPose.pose.position= pt; //and use this compatible type to set the position of the PoseStamped
  stPose.header.frame_id = tf.frame_id_; //the pose is expressed w/rt this reference frame
  stPose.header.stamp = tf.stamp_; // preserve the time stamp of the original transform
  return stPose;
}
```

Although vectors and quaternions are defined within `geometry_msgs` and `tf`, these types are not compatible. The code in 4.2 shows how these can be converted.

When instantiating a transform listener, it was noted that the lookup function should be tested for return errors. This is done in the constructor of DemoTfListener, as shown in Listing 4.3, extracted from `example_tf_listener_fncs.cpp`.

Listing 4.3: Constructor of DemoTfListener, illustrating testing a tf listener

```
DemoTfListener::DemoTfListener(ros::NodeHandle* nodehandle):nh_(*nodehandle)
{
    ROS_INFO("in class constructor of DemoTfListener");
    tfListener_ = new tf::TransformListener;  //create a transform listener and assign its pointer
    //here, the tfListener_ is a pointer to this object, so must use -> instead of "." operator
    //somewhat more complex than creating a tf_listener in "main()", but illustrates how
    // to instantiate a tf_listener within a class

    // wait to start receiving valid tf transforms between base_link and link2:
    // this example is specific to our mobot, which has a base_link and a link2
    // lookupTransform will through errors until a valid chain has been found from target to source frames
    bool tferr=true;
    ROS_INFO("waiting for tf between link2 and base_link...");
    tf::StampedTransform tfLink2WrtBaseLink;
    while (tferr) {
        tferr=false;
        try {
                //try to lookup transform, link2-frame w/rt base_link frame; this will test if
            // a valid transform chain has been published from base_frame to link2
                tfListener_->lookupTransform("base_link", "link2", ros::Time(0), tfLink2WrtBaseLink);
            } catch(tf::TransformException &exception) {
                ROS_WARN("%s; retrying...", exception.what());
                tferr=true;
                ros::Duration(0.5).sleep(); // sleep for half a second
                ros::spinOnce();
            }
    }
    ROS_INFO("tf is good");
    // from now on, tfListener will keep track of transforms; do NOT need ros::spin(), since
    // tf_listener gets spawned as a separate thread
}
```

The try/catch construct is used to trap errors from the lookupTransform() function of the transform listener. When using a transform-listener lookup function, it is advisable to always have a try/catch test. Otherwise, if the lookup function fails, the main program will crash.

Another concern with the transform listener is clock synchronization of multiple computers running nodes within a common ROS system. ROS supports distributed processing. However, with a network of computers, each computer has its own clock. This can result in time stamps within published transforms that can be out of synchronization. The transform listener may complain that some transforms appear to be posted for times in the future. Resolution of this problem may require using "chrony" (see `http://chrony.tuxfamily.org/`) or some alternative network time protocol clock synchronization.

Additional functions in `example_tf_listener_fncs.cpp` include: `multiply_stamped_tfs()`, `get_tf_from_stamped_tf()`, `get_pose_from_transform()`, `printTf()`, `printStampedTf()` and `printStampedPose()`. These will not be covered in detail here, but viewing the source code can be useful in understanding how to access or populate components of transform types.

Conversions among `geometry_msgs` types and `tf` types can be inconvenient, as shown in the illustrative code. An alternative library, "Eigen," is more convenient for performing linear-algebra operations, as introduced next.

4.3 USING THE EIGEN LIBRARY

ROS messages require serialization for network communication. As a result, these messages can be inconvenient to work with when one desires to perform operations on data. One common need is to perform linear-algebra operations. A useful C++ library for linear algebra is the "Eigen" library (see `http://eigen.tuxfamily.org`). The Eigen open-source project is independent of ROS. However, one can still use Eigen with ROS.

To use Eigen with ROS, one must include the associated header files in the `*.cpp` source code and add some lines to `CmakeLists.txt`. Our custom `cs_create_pkg` script already includes the necessary lines in `CmakeLists.txt`–they only need to be uncommented. Specifically, uncomment the following lines in the CMakeLists.txt file:

```
#uncomment the following 4 lines to use the Eigen library
find_package(cmake_modules REQUIRED)
find_package(Eigen3 REQUIRED)
include_directories(${EIGEN3_INCLUDE_DIR})
add_definitions(${EIGEN_DEFINITIONS})
```

(see `https://github.com/ros/cmake_modules/blob/0.3-devel/README.md#usage` for an explanation of `cmake_modules` in ROS). In the source code, include the header files for the functionality desired. The following include much functionality:

```
#include <Eigen/Eigen> //for the Eigen library
#include <Eigen/Dense>
#include <Eigen/Geometry>
#include <Eigen/Eigenvalues>
```

For access to additional capabilities in Eigen, more header files may be included, as described in `http://eigen.tuxfamily.org/dox/group__QuickRefPage.html#QuickRef_Headers`

An example program that illustrates some Eigen capabilities is `example_eigen_plane_fit.cpp` in the package `example_eigen`. Lines from this program are explained here.

An example vector may be defined e.g. as follows:

```
Eigen::Vector3d normal_vec(1,2,3); // here is an arbitrary 3x1 vector, initialized to ↩
    (1,2,3) upon instantiation
```

This instantiates an Eigen object that is a column vector comprised of 3 double-precision values. The object is named `normal_vec` and it is initialized to the values [1;2;3].

One of the member functions of `Eigen::Vector3d` is `norm()`, which computes the Euclidean length of the vector (square root of the sum of squares of the components). The vector can be coerced to unit length (if it is a non-zero vector!) as follows:

```
normal_vec/=normal_vec.norm(); // make this vector unit length
```

Note that, although `normal_vec` is an object, the operators "*" and "/" are defined to scale the components of the vector as expected. Thus, vector*scalar operations behave as expected (where `normal_vec.norm()` returns a scalar value).

Here is an example of instantiating a 3x3 matrix object comprised of double-precision values:

```
Eigen::Matrix3d Rot_z;
```

With the following notation, one can fill the matrix with data, one row at a time:

```
Rot_z.row(0)<<0,1,0;  // populate the first row--shorthand method
Rot_z.row(1)<<1,0,0;  //second row
Rot_z.row(2)<<0,0,1;  // third row
```

There are a variety of other methods for initializing or populating matrices and vectors. For example, one can fill a vector or matrix with zeros using:

```
Eigen::Vector3d centroid;
// here's a convenient way to initialize data to all zeros; more variants exist
centroid = Eigen::MatrixXd::Zero(3,1); // http://eigen.tuxfamily.org/dox/↩
    AsciiQuickReference.txt
```

The arguments (3,1) specify 3 rows, 1 column. (A vector is simply a special case of a matrix, for which there is either a single row or a single column).

Alternatively, one may specify initial values as arguments to the constructor upon instantiation. To initialize a vector to values of 1:

```
Eigen::VectorXd  ones_vec= Eigen::MatrixXd::Ones(npts,1);
```

In the illustrative example code, a set of points is generated that lie on (or near) a predetermined plane. Eigen methods are invoked to try to discover what was the original plane, using only the data points. This operation is valuable in point-cloud processing, where we may wish to find flat surfaces of interest, e.g. tables, walls, floors, doors, etc.

In the example code, the plane of interest is defined to have a surface normal called `normal_vec`, and the plane is offset from the origin by a distance "dist." A plane has a unique definition of distance from the origin. If one cares about positive vs negative surfaces of a plane, then the distance from the origin can be a signed number, where the offset is measured from origin to plane in the direction of the plane's normal (and thus can result in a negative offset).

To generate some sample data, we construct a pair of vectors perpendicular to the plane's normal vector. We can do so starting with some vector "v1" that is not colinear with the plane normal. In the example code, this vector is generated by rotating `normal_vec` 90 degrees about the z axis, which is accomplished with the following matrix*vector multiply:

```
v1 = Rot_z*normal_vec; //here is how to multiply a matrix times a vector
```

(note: if `normal_vec` is parallel to the z axis, v1 will be equal to `normal_vec`, and the subsequent operations will not work).

For display purposes, Eigen types are nicely formatted for "cout." Matrices are nicely formatted with new lines for each row, e.g. using:

```
cout<<Rot_z<<endl;
```

Rather than use "cout," it is preferable to use `ROS_INFO_STREAM()`, which is more versatile than `ROS_INFO()`. This function outputs the data via network communication, and it is thus visible via `rqt_console` and is loggable. To display `Rot_z` as a formatted matrix with `ROS_INFO_STREAM()`, one can use:

```
ROS_INFO_STREAM(endl<<Rot_z); // start w/ endl, so get a clean first line of data ↩
    display
```

For short column vectors, it is more convenient to display the values on a single line. To do so, one can output the transpose of the vector, e.g. as follows:

```
ROS_INFO_STREAM("v1: "<<v1.transpose()<<endl);
```

Two common vector operations are the dot product and the cross product. From the example code, here are some excerpts:

```
    double dotprod = v1.dot(normal_vec); //using the "dot()" member function
    double dotprod2 = v1.transpose()*normal_vec;// alt: turn v1 into a row vector, ↩
        then multiply times normal_vec
```

and for the cross product, v1 crossed into `normal_vec`:

```
v2 = v1.cross(normal_vec);
```

Note that the result of `v1 x normal_vec` must be mutually orthogonal to both `v1` and `normal_vec`. Since the result, `v2`, is perpendicular to `normal_vec`, it is parallel to the plane under construction.

A second vector in the plane can be computed as:

```
v1 = v2.cross(normal_vec);  // re-use v1; make it the cross product of v2 into ↩
    normal_vec
```

Using the vectors `v1`, `v2` and `normal_vec`, one can define any point in the desired plane as:

```
p = a*v1 + b*v2 + c*normal_vec
```

with the constraint c=dist, but a and b can be any scalar values.

Random points in the plane are generated and stored as column vectors in a 3xN matrix. The matrix is instantiated with the line:

```
Eigen::MatrixXd points_mat(3,npts); //create a matrix, double-precision values, 3 rows↩
     and npts cols
```

The example generates random points within the desired plane as follows:

```
    Eigen::Vector2d rand_vec; //a 2x1 vector
    //generate random points that all lie on plane defined by distance and normal_vec
    for (int ipt = 0;ipt<npts;ipt++) {
        // MatrixXd::Random returns uniform random numbers in the range (-1, 1).
        rand_vec.setRandom(2,1);  // populate 2x1 vector with random values
        //cout<<"rand_vec: "<<rand_vec.transpose()<<endl; //optionally, look at these ↩
            random values
        //construct a random point ON the plane normal to normal_vec at distance "dist↩
            " from origin:
        // a point on the plane is a*x_vec + b*y_vec + c*z_vec, where we may choose
        // x_vec = v1, y_vec = v2 (both of which are parallel to our plane) and z_vec ↩
            is the plane normal
        // choose coefficients a and b to be random numbers, but "c" must be the plane↩
            's distance from the origin, "dist"
        point =  rand_vec(0)*v1 + rand_vec(1)*v2 + dist*normal_vec;
    //save this point as the i'th column in the matrix "points_mat"
    points_mat.col(ipt) = point;
    }
```

Random noise can be added to the (ideal) data on the plane using:

```
// add random noise to these points in range [-0.1,0.1]
Eigen::MatrixXd Noise = Eigen::MatrixXd::Random(3,npts);
```

```
// add two matrices, term by term.  Also, scale all points in a matrix by a scalar: ↩
     Noise*g_noise_gain
points_mat = points_mat + Noise*g_noise_gain;
```

The matrix `points_mat` now contains points, column by column, that approximately lie on a plane distance "dist" from the origin and with surface-normal vector `normal_vec`. This dataset can be used to illustrate the operation of plane fitting. The first step is to compute the centroid of all of the points, which can be computed as follows:

```
    // first compute the centroid of the data:
    // here's a handy way to initialize data to all zeros; more variants exist
    // see http://eigen.tuxfamily.org/dox/AsciiQuickReference.txt
    Eigen::Vector3d centroid = Eigen::MatrixXd::Zero(3,1);

    //add all the points together:
    npts = points_mat.cols(); // number of points = number of columns in matrix; check↩
          the size
    cout<<"matrix has ncols = "<<npts<<endl;
    for (int ipt =0;ipt<npts;ipt++) {
    centroid+= points_mat.col(ipt); //add all the column vectors together
    }
    centroid/=npts; //divide by the number of points to get the centroid
```

The (approximately) planar points are then offset by subtracting the centroid from each point.

```
// subtract this centroid from all points in points_mat:
Eigen::MatrixXd points_offset_mat = points_mat;
for (int ipt =0;ipt<npts;ipt++) {
   points_offset_mat.col(ipt)  = points_offset_mat.col(ipt)-centroid;
}
```

The resulting matrix (three rows and npts columns) can be used to compute a covariance matrix by multiplying this matrix times its transpose, resulting in a 3x3 matrix:

```
Eigen::Matrix3d CoVar;
CoVar = points_offset_mat*(points_offset_mat.transpose());  //3xN matrix times Nx3 ↩
     matrix is 3x3
```

One of the more advanced Eigen options is computation of eigenvectors and associated eigenvalues. This can be invoked with:

```
    // here is a more complex object: a solver for eigenvalues/eigenvectors;
    // we will initialize it with our covariance matrix, which will induce computing ↩
         eval/evec pairs
    Eigen::EigenSolver<Eigen::Matrix3d> es3d(CoVar);
    Eigen::VectorXd evals; //we'll extract the eigenvalues to here
    // in general, the eigenvalues/eigenvectors can be complex numbers
    //however, since our matrix is self-adjoint (symmetric, positive semi-definite), ↩
         we expect
    // real-valued evals/evecs;  we'll need to strip off the real parts of the ↩
         solution
    evals= es3d.eigenvalues().real(); // grab just the real parts
```

The three eigenvalues are all non-negative. The smallest of the eigenvalues corresponds to the eigenvector that is the best-fit approximation to the plane normal. (The direction of smallest variance is perpendicular to the best-fit plane). If the smallest eigenvalue corresponds to index "ivec," then the corresponding eigenvector is:

```
est_plane_normal = es3d.eigenvectors().col(ivec).real();
```

The distance of the plane from the origin can computed as the projection (dot product) of the vector from the origin to the centroid onto the plane normal:

```
    double est_dist = est_plane_normal.dot(centroid);
```

The program **example_eigen_plane_fit.cpp** illustrates a variety of Eigen capabilities, and the specific algorithm illustrated is an efficient and robust technique for fitting planes to data.

Another Eigen object that will be useful is **Eigen::Affine3d**, which has the equivalent capabilities of a coordinate-transform operator. A ROS "tf" transform can be converted to an Eigen Affine object as follows:

```
Eigen::Affine3d transformTFToEigen(const tf::Transform &t) {
    Eigen::Affine3d e;
    // treat the Eigen::Affine as a 4x4 matrix:
    for (int i = 0; i < 3; i++) {
        e.matrix()(i, 3) = t.getOrigin()[i]; //copy the origin from tf to Eigen
        for (int j = 0; j < 3; j++) {
            e.matrix()(i, j) = t.getBasis()[i][j]; //and copy 3x3 rotation matrix
        }
    }
    // Fill in (0,0,0,1) in the last row
    for (int col = 0; col < 3; col++)
        e.matrix()(3, col) = 0;
    e.matrix()(3, 3) = 1;
    return e;
}
```

Subsequently Eigen Affine objects can be multiplied together, or premultiplied times Eigen Vector3d objects to transform points to new coordinates frames, as in the following example code snippet:

```
   //let's say we have a point, "p", as detected in the sensor frame;
    // arbitrarily, initialize this to [1;2;3]
    Eigen::Vector3d p_wrt_sensor(1,2,3);
    //create an affine object that defines the transform between the sensor frame and ↩
         the world frame:
    Eigen::Affine3d affine_sensor_wrt_world;
    //assume "tfTransform" has been filled in by a transform listener for sensor frame↩
          w/rt world frame
    affine_sensor_wrt_world =transformTFToEigen(tfTransform); //convert tf to Eigen::↩
         Affine
    // here's how to convert a sensor point to the world frame:
    Eigen::Vector3d p_wrt_world;
    p_wrt_world = affine_sensor_wrt_world*p_wrt_sensor; //point is now expressed in ↩
         world-frame coordinates

    //we can transform in the opposite direction with the transform inverse:
    Eigen::Vector3d p_back_in_sensor_frame;
    p_back_in_sensor_frame = affine_sensor_wrt_world.inverse()*p_wrt_world;
```

Eigen-style affine transforms will be useful in transforming sensor data to a common frame, as will be described in Part III **??**.

CHAPTER 5

Sensing and Visualization in ROS

CONTENTS

INTRODUCTION
The primary tool for visualization in ROS is "rviz" (see `http://wiki.ros.org/rviz`). With this tool, one can display sensor values–real or emulated–as well as robot pose (as inferred from a robot model and from published robot joint values). One can also superimpose graphics (e.g. to display results from perceptual processing nodes) and provide operator inputs (e.g. via a mouse).

Figure 5.1 shows an example rviz display for the Atlas robot. In this scene, the colored points corresponding to simulated data from a rotating Hokuyo LIDAR (see [**?**]) on a Carnegie-Robotics sensor head (see [**?**]). The LIDAR data has been colorized in the rviz display to indicate z-height of each point. From the rviz display, we can interpret points on the floor (red), points on a table (yellow) and multiple cylinders on the table. There are also colored points on the robot model itself, illustrating that the simulated sensor is aware of the robot model as well, appropriately displaying how the sensor would see the robot's own arm.

Figure 5.2 is an rviz display of data from an actual (physical) Boston-Dynamics Atlas robot in a lab at U. of Hong Kong. In this display, the LIDAR data points have been processed to classify specific entities, and the display colors indicate class membership, including the floor, a wall in front of the robot, a wall to the robot's left, and a door. Points not colorized are not classified in one of the defined categories, including clutter near walls, a gantry, and the ceiling. The LIDAR display from the physical robot is a good approximation to simulated LIDAR from a Gazebo model of the robot. This supports developing perceptual processing software in simulation, which is then applicable to the physical system.

Another example is shown in Fig 5.3. The left scene is an actual Baxter robot in a lab at Case Western Reserve University, with a Microsoft Kinect™attached, as well as a Yale OpenHand [**?**]. The right scene has a similar composition (a cylinder on a table in front

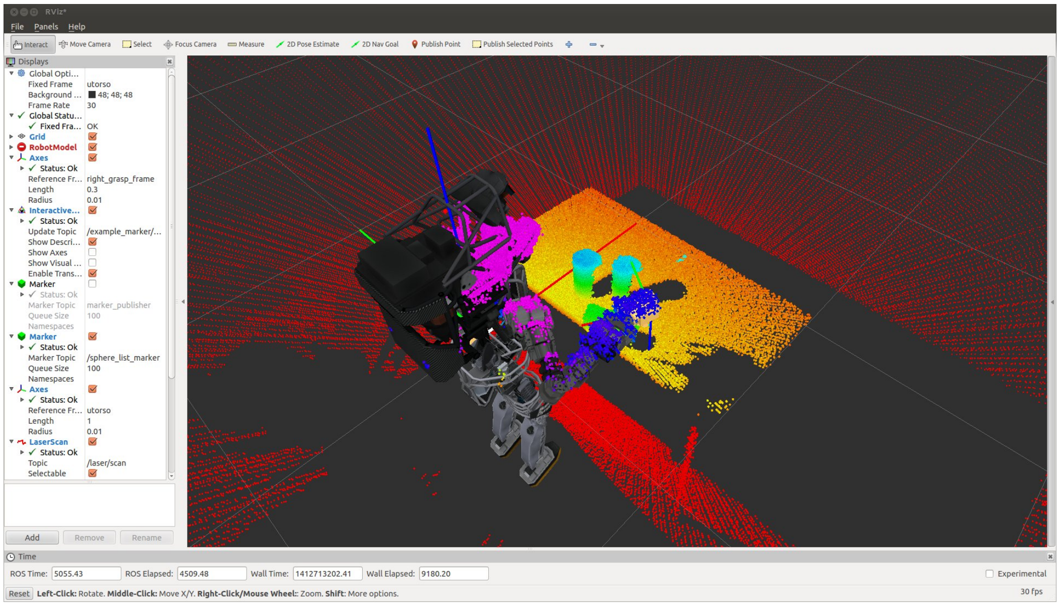


FIGURE 5.1: Rviz view of Atlas robot model with LIDAR sensor display

---

of the robot). It is an rviz view of a model of the Baxter robot and a display of colorized points from a simulated Kinect™sensor mounted on the robot. The points are colorized by z-height.

In the rviz display, a robot model's pose is rendered to be consistent with the robot's current joint angles. The position and orientation of each link of the robot is knowable from published coordinates. If the coordinates are published by a real robot, then the rviz display will be synchronized to render the robot model to mimic the real robot. Alternatively, the source of coordinates could be a Gazebo model–or simply a rosbag playback of previously recorded poses.

Similarly, display of sensor values (e.g. 3-D points from a depth sensor) can be from physical sensors streaming in real time, or from simulated sensors within a Gazebo model, or from playback of pre-recorded data.

As seen in Figures 5.1 and 5.3, data from a real or simulated sensor may include sensing of the robot's own body. Figures 5.1 and 5.3 show that sensory data is appropriately colocated with respective surfaces on the robot model. Such reconciliation of coordinates is necessary to take advantage of sensor-driven behaviors in simulation. Coordinating data from multiple models and sensors is accomplished through coordinate transformations.

Via the parameter server (which holds the kinematic and visual model of our robot under the name `robot_description`), and publications of transforms on "tf," our system is aware of all visual models of all links, as well as all transforms necessary to compute the pose of any link with respect to any other link. With this much information, it is possible for an independent node to access the parameter server, subscribe to the `tf` topic and render all links in a consistent frame of reference. Transformations and rendering are performed in ROS using 'rviz. By running:

```
rosrun rviz rviz
```

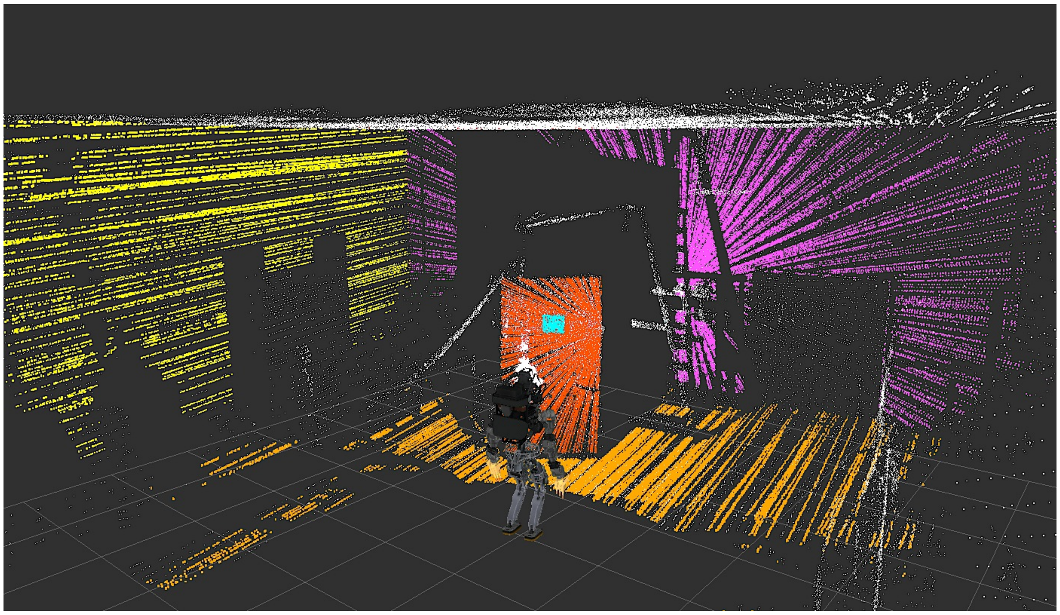


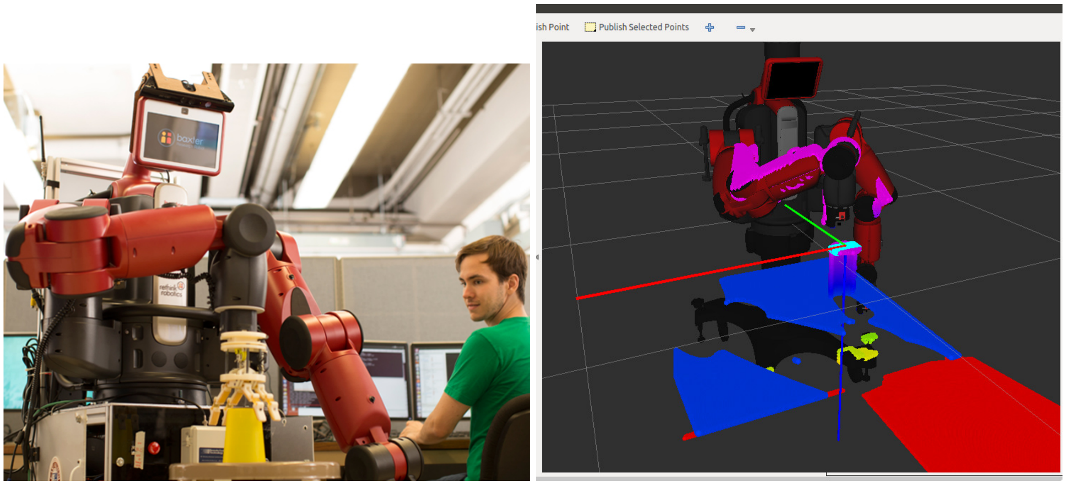
FIGURE 5.2: Rviz view of Atlas robot in lab with interpretation of actual LIDAR data

(and after some interactive configuration), a view of the robot model can be seen, as in Fig 5.4.

The view from rviz is, in fact, less appealing than the view from gazebo. One issue with visualizing the link poses in rviz is that the material color information specified for gazebo display is not compatible with colors specified for rviz (and thus all links appear red by default). This can be corrected by augmenting the model file to include color specifications to be used by gazebo. The rviz view also does not show a ground plane, shading, nor any additional modeling that might be brought into the virtual world. The unique value of rviz will not be apparent until we introduce additional features–notably, display of sensor values.

Our primitive rviz view nonetheless already illustrates multiple important points. First, the display of the robot's configuration does not generally require gazebo. As long as a robot model has been loaded into the paramter server and all of the link transforms are being published, rviz can display a rendering of the robot in a pose consistent with its joint angles. Publication of link transforms could be accomplished, e.g., from playback of a rosbag recording, without the need to run gazebo.

More importantly, the same robot model can be rendered using data from a real robot. Such rendering requires satisfying a few requirements. First, there must be an adequate visual model of the physical robot (as in the examples in the introduction to this chapter), and this model must be loaded onto the parameter server. Second, the real robot must publish its joint displacements on the `joint_states` topic. And third, one must run the `robot_state_publisher` node, which combines URDF information from the parameter server with joint-state information from the robot to compute and publish transform information on the `tf` topic. With these conditions satisfied, rviz can display an rendering of the robot that mimics the actual robot pose, updating continuously as the robot moves. This display can be used, for example, to help debug hardware problems. With the physical robot's motors disabled, one could move the joints manually and observe if the rviz model



(a) Baxter robot approaching an object (Photo by: Russell Lee)

(b) Rviz model of Baxter robot with Kinect sensor

FIGURE 5.3: Physical and simulated Baxter robots. Rviz view (b) displays points from simulated Kinect sensor with points colorized by z-height

---

follows along. If so, this test would confirm that the corresponding joint sensors are alive and reporting correction joint-displacement values.

A second valuable debugging capability of rviz is to visualize playback of records (rosbags) of tests on a real robot. Such replay can be useful for identifying clues for debugging, e.g. did a robot fall due to instability of balance, or did it trip? Did the robot controller exhibit oscillations at different poses? Did joint limits or inertial effects result in undesirable motion?

Rviz display can also be useful in understanding a robot model. In Fig 5.4, "display items" include the robot model, as well as 4 axis frames: `base_link`, `left_wheel`, `link2` and `left_casterwheel`. Coordinate frames displays are color coded: red for x, green for y and blue for z. These displays should reconcile with our expectations. The base frame has its x axis forward, its y axis to the left, and its z axis up. Its origin is at ground level, immediately below the point between the drive wheels. The left-wheel frame has its z axis pointing to the left, coincident with the wheel axle. The y and x axes are orthogonal to the z axis, and they rotate about the wheel z axis.

The left caster wheel frame also has its z axis coincident with the wheel's axle, though the caster is rotated in this scene so the wheel axle is not parallel to the base-frame y axis. The link2 frame shows that the joint1 axis is coincident with the y axis of the link2 frame.

This display shows that rviz has full information of all poses of all links of the robot.

The true value of rviz will become apparent with the introduction of sensor displays, graphical overlays and operator interaction.

Before diving deeper into rviz, a subtle point is worth mentioning. For rviz to display all links, it must have access to all transforms. All joint angles are known to gazebo, since gazebo performs runs the physics simulation. The example models have included a joint-state publisher, with which gazebo makes all joint angles available to ROS on the `joint_states` topic. On a real robot that corresponds to our mobot model, motion of the caster wheels would be similar to that of gazebo simulation. However, it would be unusual to have joint sensors on passive joints. Thus, the real robot would be unable to publish joint

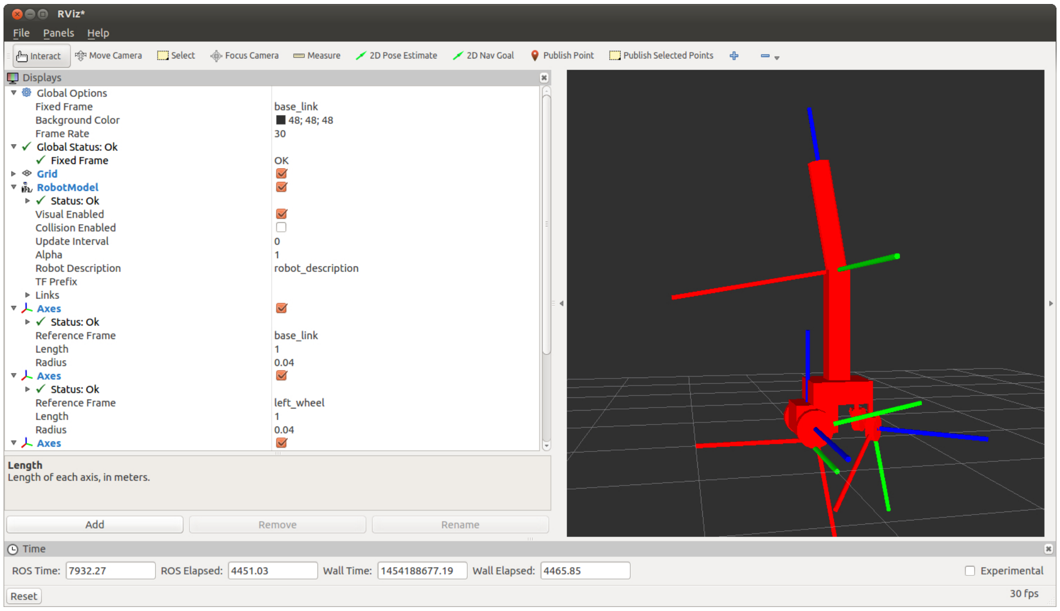


FIGURE 5.4: Rviz view of simple mobile robot with 1-DOF arm

values for the caster brackets and caster wheels. Consequently, the robot state publisher would be unable to compute the corresponding transforms, and rviz would be unable to render the casters and caster wheels. A work-around for this disparity is to run a node that publishes 0 for all 4 of the caster joints. In rviz, the casters and caster wheels would appear to be static, but at least rviz would be able to display their models without complaint.

A second subtle issue with rviz concerns pose of the robot in the world. Within gazebo, a world frame is defined, and all models (and links within models) have poses that are known to gazebo with respect to the world (a necessary requirement to perform computation of dynamics in an inertial frame). However, the pose of the robot relative to the world frame is not published to ROS. Consequently, rviz cannot know where the robot is in the world. In order to produce a consistent rendering, a (non-world) reference frame must be declared. In rviz (within the "displays" panel, "global options" item), the "Fixed Frame" can be set. When this is set to `base_link`, all rendering will be with respect to the robot's base frame. As a consequence, when sensor values are displayed in rviz, it will appear that the robot is stationary and the world moves towards the robot–a viewpoint as though the observer were riding on the robot.

Rendering the robot in a world frame requires additional steps, including defining a map of the world and running localization routines that help establish the robot's position and orientation with respect to the map. Localization for mobile robots will be introduced in Chapter **??**.

5.1 MARKERS AND INTERACTIVE MARKERS IN RVIZ

It is frequently useful to display computed graphical overlays in a scene. These can be used, e.g., to help highlight an object of interest, to show a robot's focus of attention, to display a perception module's belief in the pose of an object model with respect to sensory data, or to indicate kinematically reachable regions. A means to provide graphical overlays in rviz is publishing "markers." A more sophisticated marker type is the "Interactive Marker", which allows the operator to move around a marker in rviz with a mouse (or other input device) and publish the resulting coordinates in ROS.

5.1.1 Markers in rviz

One can display markers in the rviz view by publishing to appropriate topics. An example is in the accompanying repository, package `example_rviz_marker`. The source code, `example_rviz_marker.cpp`, is shown in Listing 5.1.

Listing 5.1: C++ code to publish a grid of markers for rviz display

```cpp
#include <ros/ros.h>
#include <visualization_msgs/Marker.h> // need this for publishing markers
#include <geometry_msgs/Point.h> //data type used for markers
#include <string.h>
#include <stdio.h>
#include <example_rviz_marker/SimpleFloatSrvMsg.h> //a custom message type defined in this package
using namespace std;

//set these two values by service callback, make available to "main"
double g_z_height = 0.0;
bool g_trigger = true;

//a service to prompt a new display computation.
// E.g., to construct a plane at height z=1.0, trigger with:
// rosservice call rviz_marker_svc 1.0

bool displaySvcCB(example_rviz_marker::SimpleFloatSrvMsgRequest& request,
    example_rviz_marker::SimpleFloatSrvMsgResponse& response) {
    g_z_height = request.request_float32;
    ROS_INFO("example_rviz_marker: received request for height %f", g_z_height);
    g_trigger = true; // inform "main" a new computation is desired
    response.resp=true;
    return true;
}

int main(int argc, char **argv) {
    ros::init(argc, argv, "example_rviz_marker");
    ros::NodeHandle nh;
    ros::Publisher vis_pub = nh.advertise<visualization_msgs::Marker>("example_marker_topic", 0);
    visualization_msgs::Marker marker; // instantiate a marker object
    geometry_msgs::Point point; // points will be used to specify where the markers go

    //set up a service to compute marker locations on request
    ros::ServiceServer service = nh.advertiseService("rviz_marker_svc", displaySvcCB);

    marker.header.frame_id = "/world"; // reference frame for marker coords
    marker.header.stamp = ros::Time();
    marker.ns = "my_namespace";
    marker.id = 0;
    // use SPHERE if you only want a single marker
    // use SPHERE_LIST for a group of markers
    marker.type = visualization_msgs::Marker::SPHERE_LIST; //SPHERE;
    marker.action = visualization_msgs::Marker::ADD;
    // if just using a single marker, specify the coordinates here, like this:

    //marker.pose.position.x = 0.4;
    //marker.pose.position.y = -0.4;
```

```
    //marker.pose.position.z = 0;
    //ROS_INFO("x,y,z = %f %f, %f",marker.pose.position.x,marker.pose.position.y,
    //         marker.pose.position.z);
    // otherwise, for a list of markers, put their coordinates in the "points" array, as below

    //whether a single marker or list of markers, need to specify marker properties
    // these will all be the same for SPHERE_LIST
    marker.pose.orientation.x = 0.0;
    marker.pose.orientation.y = 0.0;
    marker.pose.orientation.z = 0.0;
    marker.pose.orientation.w = 1.0;
    marker.scale.x = 0.02;
    marker.scale.y = 0.02;
    marker.scale.z = 0.02;
    marker.color.a = 1.0;
    marker.color.r = 1.0;
    marker.color.g = 0.0;
    marker.color.b = 0.0;

    double z_des;

    // build a wall of markers; set range and resolution
    double x_min = -1.0;
    double x_max = 1.0;
    double y_min = -1.0;
    double y_max = 1.0;
    double dx_des = 0.1;
    double dy_des = 0.1;

    while (ros::ok()) {
        if (g_trigger) {  // did service get request for a new computation?
            g_trigger = false; //reset the trigger from service
            z_des = g_z_height; //use z-value from service callback
            ROS_INFO("constructing plane of markers at height %f",z_des);
            marker.header.stamp = ros::Time();
            marker.points.clear(); // clear out this vector

            for (double x_des = x_min; x_des < x_max; x_des += dx_des) {
                for (double y_des = y_min; y_des < y_max; y_des += dy_des) {
                        point.x = x_des;
                        point.y = y_des;
                        point.z = z_des;
                        marker.points.push_back(point);
                }
            }
        }
     ros::Duration(0.1).sleep();
    //ROS_INFO("publishing...");
    vis_pub.publish(marker);
    ros::spinOnce();
    }
    return 0;
}
```

The source code of this node defines a service, called `rviz_marker_svc`. This service uses the a service message, `SimpleFloatSrvMsg`, that is defined in the `example_rviz_marker` package, which expects the client to provide a single floating-point value. This service can be invoked manually, e.g., with the terminal command:

```
rosservice call rviz_marker_svc 1.0
```

When this service is invoked, the `example_rviz_marker` node will compute a grid of points within a horizontal plane at height 1.0 (as specified in the above example command).

This package lists dependency on `visualization_msgs` and `geometry_msgs` in the `package.xml` file. Correspondingly, the source code includes the headers:

```
#include <visualization_msgs/Marker.h>
```

and

```
#include <geometry_msgs/Point.h>
```

Within "main()" of example_rviz_marker.cpp, line 29, a publisher object is instantiated using the message type:

```
ros::Publisher vis_pub = nh.advertise<visualization_msgs::Marker>("↩
    example_marker_topic", 0);
```

which publishes to the chosen topic example_marker_topic.

The message type visualization_msgs::Marker is fairly complex, as can be seen by entering:

```
rosmsg show visualization_msgs/Marker
```

This message type contains 15 fields, many of which contain sub-fields. Additional details on definition and use of this message type can be found at: http://wiki.ros.org/visualization_msgs and http://wiki.ros.org/rviz/DisplayTypes/Marker. The example code, lines 36-65, populates the fields: header (including frame_id and time stamp), type, action, pose, scale, color and the vector of points (with x,y,z coordinates). In the main loop of this example node, the x,y,z coordinates of a list of points is populated and published, with the chosen shape, color and size held constant.

To run the example node, with roscore running, enter:

```
rosrun example_rviz_mark example_rviz_marker
```

A rostopic list shows that there is a topic example_marker_topic, and rostopic info example_marker_topic shows that the node example_rviz_marker publishes messages of type visualization_msgs/Marker to this topic.

Upon start-up, this node will populate points (markers) to be displayed at elevation zero. Rviz will be able to display these markers graphicallyafter adding the display type and topic name. To do so, start up rviz:

```
rosrun rviz rviz
```

Note that it is not necessary to have gazebo running.

In the rviz display, one must add an item "Marker" to the rviz "displays" list, as shown in Fig 5.5, by clicking "Add" and choosing "Marker" from the pop-up options. After clicking "OK," the display list will include the item "Marker." Clicking on this item to expand it, one has the option of editing the topic to which this display should subscribe. As shown in Fig 5.6, the topic should be edited by typing in the text example_marker_topic (or selecting it from among drop-down list options), which is the topic name chosen in our source code for example_rviz_marker. At this point, rviz starts receiving messages from our example_rviz_marker node, resulting in a display of red spheres within a planar patch at height zero about the origin. The markers will appear at height 1.0 if the example_rviz_marker node responds to a service request of:

```
rosservice call rviz_marker_svc 1.0
```

as shown in Fig 5.7. Ordinarily, the service client would originate from another node–but manually stimulating the service from the command line is useful for incremental testing.

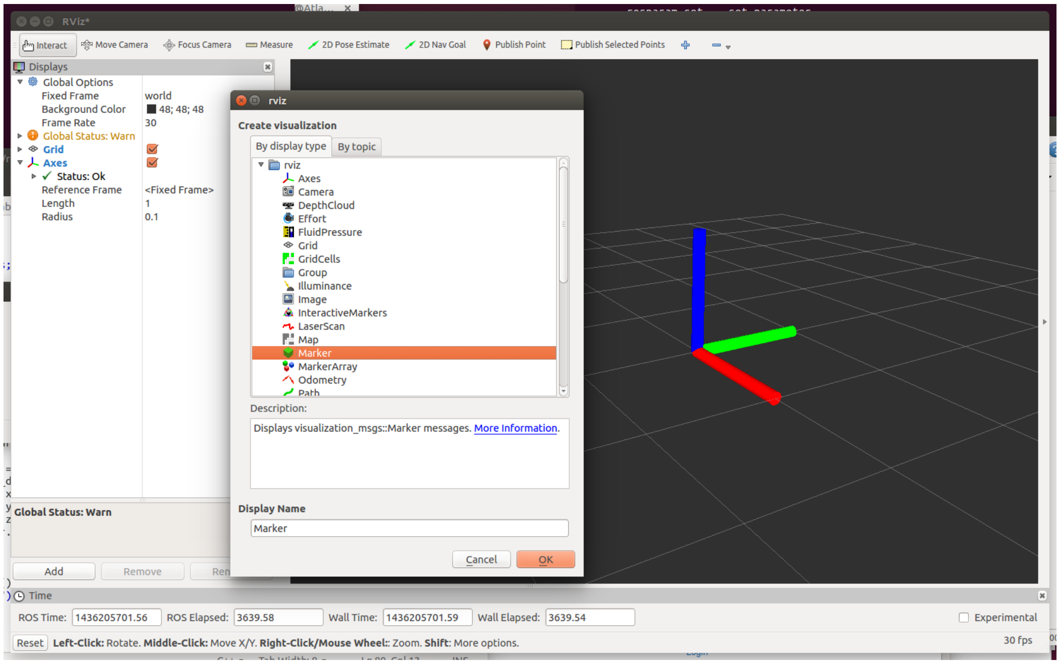


FIGURE 5.5: Adding a marker display in rviz

5.1.2 Interactive Markers in rviz

A more complex marker type is the "Interactive Marker." (see `http://wiki.ros.org/rviz/Tutorials/InteractiveMarkers:GettingStarted`). Example code for using interactive markers is in the accompanying repository package `example_interactive_marker`. The node `example_interactive_marker` is the executable name compiled from the source code in `IM_6DOF.cpp`. The (lengthy) source code is shown in Listing 5.2.

The interactive-marker code relies on the `interactive_markers` package, and thus it includes (line 5):

```
#include <interactive_markers/interactive_marker_server.h>
```

A custom service message, `ImNodeSvcMsg` is defined in the `example_interactive_marker` package. This message includes a `geometry_msgs/PoseStamped` both for the request and the response. The service message is used by a service, `IM6DofSvc`, implemented in the callback function `IM6DofSvcCB` (lines 21-56). This callback function has different behaviors depending on the command mode in the service message. If the command mode is `IM_GET_CURRENT_MARKER_POSE`, then the current marker pose is returned in the service response (lines 25-32). If the command mode is `IM_SET_NEW_MARKER_POSE`, then an object of type `geometry_msgs::PoseStamped`, `poseStamped_IM_desired` (line 36) is populated with data from the service request `poseStamped_IM_desired` field. The interactive marker is then moved programmatically by lines 50-51:

```
        g_IM_server->setPose("des_hand_pose",poseStamped_IM_desired.pose);
        g_IM_server->applyChanges();
```

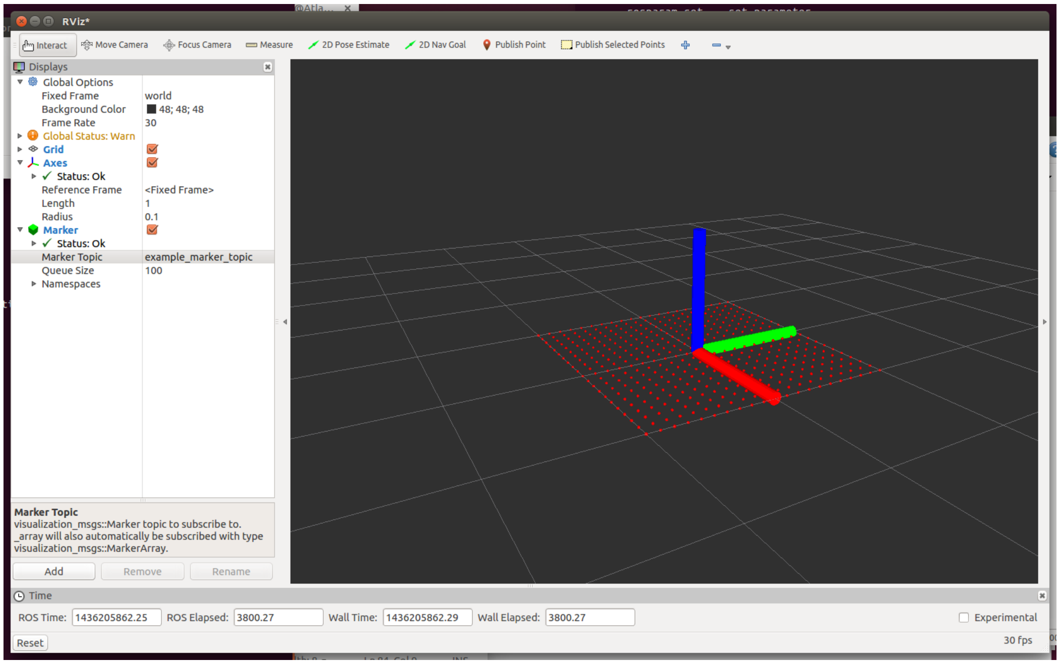


FIGURE 5.6: Markers displayed in rviz from topic `example_marker_topic`

Invoking motion of the interactive marker uses an interactive marker server, defined on lines 73-74:

```
interactive_markers::InteractiveMarkerServer server("rt_hand_marker");
g_IM_server = &server;
```

A (global) pointer to the interactive-marker server is defined so that callback functions can access the server (as in lines 50-51). The server is associated with the callback function `processFeedback` in "main()" on line 254:

```
g_IM_server->insert(int_marker, &processFeedback);
```

A limitation of interactive markers is that one cannot query them for their current pose. Instead, one must monitor publications when the marker is moved, then remember these values for future use. Marker-pose memory is implemented via the callback function `processFeedback` (lines 59-67). When the marker is moved via its graphical handles, the new pose is received by this function, and these values are saved to global variables. Subsequently, a request to the service `IM6DofSvc` can be sent at any time to get the most recently received pose of the interactive marker.

The bulk of the `IM_6DOF.cpp` code (lines 84-171) describes the type, size and color of the 6 components of the interactive marker and associates type and direction of the 6 interactive motion controls (lines 173-241). After these initializations, the main program goes into a spin (line 262), and all further interactions are handled by the service and the callback function.

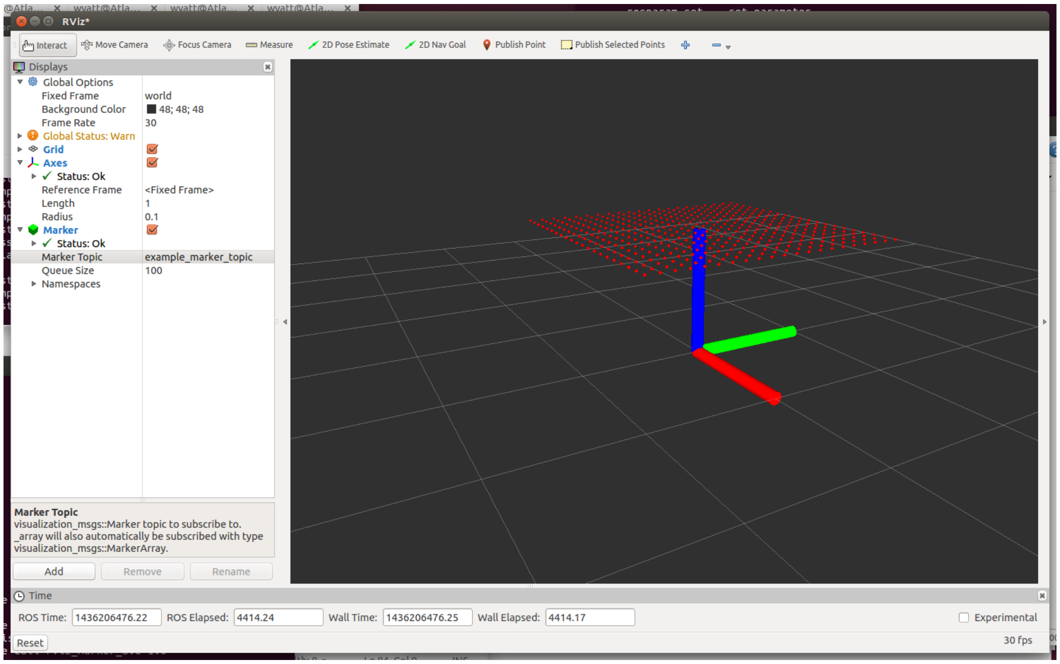


FIGURE 5.7: Markers at heigth 1.0 after rosservice call

Listing 5.2: C++ for a 6-DOF Interactive Marker

```
// IM_6DOF.cpp
// Wyatt Newman, based on ROS tutorial 4.2 on Interactive Markers
#include <ros/ros.h>
#include <iostream>
#include <interactive_markers/interactive_marker_server.h>
#include <geometry_msgs/Point.h>
#include <example_interactive_marker/ImNodeSvcMsg.h>

const int IM_GET_CURRENT_MARKER_POSE=0;
const int IM_SET_NEW_MARKER_POSE= 1;

geometry_msgs::Point g_current_point;
geometry_msgs::Quaternion g_current_quaternion;
ros::Time g_marker_time;

interactive_markers::InteractiveMarkerServer *g_IM_server; //("rt_hand_marker");
visualization_msgs::InteractiveMarkerFeedback *g_IM_feedback;

//service: return pose of marker from above globals;
// depending on mode, move IM programmatically,
bool IM6DofSvcCB(example_interactive_marker::ImNodeSvcMsgRequest& request, ↩
    example_interactive_marker::ImNodeSvcMsgResponse& response) {
    //if busy, refuse new requests;

    // for a simple status query, handle it now;
    if (request.cmd_mode == IM_GET_CURRENT_MARKER_POSE) {
        ROS_INFO("IM6DofSvcCB: rcvd request for query--GET_CURRENT_MARKER_POSE");
        response.poseStamped_IM_current.header.stamp = g_marker_time;
        response.poseStamped_IM_current.header.frame_id = "world";
        response.poseStamped_IM_current.pose.position = g_current_point;
        response.poseStamped_IM_current.pose.orientation = g_current_quaternion;
        return true;
    }

    //command to move the marker to specified pose:
     if (request.cmd_mode == IM_SET_NEW_MARKER_POSE) {
```

```
         geometry_msgs::PoseStamped poseStamped_IM_desired;
         ROS_INFO("IM6DofSvcCB: rcvd request for action--SET_NEW_MARKER_POSE");
        g_current_point =  request.poseStamped_IM_desired.pose.position;
        g_current_quaternion = request.poseStamped_IM_desired.pose.orientation;
        g_marker_time = ros::Time::now();
        poseStamped_IM_desired = request.poseStamped_IM_desired;
        poseStamped_IM_desired.header.stamp = g_marker_time;
        response.poseStamped_IM_current = poseStamped_IM_desired;
        //g_IM_feedback->pose = poseStamped_IM_desired.pose;

        response.poseStamped_IM_current.header.stamp = g_marker_time;
        response.poseStamped_IM_current.header.frame_id = "torso";
        response.poseStamped_IM_current.pose.position = g_current_point;
        response.poseStamped_IM_current.pose.orientation = g_current_quaternion;
        g_IM_server->setPose("des_hand_pose",poseStamped_IM_desired.pose); //g_IM_feedback->marker_name,poseStamped_IM_desired.pose);
        g_IM_server->applyChanges();
        return true;
    }
    ROS_WARN("IM6DofSvcCB: case not recognized");
    return false;
}


void processFeedback(
        const visualization_msgs::InteractiveMarkerFeedbackConstPtr &feedback) {
    ROS_INFO_STREAM(feedback->marker_name << " is now at "
            << feedback->pose.position.x << ", " << feedback->pose.position.y
            << ", " << feedback->pose.position.z);
    g_current_quaternion = feedback->pose.orientation;
    g_current_point =feedback->pose.position;
    g_marker_time = ros::Time::now();
}

int main(int argc, char** argv) {
    ros::init(argc, argv, "simple_marker"); // this will be the node name;
    ros::NodeHandle nh; //standard ros node handle
    // create an interactive marker server on the topic namespace simple_marker
    interactive_markers::InteractiveMarkerServer server("rt_hand_marker");
    g_IM_server = &server;
    ros::ServiceServer IM_6dof_interface_service = nh.advertiseService("IM6DofSvc",&IM6DofSvcCB);
    // look for resulting pose messages on the topic: /rt_hand_marker/feedback,
    // which publishes a message of type visualization_msgs/InteractiveMarkerFeedback, which
    // includes a full "pose" of the marker.
    // Coordinates of the pose are with respect to the named frame

    // create an interactive marker for our server
    visualization_msgs::InteractiveMarker int_marker;

    int_marker.header.frame_id = "world"; //base_link"; ///world"; // the reference frame for pose coordinates
    int_marker.name = "des_hand_pose"; //name the marker
    int_marker.description = "Interactive Marker";

    geometry_msgs::Point temp_point_start;
    /** specify/push-in the origin for this marker */
    temp_point_start.x = 0.5;
    temp_point_start.y = -0.5;
    temp_point_start.z = 0.2;
    g_current_point.x =temp_point_start.x;
    g_current_point.y = temp_point_start.y;
    g_current_point.z =      temp_point_start.z;

    // create an arrow marker; do this 3 times to create a triad (frame)
    visualization_msgs::Marker arrow_marker_x; //this one for the x axis
    geometry_msgs::Point temp_point;

    arrow_marker_x.type = visualization_msgs::Marker::ARROW; //ROS example was a CUBE; changed to ARROW
    // specify/push-in the origin point for the arrow
    temp_point.x = temp_point.y = temp_point.z = 0;
    arrow_marker_x.points.push_back(temp_point);
    // Specify and push in the end point for the arrow
    temp_point = temp_point_start;
```

```
    temp_point.x = 0.2; // arrow is this long in x direction
    temp_point.y = 0.0;
    temp_point.z = 0.0;
    arrow_marker_x.points.push_back(temp_point);

    // make the arrow very thin
    arrow_marker_x.scale.x = 0.01;
    arrow_marker_x.scale.y = 0.01;
    arrow_marker_x.scale.z = 0.01;

    arrow_marker_x.color.r = 1.0; // red, for the x axis
    arrow_marker_x.color.g = 0.0;
    arrow_marker_x.color.b = 0.0;
    arrow_marker_x.color.a = 1.0;

    // do this again for the y axis:
    visualization_msgs::Marker arrow_marker_y;
    arrow_marker_y.type = visualization_msgs::Marker::ARROW;
    // Push in the origin point for the arrow
    temp_point.x = temp_point.y = temp_point.z = 0;
    arrow_marker_y.points.push_back(temp_point);
    // Push in the end point for the arrow
    temp_point.x = 0.0;
    temp_point.y = 0.2; // points in the y direction
    temp_point.z = 0.0;
    arrow_marker_y.points.push_back(temp_point);

    arrow_marker_y.scale.x = 0.01;
    arrow_marker_y.scale.y = 0.01;
    arrow_marker_y.scale.z = 0.01;

    arrow_marker_y.color.r = 0.0;
    arrow_marker_y.color.g = 1.0; // color it green, for y axis
    arrow_marker_y.color.b = 0.0;
    arrow_marker_y.color.a = 1.0;

    // now the z axis
    visualization_msgs::Marker arrow_marker_z;
    arrow_marker_z.type = visualization_msgs::Marker::ARROW; //CUBE;
    // Push in the origin point for the arrow
    temp_point.x = temp_point.y = temp_point.z = 0;
    arrow_marker_z.points.push_back(temp_point);
   // Push in the end point for the arrow
    temp_point.x = 0.0;
    temp_point.y = 0.0;
    temp_point.z = 0.2;
    arrow_marker_z.points.push_back(temp_point);

    arrow_marker_z.scale.x = 0.01;
    arrow_marker_z.scale.y = 0.01;
    arrow_marker_z.scale.z = 0.01;

    arrow_marker_z.color.r = 0.0;
    arrow_marker_z.color.g = 0.0;
    arrow_marker_z.color.b = 1.0;
    arrow_marker_z.color.a = 1.0;
/**/
    // create a control that contains the markers
    visualization_msgs::InteractiveMarkerControl IM_control;
    IM_control.always_visible = true;
    //IM_control.markers.push_back(sphere_marker);

    IM_control.markers.push_back(arrow_marker_x);
    IM_control.markers.push_back(arrow_marker_y);
    IM_control.markers.push_back(arrow_marker_z);

    // add the control to the interactive marker
    int_marker.controls.push_back(IM_control);

    // create a control that will move the marker
    // this control does not contain any markers,
    // which will cause RViz to insert two arrows
    visualization_msgs::InteractiveMarkerControl translate_control_x;
    translate_control_x.name = "move_x";
    translate_control_x.interaction_mode =
        visualization_msgs::InteractiveMarkerControl::MOVE_AXIS;
```

```

    /** Create the Z-Axis Control*/
    visualization_msgs::InteractiveMarkerControl translate_control_z;
    translate_control_z.name = "move_z";
    translate_control_z.interaction_mode =
            visualization_msgs::InteractiveMarkerControl::MOVE_AXIS;
    translate_control_z.orientation.x = 0; //point this in the y direction
    translate_control_z.orientation.y = 1;
    translate_control_z.orientation.z = 0;
    translate_control_z.orientation.w = 1;

    /** Create the Y-Axis Control*/
    visualization_msgs::InteractiveMarkerControl translate_control_y;
    translate_control_y.name = "move_y";
    translate_control_y.interaction_mode =
        visualization_msgs::InteractiveMarkerControl::MOVE_AXIS;
    translate_control_y.orientation.x = 0; //point this in the y direction
    translate_control_y.orientation.y = 0;
    translate_control_y.orientation.z = 1;
    translate_control_y.orientation.w = 1;

    // add x-rotation control
  /**/
    visualization_msgs::InteractiveMarkerControl rotx_control;
    rotx_control.always_visible = true;
    rotx_control.interaction_mode = visualization_msgs::InteractiveMarkerControl::ROTATE_AXIS;
    rotx_control.orientation.x = 1;
    rotx_control.orientation.y = 0;
    rotx_control.orientation.z = 0;
    rotx_control.orientation.w = 1;
    rotx_control.name = "rot_x";

    // add z-rotation control
    visualization_msgs::InteractiveMarkerControl rotz_control;
    rotz_control.always_visible = true;
    rotz_control.interaction_mode = visualization_msgs::InteractiveMarkerControl::ROTATE_AXIS;
    rotz_control.orientation.x = 0;
    rotz_control.orientation.y = 1;
    rotz_control.orientation.z = 0;
    rotz_control.orientation.w = 1;
    rotz_control.name = "rot_z";

    // add y-rotation control
    visualization_msgs::InteractiveMarkerControl roty_control;
    roty_control.always_visible = true;
    roty_control.interaction_mode = visualization_msgs::InteractiveMarkerControl::ROTATE_AXIS;
    roty_control.orientation.x = 0;
    roty_control.orientation.y = 0;
    roty_control.orientation.z = 1;
    roty_control.orientation.w = 1;
    roty_control.name = "rot_y";
/**/
    // add the controls to the interactive marker
    int_marker.controls.push_back(translate_control_x);
    int_marker.controls.push_back(translate_control_y);
    int_marker.controls.push_back(translate_control_z);
    int_marker.controls.push_back(rotx_control);
    int_marker.controls.push_back(rotz_control);
    int_marker.controls.push_back(roty_control);

    /** Scale Down: this makes all of the arrows/disks for the user controls smaller than the default size */
    int_marker.scale = 0.2;

    //let's pre-position the marker, else it will show up at the frame origin by default
    int_marker.pose.position.x = temp_point_start.x;
    int_marker.pose.position.y = temp_point_start.y;
    int_marker.pose.position.z = temp_point_start.z;

    // add the interactive marker to our collection &
    // tell the server to call processFeedback() when feedback arrives for it
    //server.insert(int_marker, &processFeedback);
```

```
    g_IM_server->insert(int_marker, &processFeedback);
    // 'commit' changes and send to all clients
    //server.applyChanges();
    g_IM_server->applyChanges();


    // start the ROS main loop
    ROS_INFO("going into spin...");
    ros::spin();
}
```

To run the example interactive marker, with roscore and rviz running (optionally, additional nodes, such as `example_rviz_marker`, displaying a robot model, etc.), start this node with:

```
rosrun  example_interactive_marker interactive_marker_node
```

In rviz, one must add a display and enter the appropriate topic name to visualize the interactive marker. To do so, in rviz click "Add" and choose the item "InteractiveMarkers", as shown in Fig 5.8.

The new InteractiveMarker item is expanded, and the topic `rt_hand_marker/update` is selected from the drop-down menu. An interactive marker then appears, as in Fig 5.9.

This display has 9 interactive handles for moving the marker +/- x, y, z, and +/- rotation about x, y and z. In rviz, if one hovers the mouse over one of these controls, the color of the handle will bolden, and by clicking and dragging, the marker will change its displacement or orientation. As the marker is moved interactively, its "pose" (6-D position and orientation) in space changes. The new values are published, invoking a callback response from the function:

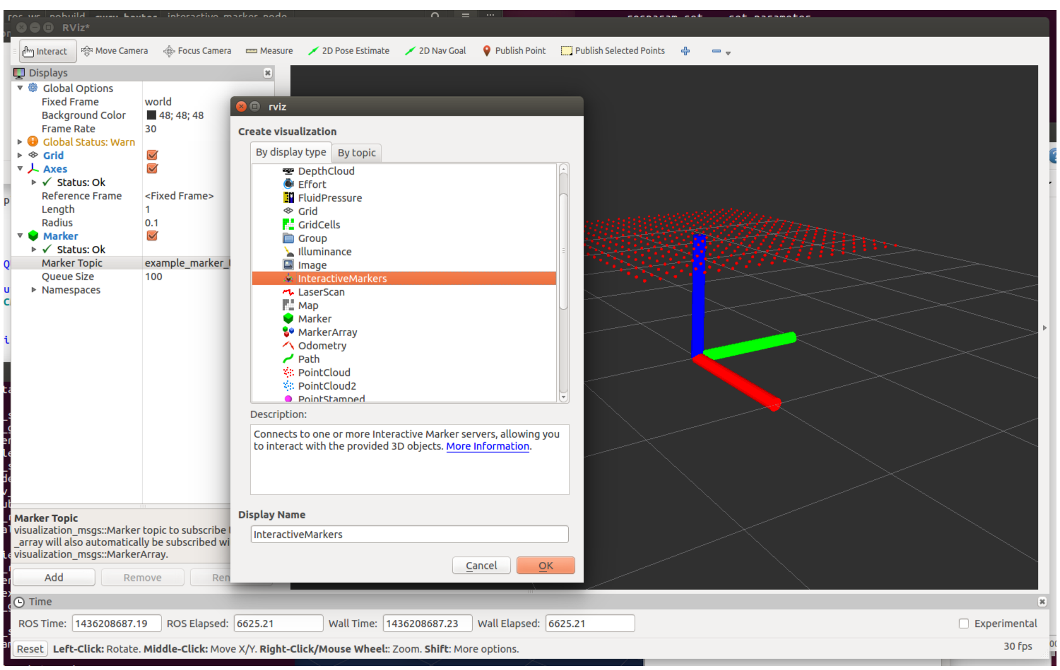


FIGURE 5.8: Adding an Interactive Marker to rviz display

Interactive markers can be used to input full 6-D poses of interest. Such inputs can be

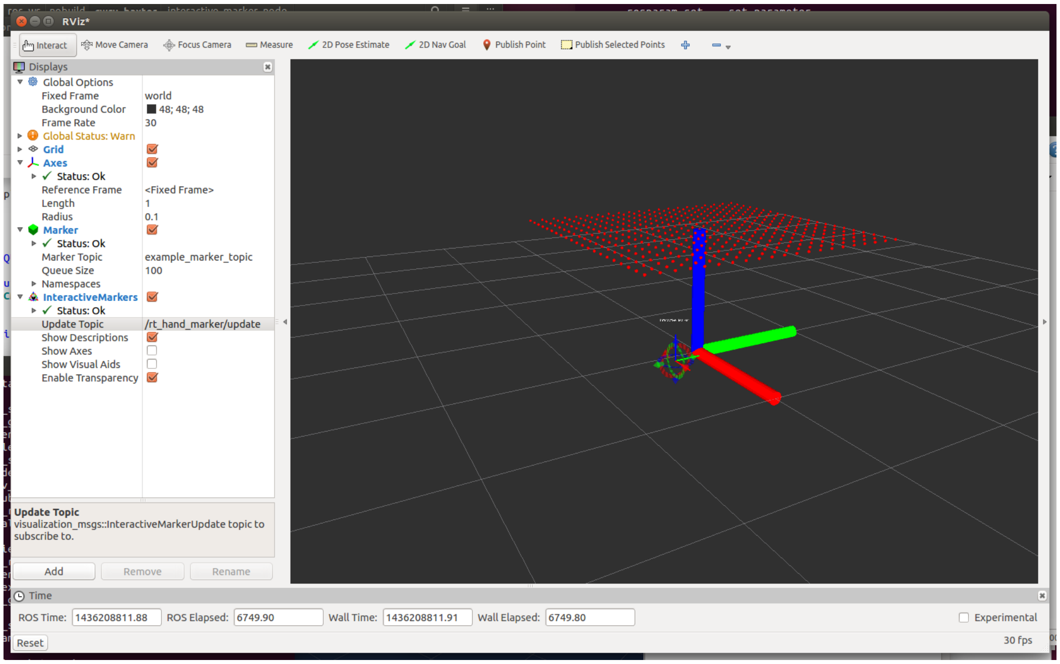


FIGURE 5.9: Display of an Interactive Marker in rviz

used, e.g. to specify a desired hand pose or to direct attention to an object of interest. Moving markers under program control can be used to illustrate computed/proposed poses to achieve or to indicate computed interpretations of poses of objects of interest.

Markers in rviz are particularly useful when then can be placed with respect to sensory data. We next consider how sensor data can be displayed in rviz.

5.2 DISPLAYING SENSOR VALUES IN RVIZ

One of the most valuable aspects of Gazebo (together with rviz) is the ability to perform realistic emulation of common sensors. With a virtual world, simulated sensors provide corresponding data that can be used for off-line program development of sensor-driven behaviors. This section will show how a few common sensors that can be incorporated in gazebo and interpreted in rviz. It should be noted that the rviz display of emulated sensors is performed the same way as rviz display of real sensors. Rviz visualization is unaware of the origin of the sensor signals, whether physical, rosbag logs being played back, or simulated sensors.

5.2.1 Simulating and displaying LIDAR

One of the most common sensors used in robots (including autonomous vehicles) is LIDAR (LIght Detection And Ranging, or LIght-RADAR). LIDAR sensors send out very brief pulses of laser light and measure the time of flight of reflected light to compute distance. Most commonly, a LIDAR uses a spinning mirror, resulting in samples of the environment in a single plane. Popular manufacturers include Sick [**?**] and Hokuyo [**?**]. Data from these devices is obtained at regular angular intervals (typically between 1 and 0.25 deg). The output is streamed in a format consisting of a list of radii, one list per revolution of the LIDAR's mirror. For example, a Sick LMS200 LIDAR can provide 181 radial distances sampled over a 180-deg semicircle with a 1-degree sample resolution repeated at 75Hz. With known start angle, end angle and angular resolution, it is only necessary to transmit a list of radii, and these distances can be inferred to associate with a corresponding angle, thus providing samples in polar coordinates. LIDAR sensors also are used to get 3-D panoramic data. In the DARPA Urban Challenge [**?**] and subsequently with Google cars [**?**], 3-D Velodyne sensors [**?**] were used, essentially equivalent to 64 LIDARS in parallel. Lower-cost LIDARs have been used to get 3-D data by adding mechanism to change the LIDAR's viewpoint, either with a wobbler (oscillating the mirror's spin axis) or by spinning the LIDAR about an axis orthogonal to the mirror's spin axis, as was used on the sensor head [**?**] of the Boston-Dynamics Atlas robots in the DARPA Robotics Challenge [**?**].

The simple two-dimensional robot simulator (STDR), introduced in Section 3.1 illustrated the concept of LIDAR graphically with red-colored rays emanating from an abstracted mobile robot. As shown in Fig 3.1, each LIDAR line originates from the sensor and "pings" a point in the environment. By knowing the length of this line (as deduced from time of flight) and the angle of this line, a single point in the environment is sampled. With careful attention to transforms, the 3-D vector of the line of sight of this pulse is known in some reference frame, from which one can compute corresponding 3-D coordinates of a point in the environment with respect to this reference frame.

Messages from LIDAR can use the ROS message type: `sensor_msgs/LaserScan`. The example code in package `lidar_alarm` in the accompanying repository includes the file `lidar_alarm.cpp`. This code shows how to interpret messages of type `sensor_msgs/LaserScan`. Although these LIDAR messages are generated by STDR, the format is identical for LIDAR sources that are from physical sensors or from gazebo emulation of a LIDAR.

To create simulated LIDAR data, we need to augment our robot model to bring in a gazebo plug-in for LIDAR emulation. This is illustrated by the model `mobot_w_lidar.xacro` in the package `mobot_urdf`. This model is identical to the model `mobot_w_jnt_pub.xacro` described in Section **??** (and is thus not repeated here), except for the following two inserted blocks. First, a visual, collision, and dynamic model of a LIDAR (a simple box) is defined as a new link, and this link is attached to the robot with a static joint, as per Listing 5.3.

Listing 5.3: Link and Joint modeling for adding LIDAR to mobot model

```
1  <!-- add a simulated lidar, including visual, collision and inertial properties, and physics simulation-->
2    <link name="lidar_link">
3        <collision>
4            <origin xyz="0 0 0" rpy="0 0 0"/>
5            <geometry>
6                <!-- coarse LIDAR model; a simple box -->
7                <box size="0.2 0.2 0.2"/>
8            </geometry>
9        </collision>
10
11       <visual>
12           <origin xyz="0 0 0" rpy="0 0 0" />
13           <geometry>
14               <box size="0.2 0.2 0.2" />
15           </geometry>
16           <material name="sick_box">
17               <color rgba="0.7 0.5 0.3 1.0"/>
18           </material>
19       </visual>
20
21       <inertial>
22           <mass value="4.0" />
23           <origin xyz="0 0 0" rpy="0 0 0"/>
24           <inertia ixx="0.01" ixy="0" ixz="0" iyy="0.01" iyz="0" izz="0.01" />
25       </inertial>
26   </link>
27   <!--the above displays a box meant to imply Lidar-->
28
29   <joint name="lidar_joint" type="fixed">
30       <axis xyz="0 1 0" />
31       <origin xyz="0.1 0 0.56" rpy="0 0 0"/>
32       <parent link="base_link"/>
33       <child link="lidar_link"/>
34   </joint>
```

In listing 5.3, lines 11-19 describe the visual appearance of a simple box meant to represent a LIDAR sensor. Within this block, lines 16-18 show how one can set color for rviz display. Recall that rviz and gazebo use different color representations. An additional ¡gazebo¿ field could be added to describe the color to gazebo as well, but lacking this, the gazebo appearance will default to light gray.

In fact, defining visual, collision and inertial properties for the sensor seem to be overkill, since we are primarily concerned with emulating the sensor physics. To compute transforms consistently, however, we must associate the sensor with a link in the model–and a link must be attached to the model via a joint. The physics engine of Gazebo also insists that every link includes inertial properties. The collision and visual blocks could be ignored, but including them allows us to be more realistic in the model.

To include computations of an equivalent LIDAR, a gazebo plug-in is used. Following the tutorial at `http://gazebosim.org/tutorials?tut=ros_gzplugins#GPULaser`, and modifying values lightly to apply to a Sick200 LIDAR, emulation of LIDAR is enabled by including the block of code in Listing 5.4 (extracted from `mobot_w_lidar.xacro`).

Listing 5.4: Gazebo block to include LIDAR emulation plug-in in the mobot model

```
    <!-- here is the gazebo plug-in to simulate a lidar sensor -->
  <gazebo reference="lidar_link">
    <sensor type="gpu_ray" name="sick_lidar_sensor">
      <pose>0 0 0 0 0 0</pose>
      <visualize>false</visualize>
      <update_rate>40</update_rate>
      <ray>
        <scan>
```

```
9           <horizontal>
10            <samples>181</samples>
11            <resolution>1</resolution>
12            <min_angle>-1.570796</min_angle>
13            <max_angle>1.570796</max_angle>
14          </horizontal>
15        </scan>
16        <range>
17          <min>0.10</min>
18          <max>80.0</max>
19          <resolution>0.01</resolution>
20        </range>
21        <noise>
22          <type>gaussian</type>
23          <mean>0.0</mean>
24          <stddev>0.01</stddev>
25        </noise>
26      </ray>
27      <plugin name="gazebo_ros_lidar_controller" filename="libgazebo_ros_gpu_laser.so"↩
            >
28        <topicName>/scan</topicName>
29        <frameName>lidar_link</frameName>
30      </plugin>
31    </sensor>
32  </gazebo>
```

In Listing 5.4, lines 2-4 declare that the sensor is to be located coincident with the `lidar_link` frame. That is, relative to the `lidar_link` frame, the sensor frame transform is (x,y,z) = (0,0,0), and (roll,pitch,yaw) = (0,0,0), i.e. identical to the `lidar_link` frame.

Lines 6-26 set various parameters of the LIDAR to be emulated, including scan repetition rate, start angle, end angle, angular resolution (angle increments between samples), min and max range, and an option for adding noise to the computed result (so as to more realistically simulate an actual LIDAR).

Line 27,

```
<plugin name="gazebo_ros_lidar_controller" filename="libgazebo_ros_gpu_laser.so">
```

references the gazebo library that contains the code for simulating a LIDAR sensor. Importantly, this specific library assumes use of a Graphical Processing Unit (GPU) on the host computer. Use of a GPU makes this computation much faster. However, it also imposes constraints on the hardware. If a GPU is not present, the LIDAR simulator will attempt to run, but it will output meaningless range values –e.g., all values set to the minimum LIDAR range. (If you run into this problem, it may help to install "bumblebee" and launch gazebo with "optirun" to direct the gpu-based code to run appropriately on available graphics chips; search on these keywords for possible solutions, if needed).

With the LIDAR additions to our mobile-robot model, we can start it up by: starting gazebo, loading the robot model onto the parameter server, spawning the model into gazebo, bringing an interesting virtual world into gazebo (something for the LIDAR to sense), starting a `robot_state_publisher`, starting rviz, and configuring rviz to display the LIDAR sensor topic. This process can become tedious, and fortunately it can be automated with launch files. But for now, to illustrate the steps, we will start these separately with the following commands. First, launch gazebo (optionally, with optirun):

```
(optirun) roslaunch gazebo_ros empty_world.launch
```

A launch file to bring in the modified robot is in the package `mobot_urdf` (in the "launch" subdirectory), called `mobot_w_lidar.launch`. The contents are shown in Listing 5.5.

Listing 5.5: Launch file for mobot with lidar

```
<launch>
<!-- Convert xacro model file and put on parameter server -->
<param name="robot_description" command="$(find xacro)/xacro.py '$(find mobot_urdf)/urdf/mobot_w_lidar.xacro'" />

<!-- Spawn the robot from parameter server into Gazebo -->
<node name="spawn_urdf" pkg="gazebo_ros" type="spawn_model" args="-param robot_description -urdf -model mobot" />

<!-- start a robot_state_publisher -->
<node name="robot_state_publisher" pkg="robot_state_publisher" type="robot_state_publisher" />
</launch>
```

This launch file brings in the modified robot-with-lidar model, puts it on the parameter server, and spawns it in gazebo. Additionally, this launch file starts up the robot state publisher node.

A more interesting gazebo environment can be brought in. In the gazebo display, under the "insert" tab, one can select "Starting Pen." It will move around in the gazebo scene until the mouse is clicked, which establishes its location in the world. When doing this, be careful not to place it such that the robot is embedded in a wall, or the physics simulation will blow up.

A view of the gazebo display with the modified robot in the starting pen is shown in Fig 5.10.

FIGURE 5.10: Gazebo view of simple mobile robot with LIDAR sensor in a virtual world

At this point, gazebo is computing simulated LIDAR. A `rostopic echo scan` will display output like the following (truncated):

```
frame_id: lidar_link
angle_min: -1.57079994678
angle_max: 1.57079994678
angle_increment: 0.0174533333629
```

```
time_increment: 0.0
scan_time: 0.0
range_min: 0.10000000149
range_max: 80.0
ranges: [1.4379777908325195, 1.458155632019043, 1.430367350578308, 1.4546812772750854, 1.4458516836166382, 1.
```

The values for frame id, max angle, min angle, angle increment, range min, and range max correspond to the vales in our URDF model within the LIDAR gazebo block. The vector of range values contains 181 radii (in meters) corresponding to individual LIDAR rays. Note also that the topic name "scan" is the "topicName" value set in the gazebo plug-in.

Next, bring up rviz, with: gazebo (optionally, with optirun):

```
rosrun rviz rviz
```

In the rviz display, "Add" a display item called "LaserScan." Expand this item in the "displays" window, and beside the "Topic" field, click to show the drop-down menu of options and choose the topic "/scan," which is the topic to which are emulated LIDAR instrument publishes its data.

With these settings, the rviz display appears as in Fig 5.11.

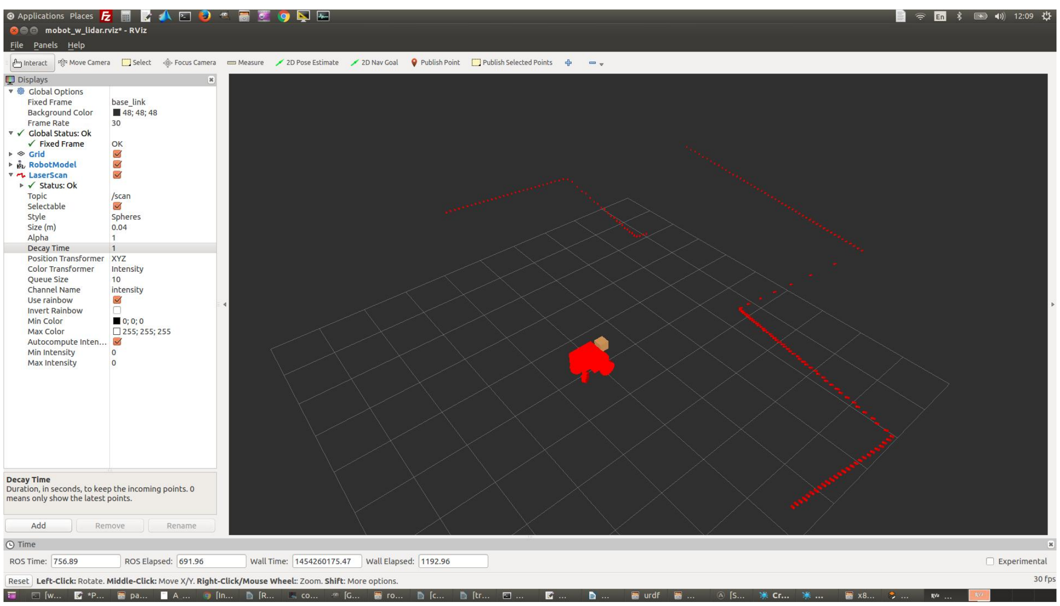


FIGURE 5.11: Rviz view of simple mobile robot with LIDAR sensor data being displayed

The rviz view is still not as interesting as the gazebo view. However, what we can see in the rviz view is the information available to the robot to perceive its environment. Lacking other visual sensors, the robot cannot know the detail of its world, as displayed by gazebo.

A software developer can interpret the rviz display to help understand what signal processing would be appropriate for the robot to function usefully in the world. Code can be written, e.g., to make maps of the environment (at least at the height of the LIDAR's slice plane) and to reconcile sensor data with such maps to estimate the robot's pose in the world. Alternatively, the LIDAR data could be used to attempt path planning without a map, e.g. by following walls or seeking corridors of suitable clearance.

The rviz view can be customized with more than a dozen display options to make the sensor display easier to interpret. Once the settings are as desired (including display items, topics, colorization, etc), one can save the rviz settings for future use. In rviz, on the top menu, under `file->save config as`, one gets the option to save the current rviz settings with a name and directory of choice. In the present case, rviz settings were saved to a file called `mobot_w_lidar.rviz` in a subdirectory `rviz_config` within the package `mobot_urdf`.

Rviz can be launched automatically from a launch file and directed to use a desired configuration file. The launch file `robot_w_lidar_and_rviz.launch` within the `launch` subdirectory of package `mobot_urdf` is identical to `mobot_w_lidar.launch` except for one additional line:

```
<node pkg="rviz" type="rviz" name="rviz" args="-d $(find mobot_urdf)/rviz_config/↩
    mobot_w_lidar.rviz"/>
```

This addition to the launch file starts rviz running, specifically directed to start up with a specified configuration file.

Although the rviz view in Fig 5.11 is relatively impoverished, it is appropriate and realistic. If a physical robot with a LIDAR sensor publishes its LIDAR data, the result can be visualized in rviz, and it would look essentially the same as Fig 5.11, except that individual points would correspond to samples in a real environment.

In addition to using rviz to help develop sensor-based behaviors, rviz can also be used as a remote operator interface for either teleoperation or supervisory control of robots. If the remote operator has access to the sensor topics from the robot, the operator can interpret the robot's environment in terms of the sensor display. Further, the operator can interact directly with rviz, e.g. using interactive markers, to specify regions of interest or robot or end-effector goal poses.

Rviz also supports plug-ins for adding functionality. A useful rviz tool for interacting with data is the `selected_points_publisher` (from T.U. Berlin, Robotics and Biology Laboratory [?]). A copy of this package is contained in the accompanying repository `learning_ros_external_packages` within the package `rviz_plugin_selected_points_topic`. When you run rviz for the first time, this tool is not normally present. There are installation instructions within the README file in this package. It may be necessary to run:

```
catkin_make install
```

for ROS to be able to find this plug-in. From rviz, on the title bar there is a blue "+" sign. Upon clicking this symbol, a menu will pop up, as shown in Fig **??**. In the example of Fig **??**, "PublishSelectedPoints" is grayed out, since this tool was alread installed in rviz.

With the Publish Selected Points tool installed, it can be enabled by clicking its icon on the rviz titlebar. One may then click/drag within the rviz scene to select and publish points of interest.

Figure 5.13 shows the LIDAR display with the mobile robot closer to the exit from the starting pen. The individual LIDAR points are displayed using relatively large, red spheres as markers (selectable in rviz within the LaserScan display item). Within the LaserScan display item, the option "selectable" is checked (enabled). Consequently, when when click-/drags to enclose one or more of these points, the corresponding coordinates are published. Figure 5.13 shows a single LIDAR point selected (indicated by being enclosed by a light-blue wireframe box).

The PublishSelectedPoints tool publishes information about user-selected points to the topic `selected_points`. This topic carries messages of type `sensor_msgs/PointCloud2`.

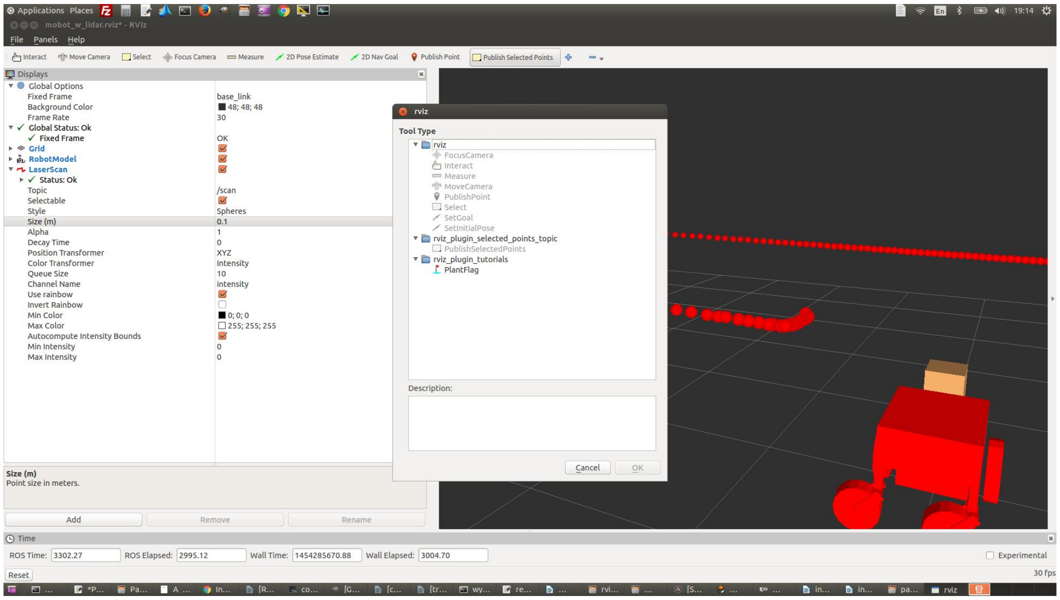


FIGURE 5.12: View of rviz in operation of adding a plug-in tool

With `rostopic echo selected_points`, we can see the result of the selected-points publisher. After selecting the LIDAR point shown in Fig 5.13, the selected-points topic echo displays:

```
fields:
  -
    name: x
    offset: 0
    datatype: 7
    count: 1
  -
    name: y
    offset: 4
    datatype: 7
    count: 1
  -
    name: z
    offset: 8
    datatype: 7
    count: 1
is_bigendian: False
point_step: 12
row_step: 12
data: [230, 233, 118, 63, 26, 249, 52, 64, 41, 92, 15, 63]
```

This format is not as obvious as previous sensor messages. Briefly, it declares in a header that the points will be represented by x, y and z coordinates, represented with 4 bytes per coordinate (i.e., single-precision floating point). The "data" component of the message carries a potentially large number of bytes, but this this example there are only 12 bytes (3 coordinates at 4 bytes each). The format for PointCloud messages is less convenient than most ROS messages since it is had to be designed to carry potentially large amounts of data

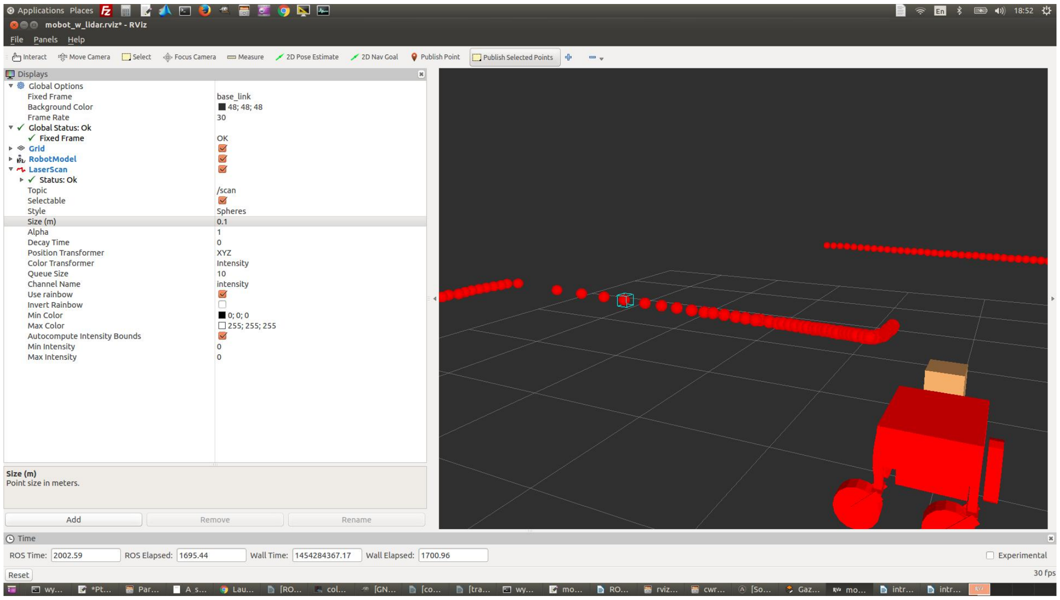


FIGURE 5.13: Rviz view showing selection of a single LIDAR point to be published

that may need to be processed efficiently. More interpretation of PointCloud messages will be covered in Chapter ?? in the context of point-cloud processing.

It can be illuminating to see the LIDAR data change dynamically in rviz as the robot moves around. The robot can be commanded to move (open-loop) in circles from a command line by entering:

```
rostopic pub -r 2 cmd_vel geometry_msgs/Twist  '{linear:  {x: 0.5, y: 0.0, z: 0.0}, 
    angular: {x: 0.0,y: 0.0,z: 0.2}}'
```

The resulting rviz view will show the LIDAR points refreshing as the robot changes its perspective. The rviz view, with its fixed frame set to `base_link`, shows a stationary robot with sensor data translating and rotating with respect to the robot, i.e. from the perspective of the robot. In contrast, the gazebo view shows the robot moving within a stationary (virtual) world. Although rviz sensor data is displayed with respect to the robot, the viewpoint can be translated and rotated, which can help the observer get a better sense of 3-D.

The value of rotating an rviz viewpoint to get a sense of 3-D is more striking when displaying richer sets of 3-D data. For this, we will need to introduce cameras.

5.2.2 Simulating and displaying color-camera data

An impressive capability of gazebo is simulation of color cameras. As with the LIDAR, we can use a gazebo plug-in to emulate a color camera. The format to do so is similar.

One option is to edit the `mobot_w_lidar.xacro` file to add in more detail to include emulation of a camera. However, it is more convenient to model the camera separately, then include the camera model in an integrated robot model. A camera model, `example_camera.xacro`, is included in the `mobot_urdf` package. The contents of this model file is shown in Listing 5.6.

Listing 5.6: Example model file for a camera

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

  <!-- add a simulated camera, including visual, collision and inertial properties, and physics simulation-->
  <link name="camera_link">
      <!-- here is the physical body (case) of the camera-->
      <collision>
          <origin xyz="0 0 0" rpy="0 0 0"/>
          <geometry>
              <box size="0.1 0.02 0.02"/>
          </geometry>
      </collision>

      <visual>
          <origin xyz="0 0 0" rpy="0 0 0" />
          <geometry>
              <box size="0.1 0.02 0.02"/>
          </geometry>
          <material name="camera_case">
              <color rgba="0.7 0.0 0.0 1.0"/>
          </material>
      </visual>

      <inertial>
          <mass value="0.1" />
          <origin xyz="0 0 0" rpy="0 0 0"/>
          <inertia ixx="0.0001" ixy="0" ixz="0" iyy="0.0001" iyz="0" izz="0.0001" />
      </inertial>
  </link>

 <!-- here is the gazebo plug-in to simulate a color camera -->
 <!--must refer to the above-defined link to place the camera in space-->
  <gazebo reference="camera_link">
   <!--optionally, displace/rotate the optical frame relative to the enclosure-->
   <pose>0.1 00 0.0 0 0 0</pose>
    <sensor type="camera" name="example_camera">
      <update_rate>30.0</update_rate>
      <camera name="example_camera">
        <!--describe some optical properties of the camera-->
        <!--field of view is expressed as an angle, in radians-->
        <horizontal_fov>1.0</horizontal_fov>
        <!--set resolution of pixels of image sensor, e.g. 640x480-->
        <image>
          <width>640</width>
          <height>480</height>
          <format>R8G8B8</format>
        </image>
        <clip>
          <!--min and max range of camera-->
          <near>0.01</near>
          <far>100.0</far>
        </clip>
          <!--optionally, add noise, to make images more realistic-->
        <noise>
          <type>gaussian</type>
          <mean>0.0</mean>
          <stddev>0.007</stddev>
        </noise>
      </camera>
      <!--here is the plug-in that does the work of camera emulation-->
      <plugin name="camera_controller" filename="libgazebo_ros_camera.so">
        <alwaysOn>true</alwaysOn>
        <updateRate>10.0</updateRate> <!--can set the publication rate-->
        <cameraName>example_camera</cameraName> <!--topics will be example_camera/... -->
        <!--listen to the following topic name to get streaming images-->
        <imageTopicName>image_raw</imageTopicName>
        <!--the following topic carries info about the camera, e.g. 640x480, etc-->
        <cameraInfoTopicName>camera_info</cameraInfoTopicName>
        <!--frameName must match gazebo reference name...seems redundant-->
        <!-- this name will be the frame_id name in header of published frames-->
```

```
72         <frameName>camera_link</frameName>
73         <!-- optionally, add some lens distortion -->
74         <distortionK1>0.0</distortionK1>
75         <distortionK2>0.0</distortionK2>
76         <distortionK3>0.0</distortionK3>
77         <distortionT1>0.0</distortionT1>
78         <distortionT2>0.0</distortionT2>
79       </plugin>
80     </sensor>
81   </gazebo>
82
83 </robot>
```

In Listing 5.6, a “robot” model is defined, although this pseudo-robot consists only of a single link. As usual, the link is defined to have visual, collision and inertial properties. This link is defined to be a simple box, meant to suggest the enclosure of a camera.

The listing is more interesting starting with line 32. A gazebo tag introduces a sensor, specifically a camera, and the frame rate for publications is set to 30Hz (line 38). Camera parameters are defined on lines 39-60, including the array dimensions (640x480), the optics (equivalently, a pin-hole camera with a field-of-view angle of 1.0 rad projecting onto the image plane 640 pixels wide). When images are broadcast, they will be encoded as 8-bit values each of red, green, blue (in that order), as specified on line 47.

A minimum and maximum range for the camera are set (lines 49-53). The maximum range is a matter of computational pragmatism rather than physics. Since synthetic images are computed from ray-tracing in the simulated environment, one must put an upper bound on how far to extend the rays to make this computation practical.

Lines 55-59 add noise to the image. (see `http://gazebosim.org/tutorials?tut=sensor_noise` for details). Introducing noise helps make the synthetic images more realistic. Image processing developed using such images would be less likely to fail in practice due to unrealistic assumptions of image quality.

Lines 62-79 introduce the gazebo plug-in for camera emulation. The camera software library computes ray-tracing in the simulated world to evaluate intensities of colors for each pixel in the camera's image plane, updated at the specified frequency (if this update rate can be achieved on the target simulation computer). The topic name `image_raw` line (67) is the conventional name chosen in ROS for transmitting camera images (of type `sensor_msgs/Image`). The topic `camera_info` is the conventional name of the topic for messages describing camera paramters, via message type `sensor_msgs/CameraInfo`. As specified by line 65 the camera topic names are pre-pended with the namespace `example_camera` (i.e., yielding topics `/example_camera/image_raw` and `/example_camera/camera_info`).

The gazebo plug-in computes synthetic images and publishes them to the `image_raw` topic. These messages will have a header with frame id set to `camera_link`, as specified on line 72. Note that the frame name on line 72 must agree with the gazebo reference name on line 34, and the named frame must be associated with a corresponding link in the model file (`camera_link`, in this case).

In addition to noise, one can also introduce lens distortion effects (lines 74-78). (see `http://gazebosim.org/tutorials?tut=camera_distortion` for details). Typically, camera calibration is performed to find these parameters. Given calibration coefficients, an additional node is run that subscribes to the “raw” images, undistorts the images, and republishes them as “rectified” images. This process, however, is not part of the gazebo simulation. Rather, the gazebo simulation attempts to create and publish streaming images with realism, including noise and distortion, attempting to emulate physical cameras.

Our camera model in `example_camera.xacro` can be added to our mobile robot model, just as was done when adding an arm to the mobile base. To do so, one must specify a joint that connects the camera link to the base link. A xacro file that combines the base and camera is `mobot_w_lidar_and_camera.xacro`, given in Listing 5.7.

Listing 5.7: Model file combining base and camera

```
<?xml version="1.0"?>
<robot
     xmlns:xacro="http://www.ros.org/wiki/xacro" name="mobot">
  <xacro:include filename="$(find mobot_urdf)/urdf/mobot_w_lidar.xacro" />
  <xacro:include filename="$(find mobot_urdf)/urdf/example_camera.xacro" />

  <!-- attach the camera to the mobile robot -->
  <joint name="camera_joint" type="fixed">
    <parent link="base_link" />
    <child link="camera_link" />
    <origin rpy="0 0 0 " xyz="0.1 0 0.7"/>
  </joint>
</robot>
```

Listing 5.7 includes the prior mobile-robot model (including its LIDAR and joint publications) and also includes the example camera model. The two models are joined together by declaring a joint, `camera_joint`, that establishes the `camera_link` (the base of the camera model) as a childe of `base_link` (the base of the mobile platform). It is not necessary that the parent link be a base link. Any defined link on the mobile robot would do (including a frame on an attached arm, if desired).

The combined model can be launched with `mobot_w_lidar_and_camera.launch`, given in Listing 5.8.

Listing 5.8: Launch file for combined base and camera

```
<launch>
<!-- Convert xacro model file and put on parameter server -->
<param name="robot_description" command="$(find xacro)/xacro.py '$(find mobot_urdf)/urdf/mobot_w_lidar_and_camera.xacro'" />

<!-- Spawn the robot from parameter server into Gazebo -->
<node name="spawn_urdf" pkg="gazebo_ros" type="spawn_model" args="-param robot_description -urdf -model mobot" />

<!-- start a robot_state_publisher -->
<node name="robot_state_publisher" pkg="robot_state_publisher" type="robot_state_publisher" />

<!-- launch rviz using a specific config file -->
 <node pkg="rviz" type="rviz" name="rviz" args="-d $(find mobot_urdf)/rviz_config/mobot_w_lidar.rviz"/>

</launch>
```

To launch the new, combined model, first start gazebo:

```
roslaunch gazebo_ros empty_world.launch
```

then load the robot model onto the parameter server, spawn the robot model into gazebo, start up a robot state publisher and start rviz:

```
roslaunch mobot_urdf mobot_w_lidar_and_camera.launch
```

Rviz can display camera views by adding a “camera” item. However, a separate node can be used for this (which is somewhat more convenient and stable). By running:

```
rosrun image_view image_view image:=example_camera/image_raw
```

we start up the node `image_view` from the package `image_view` (see `http://wiki.ros.org/image_view`). As specified by the command-line argument, this will subscribe to the topic `example_camera/image_raw` and display images published on this topic. Initially, the output will be bland, since the robot is in an empty world. From gazebo, one can insert existing world models, such as the Starting Pen. Doing so, the resulting displays of Gazebo and `image_view` appear as in Fig 5.14.

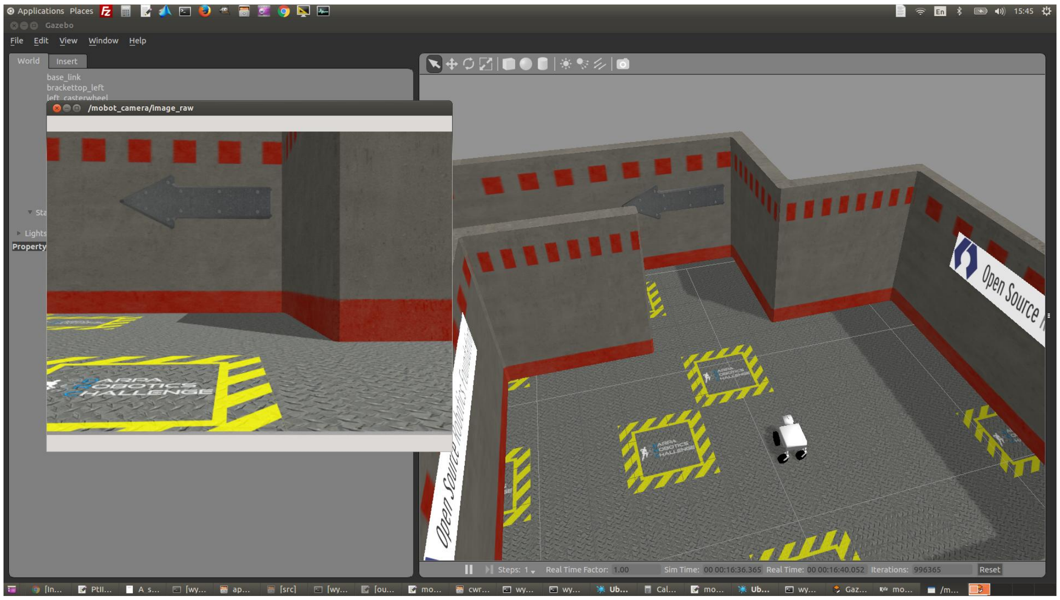


FIGURE 5.14: Gazebo view of simple mobile robot in a virtual world and display of emulated camera

Figure 5.14 shows the model robot in the Starting Pen world, as well as a display of the simulated camera. It can be seen that the camera view makes sense in terms of the pose of the robot in the world. Directions, colors and perspective are appropriate for this pose. As the robot drives around in the world, images transmitted will continue to update to reflect the robot's viewpoint. Image-processing code can be written that subscribes to this sensor topic, developed and tested in simulation, then applied to a physical system with few changes. In practice, one would need to calibrate the physical camera for its intrinsic parameters (focal length, central pixel and distortion coefficients) as well as for its extrinsic parameters (the true values for the `camera_joint` transform, specifying precisely how the camera is mounted to the robot). The gazebo model should be reconciled with the corresponding physical system to get all camera parameters in agreement. Subsequently, code developed in simulation should behave well on the real system–depending on fidelity of the virtual world model representing a real setting, although the image-processing code would likely need additional tuning on the real system.

5.2.3 Simulating and displaying depth-camera data

Another valuable sensor is the "depth camera." Various sensors, including stereo vision systems, the Kinect™camera, and some LIDARs, are able to sense 3-D coordinates of points in the environment. Some sensors, including the Kinect™, also associate color with each 3-D point, constituting an RGBD (red-green-blue-depth) camera. Similar to the LIDAR, line-of-sight trigonometry can be applied to vectors associated with image-plane pixels (and passing through a focal point), such that pixel coordinates augmented with depth information implies 3-D (and each such pixel may have associated RGB color values). Performing such computations, one can express the result as a "point cloud" (see `http://pointclouds.org/`).

To simulate a Kinect™camera, we can construct a similar model file, provided in the package `mobot_urdf` within `example_kinect.xacro`. The contents of this file appears in Listing 5.9.

Listing 5.9: Example model file for Kinect sensor

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

  <!-- add a simulated Kinecct camera, including visual, collision and inertial properties, and physics simulation-->
  <link name="kinect_link">
      <!-- here is the physical body (case) of the camera-->
      <collision>
          <origin xyz="0 0 0" rpy="0 0 0"/>
          <geometry>
              <box size="0.02 0.1 0.02"/>
          </geometry>
      </collision>

      <visual>
          <origin xyz="0 0 0" rpy="0 0 0" />
          <geometry>
              <box size="0.02 0.1 0.02"/>
          </geometry>
          <material name="camera_case">
              <color rgba="0.0 0.0 0.7 1.0"/>
          </material>
      </visual>

      <inertial>
          <mass value="0.1" />
          <origin xyz="0 0 0" rpy="0 0 0"/>
          <inertia ixx="0.0001" ixy="0" ixz="0" iyy="0.0001" iyz="0" izz="0.0001" />
      </inertial>
  </link>

 <!-- here is the gazebo plug-in to simulate a color camera -->
 <!--must refer to the above-defined link to place the camera in space-->
  <gazebo reference="kinect_link">
    <sensor type="depth" name="openni_camera_camera">
        <always_on>1</always_on>
        <visualize>true</visualize>
        <camera>
            <horizontal_fov>1.047</horizontal_fov>
            <image>
                <width>640</width>
                <height>480</height>
                <format>R8G8B8</format>
            </image>
            <depth_camera>

            </depth_camera>
            <clip>
                <near>0.1</near>
                <far>100</far>
            </clip>
        </camera>
      <!--here is the plug-in that does the work of kinect emulation-->
       <plugin name="camera_controller" filename="libgazebo_ros_openni_kinect.so">
            <alwaysOn>true</alwaysOn>
                <updateRate>10.0</updateRate>
                <cameraName>kinect</cameraName>
                <frameName>kinect_depth_frame</frameName>
            <imageTopicName>rgb/image_raw</imageTopicName>
```

```
60             <depthImageTopicName>depth/image_raw</depthImageTopicName>
61             <pointCloudTopicName>depth/points</pointCloudTopicName>
62             <cameraInfoTopicName>rgb/camera_info</cameraInfoTopicName>
63             <depthImageCameraInfoTopicName>depth/camera_info</depthImageCameraInfoTopicName>
64             <pointCloudCutoff>0.4</pointCloudCutoff>
65                 <hackBaseline>0.07</hackBaseline>
66                 <distortionK1>0.0</distortionK1>
67                 <distortionK2>0.0</distortionK2>
68                 <distortionK3>0.0</distortionK3>
69                 <distortionT1>0.0</distortionT1>
70                 <distortionT2>0.0</distortionT2>
71             <CxPrime>0.0</CxPrime>
72             <Cx>0.0</Cx>
73             <Cy>0.0</Cy>
74             <focalLength>0.0</focalLength>
75             </plugin>
76     </sensor>
77   </gazebo>
78 
79 </robot>
```

As with the example camera, this model file is called a "robot", even though it has no degrees of freedom. A link is defined to represent the housing of the Kinect sensor (lines 6-30). The gazebo tag references this link (line 34). The Kinect includes both depth information (from an infra-red camera) and color information (from an RGB camera). Many of the specifications for the Kinect camera are similar to the previous example camera model, including field of view, dimensions of image array (in pixels), range clipping, optional noise and distortion coefficients, and update rate.

The plug-in library used to emulate the kinect is referenced on line 54 (`libgazebo_ros_openni_kinect.so`).

The value of tag `cameraName` is set to "kinect", and thus all topics published by this gazebo plug-in will be in the namespace "kinect." The topic for the RBG camera, `imageTopicName` is set on line 59 to `rgb/image_raw`. To display images from the RGB camera, the appropriate topic is thus `kinect/rgb/image_raw`.

The reference frame for kinect topics, tag `frameName`, is set on line 58 to `kinect_depth_frame`. Note that this is different from the previous camera example, in which this frame was identical to the gazebo reference. The Kinect model, inconveniently, requires an additional transform to align the sensor correctly relative to its mounting link. This issue is addressed in the launch file, described later.

The kinect model is incorporated with a robot using the same technique as before–by including it hierarchically in another xacro file. The file `mobot_w_lidar_and_kinect.xacro`, given in Listing 5.10.

Listing 5.10: Model file combining robot and kinect sensor

```
<?xml version="1.0"?>
<robot
     xmlns:xacro="http://www.ros.org/wiki/xacro" name="mobot">
  <xacro:include filename="$(find mobot_urdf)/urdf/mobot_w_lidar.xacro" />
  <xacro:include filename="$(find mobot_urdf)/urdf/example_camera.xacro" />
  <xacro:include filename="$(find mobot_urdf)/urdf/example_kinect.xacro" />
  <!-- attach the camera to the mobile robot -->
  <joint name="camera_joint" type="fixed">
    <parent link="base_link" />
    <child link="camera_link" />
    <origin rpy="0 0 0 " xyz="0.1 0 0.7"/>
  </joint>
  <!-- attach the kinect to the mobile robot -->
  <joint name="kinect_joint" type="fixed">
    <parent link="base_link" />
    <child link="kinect_link" />
    <origin rpy="0 0 0 " xyz="0.1 0 0.72"/>
  </joint>
```

```
19   <!-- kinect depth frame has a different viewpoint; publish it separately-->
20 </robot>
```

Most of Listing 5.10 is the same as our prior camera model (Listing 5.7). Line 6 brings in the example kinect xacro file. Lines 14-18 define a static joint attaching the kinect link in the kinect model to the base link of the mobile-robot model.

Launching the combined model is done with the launch file `mobot_w_lidar_and_kinect.launch`, given in Listing 5.11.

Listing 5.11: Launch file for combined robot and kinect sensor

```
1  <launch>
2  <!-- Convert xacro model file and put on parameter server -->
3  <param name="robot_description" command="$(find xacro)/xacro.py '$(find mobot_urdf)/urdf/mobot_w_lidar_and_kinect.xacro'" />
4
5  <!-- Spawn the robot from parameter server into Gazebo -->
6  <node name="spawn_urdf" pkg="gazebo_ros" type="spawn_model" args="-param robot_description -urdf -model mobot" />
7
8  <node pkg="tf" type="static_transform_publisher" name="kinect_broadcaster" args="0 0 0 -0.500 0.500 -0.500 0.500 kinect_link kinect_depth_frame 100" />
9
10 <!-- start a robot_state_publisher -->
11 <node name="robot_state_publisher" pkg="robot_state_publisher" type="robot_state_publisher" />
12
13 <!-- launch rviz using a specific config file -->
14  <node pkg="rviz" type="rviz" name="rviz" args="-d $(find mobot_urdf)/rviz_config/mobot_w_lidar_and_kinect.rviz"/>
15
16 <!-- launch image_view as well -->
17  <node pkg="image_view" type="image_view" name="image_view">
18    <remap from="image" to="/kinect/rgb/image_raw" />
19  </node>
20
21 </launch>
```

This launch file illustrates two new features. First, line 8:

```
<node pkg="tf" type="static_transform_publisher" name="kinect_broadcaster" args="0 0 0 -0.500 0.500 -0.500 0.500 kinect_link kinect_depth_frame 100" />
```

starts a node from the "tf" package called `static_transform_publisher`. This node publishes a transform relationship between specified frames on the "tf" topic. Our kinect model file defined a frame for the image data, called `kinect_depth_frame`, but our URDF contains no information regarding how this frame relates to any other frame in the model. The `kinect_depth_frame` is not associated with any physical link–only with reference frame defined with respect to the sensor itself. The static transform publisher node is provided with the names of a child frame, `kinect_depth_frame`, and a parent frame, `kinect_link`, and transform parameters between them. The arguments (0,0,0) state that the depth frame origin is coincident with the kinect-link frame origin. The arguments (-0.5,0.5,-0.5,0.5) describe a quaternion orientation transformation involving 90-deg rotations about each of the x, y and z axes. By starting this static transform publisher, rviz is able to fully connect frames from the kinect depth frame to the mobile-robot's base frame. (see `http://wiki.ros.org/tf#static_transform_publisher` for more details on the static transform publisher).

A second new feature of the launch file in Listing 5.11 appears in lines 17-19:

```
<node pkg="image_view" type="image_view" name="image_view">
  <remap from="image" to="/kinect/rgb/image_raw" />
</node>
```

This instruction automates starting up the `image_view` node and instructs it to subscribe to the topic `/kinect/rgb/image_raw`. Note that from the command line, this was done with:

```
rosrun image_view image_view image:=example_camera/image_raw
```

but when started from a launch file, the syntax for topic assignment uses the `<remap>` tag. Topic remapping in launch files is a highly useful capability that will be revisited.

To try out the newly-augmented robot model, start gazebo with:

```
roslaunch gazebo_ros empty_world.launch
```

then run the launch script, which loads the robot model onto the parameter server, spawns the robot model into gazebo, starts up a robot state publisher, starts up a static transform publisher, and starts up rviz (with a new config file):

```
roslaunch mobot_urdf mobot_w_lidar_and_kinect.launch
```

The gazebo model is made more interesting by bringing in the world model "gas station." Figure 5.15 shows three displays (overlapping on the screen). The gazebo view shows the model robot within a gas-station virtual world. Recall that gazebo is a stand-in for reality. It is useful for developing code in simulation, but it is ultimately replaced by a real robot, real sensors and a real environment.

The `image_view` view shows the synthetic image computed by gazebo equivalent to the viewpoint of the Kinect's color camera viewing a gas pump. The image display appears virtually identical to the gazebo display, although it should be remembered that the gazebo display has 3-D information that can be inspected by moving the observer around relative to the virtual world. In contrast, the camera display only containts 2-D information, equivalent to values from an actual camera's image sensor.

The most interesting addition to Fig 5.15 is in the rviz view. A new display item has been added: a "PointCloud2." Within this display item, the topic is set to `kinect/depth/points`. This topic carries the Kinect 3-D points. These are rendered in rviz with consistent transformations, resulting in points being displayed that are an appropriate distance from the robot. Further, these points align virtually perfectly with the red markers indicating "pings" from the robot's LIDAR. Unlike the `image_view` scene, the rviz scene is fully 3-D. The observer can rotate this view to observe the data from alternative viewpoints. Even though gazebo gets replaced with reality, rviz views, such as in Fig 5.15, can still be displayed. This capability can give the operator perception of the robot's surroundings, thus providing potential for either teleoperation or supervisory control.

A nuisance inconsistency is that colors in the rviz display are different from the colors in the gazebo display. This is because rviz and gazebo are separate developments that use different color representations. In practice, this is not a problem. Rviz display from real cameras (including the Kinect) appears correctly. Also, many 3-D sensors do not have associated color, making this a non-issue even for gazebo simulations.

A very useful capability of point-cloud displays in rviz is that points (or collections of points) can be selected interactively and published for use by nodes. This capability is described next.

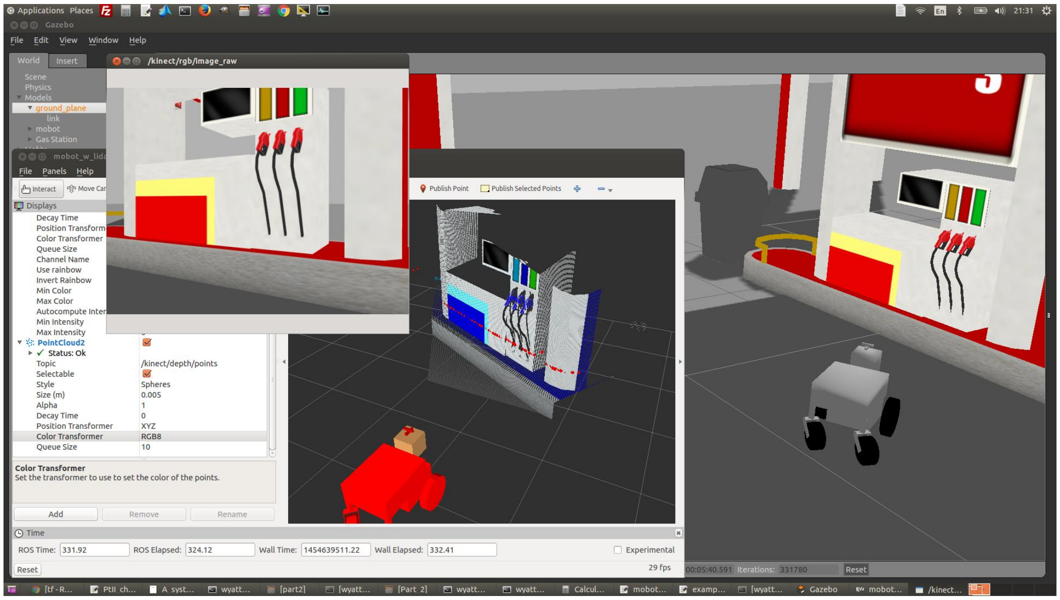


FIGURE 5.15: Gazebo, rviz and image-view displays of simple mobile robot with LIDAR, camera and Kinect sensor in a virtual world

5.2.4 Selection of Points in rviz

We have seen that rviz is a useful tool for visualization of sensor data, allowing the operator to view a display of values with respect to a robot model. As introduced in Section 5.1, one can also overlay graphics (markers) to draw attention to specific regions of the display. In addition to visualization, rviz can also be used for operator input, as we have seen with interactive markers in Section 5.1.2. A valuable additional input option is the ability to select points of interest in the rviz display, and have these point coordinates published for consumption by perceptual-processing nodes. The tool "Publish Selected Points" is a plug-in of rviz that provides this capability.

Figure 5.16 shows both gazebo and rviz displays of our simple robot with the simulated gas station world. The Kinect camera can see gas pumps. The rviz display is zoomed on the pumps to emphasize the pump handles. Note on the title bar of rviz that the tool "Publish Selected Points" is highlighted. With this tool selected, one can click-drag on the rviz scene to select a set of point-cloud points. Although the rviz view is a 2-D display, the display is generated by 3-D data. By choosing points in 2-D, the underlying 3-D source of this data can be referenced, and thus the corresponding 3-D coordinates of the selected points can be published. In Fig 5.16, a small patch of points on the pump handle has been selected, which appears highlighted in light blue.

In a terminal, the command `rostopic echo selected_points` was run to see the effect of selecting points. Once points were selected in rviz, a message (of type PointCloud2) was published on the topic `selected_points`. This message was echoed in a terminal for display. Display of the message shows that the point selection consisted of a mere 8 points. Each point is described in terms of x, y and z coordinates, each of which is encoded as 4 bytes (corresponding to datatype 7, which is defined in `sensor_msgs/PointCloud2` as a FLOAT32). This shows that one can obtain 3-D data by interacting with displayed sensor

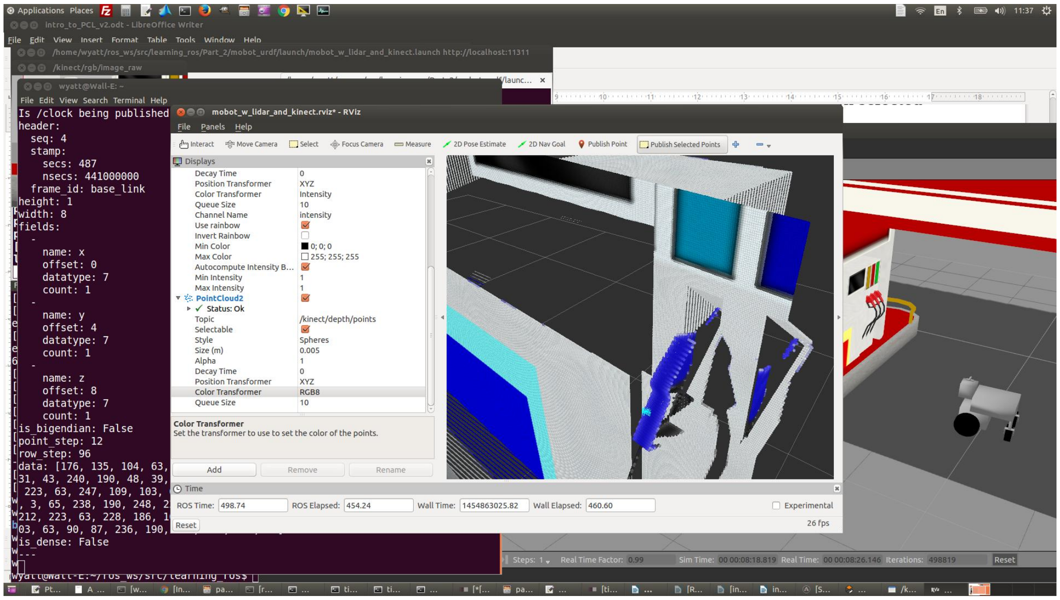


FIGURE 5.16: Gazebo view of mobot in simulated gas station, and Rviz display of simulated Kinect data. A small, light-blue patch of points on the pump handle displays user-selected points.

values in rviz. The resulting publications can be received and interpreted by ROS nodes. Further details of point-cloud messages and interpretation will be covered in Section ??.

5.3 WRAP-UP

This concludes our introduction to sensing and visualization in ROS. It should be appreciated that gazebo simulations can and should be designed such that the interfaces are indentical to corresponding physical robot systems. With attention to this commonality, it is possible to perform extensive software design and debugging in simulation, then apply the results to corresponding physical robots productively. Inevitably, some tuning is required to account for modeling imperfections, but the vast majority of programming can be done in simulation.

The rviz interface is highly useful for interpreting data from a robot–whether physical or simulated. If the simulated robot is designed to be consistent with the physical robot, then the rviz display should be realistic in simulation. With the addition of user-designed markers displayed in rviz, one can help visualize the results of perceptual processing and/or path planning, which is highly useful for development and debugging. In addition, the same display can be used as an operator interface to interpret sensory data from a remote robot, including fusion of LIDAR, point-cloud and camera data, together with display of any available a priori models.

With the addition of interactive markers and publication of selected points, the rviz display can also perform as an intuitive operator interface for supervisory control of robots.

Given this introduction to the foundations of ROS, we are ready to illustrate use of ROS for robot programming, including both re-use of existing packages and design of new capabilities.

III

Perceptual Processing in ROS

INTRODUCTION
Intro here.

- LIDAR
- image processing with OpenCV
- calibration
- image processing
- stereo calibration
- stereo images: disparity and point clouds
- point-cloud processing with PCL

CHAPTER 6

Point Cloud Processing

CONTENTS

3-D point-cloud sources; PCL;

Interpreting 3-D sensory data is enabled with the use of the "Point Cloud Library" (see `http://pointclouds.org/`). This open-source effort is independent of ROS, but it is compatible with ROS. The Point Cloud Library (PCL) offers an array of functions for interpreting 3-D data. The present treatment is not intended to be a comprehensive tutorial of PCL. Rather, a few simple capabilities are introduced that provide useful functionality. Greater expertise can be gained by consulting the on-line tutorials and code examples. (At the time of this writing, there appears to be no textbook for teaching PCL). It is hoped that the incremental examples discussed here will help to make the on-line resources more accessible.

6.1 A SIMPLE POINT-CLOUD DISPLAY NODE

The package `pcl_utils` contains the source program `display_ellipse.cpp` and the associated module `make_clouds.cpp`. This program introduces use of some basic PCL datatypes and conversions to ROS-compatible messages. This package was created using `cs_create_pkg` with options of: `roscpp`, `sensor_msgs`, `pcl_ros` and `pcl_conversions`. (Additional dependencies of `std_msgs` and `tf` are used in later examples).

Within the `display_ellipse` node, a point-cloud object is populated with computed values that describe an ellipse that is extruded in the z direction and has coloration that varies in the z direction.

The source code of the function that populates a point cloud is shown in Listing 6.1.

Listing 6.1: C++ code illustrating populating a Point Cloud

```
//make_clouds.cpp
//a function to populate two point clouds with computed points
// modified from: from: http://docs.ros.org/hydro/api/pcl/html/↩
    pcl__visualizer__demo_8cpp_source.html
#include<ros/ros.h>
#include <stdlib.h>
#include <math.h>

#include <sensor_msgs/PointCloud2.h> //ROS message type to publish a pointCloud
#include <pcl_ros/point_cloud.h> //use these to convert between PCL and ROS datatypes
#include <pcl/ros/conversions.h>
```

```

#include <pcl-1.7/pcl/point_cloud.h>
#include <pcl-1.7/pcl/PCLHeader.h>


using namespace std;

//a function to populate a pointCloud and a colored pointCloud;
// provide pointers to these, and this function will fill them with data
void make_clouds(pcl::PointCloud<pcl::PointXYZ>::Ptr basic_cloud_ptr,
        pcl::PointCloud<pcl::PointXYZRGB>::Ptr point_cloud_ptr) {
    // make an ellipse extruded along the z-axis. The color for
    // the XYZRGB cloud will gradually go from red to green to blue.

    uint8_t r(255), g(15), b(15); //declare and initialize red, green, blue component values

    //here are "point" objects that are compatible as building-blocks of point clouds
    pcl::PointXYZ basic_point; // simple points have x,y,z, but no color
    pcl::PointXYZRGB point; //colored point clouds also have RGB values

    for (float z = -1.0; z <= 1.0; z += 0.05) //build cloud in z direction
    {
        // color is encoded strangely, but efficiently.  Stored as a 4-byte "float", but
        // interpreted as individual byte values for 3 colors
        // bits 0-7 are blue value, bits 8-15 are green, bits 16-23 are red;
        // Can build the rgb encoding with bit-level operations:
        uint32_t rgb = (static_cast<uint32_t> (r) << 16 |
                static_cast<uint32_t> (g) << 8 | static_cast<uint32_t> (b));

        // and encode these bits as a single-precision (4-byte) float:
        float rgb_float = *reinterpret_cast<float*> (&rgb);

        //using fixed color and fixed z, compute coords of an ellipse in x-y plane
        for (float ang = 0.0; ang <= 2.0 * M_PI; ang += 2.0 * M_PI / 72.0) {
            //choose minor axis length= 0.5, major axis length = 1.0
            // compute and fill in components of point
            basic_point.x = 0.5 * cosf(ang); //cosf is cosine, operates on and returns single-precision floats
            basic_point.y = sinf(ang);
            basic_point.z = z;
            basic_cloud_ptr->points.push_back(basic_point); //append this point to the vector of points

            //use the same point coordinates for our colored pointcloud
            point.x = basic_point.x;
            point.y = basic_point.y;
            point.z = basic_point.z;
            //but also add rgb information
            point.rgb = rgb_float; //*reinterpret_cast<float*> (&rgb);
            point_cloud_ptr->points.push_back(point);
        }
        if (z < 0.0) //alter the color smoothly in the z direction
        {
            r -= 12; //less red
            g += 12; //more green
        } else {
            g -= 12; // for positive z, lower the green
            b += 12; // and increase the blue
        }
    }

    //these will be unordered point clouds, i.e. a random bucket of points
    basic_cloud_ptr->width = (int) basic_cloud_ptr->points.size();
    basic_cloud_ptr->height = 1; //height=1 implies this is not an "ordered" point cloud
    basic_cloud_ptr->header.frame_id = "camera"; // need to assign a frame id

    point_cloud_ptr->width = (int) point_cloud_ptr->points.size();
    point_cloud_ptr->height = 1;
    point_cloud_ptr->header.frame_id = "camera";

}
```

The function `make_clouds()` accepts pointers to PCL PointCloud objects and fills these objects with computed data. The object `pcl::PointCloud <pcl::PointXYZ >` is "templated" to accommodate different types of point clouds. We specifically consider "basic" point clouds (type `pcl::PointXYZ`), which have no color associated with individual points, and "colored" point clouds (type `pcl::PointXYZRGB`).

A PCL point cloud object contains fields for a header, which includes a field for a frame id, and components that define the "height" and "width" of the point cloud data. Point clouds can be unordered or ordered. In the former case, the point-cloud width will be the number of points, and the point-cloud height will be 1. An unordered point cloud is a "bucket of points" in no particular order.

The first argument of this function is a pointer to a simple point cloud, consisting of (x,y,z) points with no associated intensity or color. The second argument is pointer to an object of type `pcl::PointXYZRGB`, which will be populated with points that have color as well as 3-D coordinates.

Lines 28 and 29 instantiate variables of type `pcl::PointXYZ` and `pcl::PointXYZRGB`, which are consistent with elements within corresponding point clouds. An outer loop, starting on line 31, steps through values of z coordinates. At each elevation of z, coordinates x and y are computed corresponding to an ellipse. Lines 47-49 compute these points and assign them to elements of the `basic_point` object:

```
basic_point.x = 0.5*cosf(ang);
basic_point.y = sinf(ang);
basic_point.z = z;
```

On line 50, this point is appended to the vector of points within the uncolored point cloud:

```
basic_cloud_ptr->points.push_back(basic_point);
```

In lines 53-55, these same coordinates are assigned to the colored point object, `point`. The associated color is added on line 57:

```
point.rgb = rgb_float;
```

And the colored point is added to the colored pint cloud (line 58):

```
point_cloud_ptr->points.push_back(point);
```

Encoding of color is somewhat complex. The red, green and blue intensities are represented as values from 0 to 255 in unsigned short (8-bit) integers (line 25). Somewhat awkwardly, these are encoded as 3 bytes within a 4-byte, single-precision float (lines 37-41).

At each increment of z-value, the R, G and B values are altered (lines 60-67) to smoothly change the color of points in each z-plane of the extruded ellipse.

Lines 71-77 set some meta-data of the populated point clouds. The lines:

```
    basic_cloud_ptr->width = (int) basic_cloud_ptr->points.size();
    basic_cloud_ptr->height = 1; //height=1 implies this is not an "ordered" point ↩
        cloud
```

specify a "height" of 1, which implies that the point cloud does not have an associated mapping to a 2-D array, which is considered an "unordered" point cloud. Correspondingly, the "width" of the point cloud is equal to the total number of points.

When the function `make_clouds()` concludes, the pointer arguments `basic_cloud_ptr` and `point_cloud_ptr` point to "cloud" objects that are populated with data and header information and are suitable for analysis and display.

The file `display_ellipse.cpp`, displayed in Listing 6.2 shows how save a point-cloud image to disk, as well as publish a point cloud to a topic consistent with visualization via Rviz. This function instantiates point-cloud pointers (lines 24-25), which are used as arguments to the `make_clouds()` function (line 31).

Listing 6.2: C++ code illustrating publishing a Point Cloud

```
1  //display_ellipse.cpp
2  //example of creating a point cloud and publishing it for rviz display
3
4  #include<ros/ros.h>
5  #include <stdlib.h>
6  #include <math.h>
7  #include <sensor_msgs/PointCloud2.h> //ROS message type to publish a pointCloud
8  #include <pcl_ros/point_cloud.h> //use these to convert between PCL and ROS datatypes
9  #include <pcl/ros/conversions.h>
10 #include <pcl-1.7/pcl/point_cloud.h>
11 #include <pcl-1.7/pcl/PCLHeader.h>
12
13 using namespace std;
14
15 //this function is defined in: make_clouds.cpp
16 extern void make_clouds(pcl::PointCloud<pcl::PointXYZ>::Ptr basic_cloud_ptr,
17         pcl::PointCloud<pcl::PointXYZRGB>::Ptr point_cloud_ptr);
18
19 int main(int argc, char** argv) {
20     ros::init(argc, argv, "ellipse"); //node name
21     ros::NodeHandle nh;
22
23     // create some point-cloud objects to hold data
24     pcl::PointCloud<pcl::PointXYZ>::Ptr basic_cloud_ptr(new pcl::PointCloud<pcl::PointXYZ>); //no color
25     pcl::PointCloud<pcl::PointXYZRGB>::Ptr point_cloud_clr_ptr(new pcl::PointCloud<pcl::PointXYZRGB>); //colored
26
27     cout << "Generating example point-cloud ellipse.\n\n";
28     cout << "view in rviz; choose: topic= ellipse; and fixed frame= camera" << endl;
29
30     // -----use fnc to create example point clouds: basic and colored-----
31     make_clouds(basic_cloud_ptr, point_cloud_clr_ptr);
32
33     // we now have "interesting" point clouds in basic_cloud_ptr and point_cloud_clr_ptr
34     pcl::io::savePCDFileASCII ("ellipse.pcd", *point_cloud_clr_ptr); //save image to disk
35
36     sensor_msgs::PointCloud2 ros_cloud; //a ROS-compatible pointCloud message
37
38     pcl::toROSMsg(*point_cloud_clr_ptr, ros_cloud); //convert from PCL to ROS type this way
39
40     //publish the colored point cloud in a ROS-compatible message on topic "ellipse"
41     ros::Publisher pubCloud = nh.advertise<sensor_msgs::PointCloud2> ("/ellipse", 1);
42
43     //publish the ROS-type message; can view this in rviz on topic "/ellipse"
44     //need to set the Rviz fixed frame to "camera"
45     while (ros::ok()) {
46         pubCloud.publish(ros_cloud);
47         ros::Duration(0.5).sleep(); //keep refreshing the publication periodically
48     }
49     return 0;
50 }
```

On line 34, a PCL function is called to save the generated point cloud to disk:

```
pcl::io::savePCDFileASCII ("ellipse.pcd", *point_cloud_clr_ptr); //save image
```

This instruction causes the point cloud to be saved to a file name "ellipse.pcd" (in the current directory, i.e. the directory from which this node is run).

For ROS compatibility, point-cloud data is published using the ROS message type `sensor_msgs/PointCloud2`. This message type is suitable for serialization and publications, but it is not compatible with PCL operations. The packages `pcl_ros` and `pcl_conversions` offer means to convert between ROS point-cloud message types and PCL point-cloud objects.

The PCL-style point cloud is converted to a ROS-style point-cloud message, suitable for publication (lines 36-37). The ROS message `ros_cloud` is then published (line 46) on topic `ellipse` (as specified on line 41, upon construction of the publisher object).

To compile the example program, the `CMakeLists.txt` file must be edited to reference the point-cloud library. Specifically, the lines:

```
find_package(PCL 1.7 REQUIRED)
include_directories(${PCL_INCLUDE_DIRS})
```

are uncommented in the provided `CMakeLists.txt` auto-generated file. This change is sufficient to bring in the PCL library and invoke linking this library with the main program.

With a roscore running, the example program can be started with `rosrun pcl_utils display_ellipse`. This node has little output, except to remind the user to select topic "ellipse" and frame "camera" to view the point-cloud publication in Rviz. Startin up Rviz and selecting "camera" as the fixed frame and selecting "ellipse" for the topic in a "PointCloud2" display yields the result in Fig 6.1.

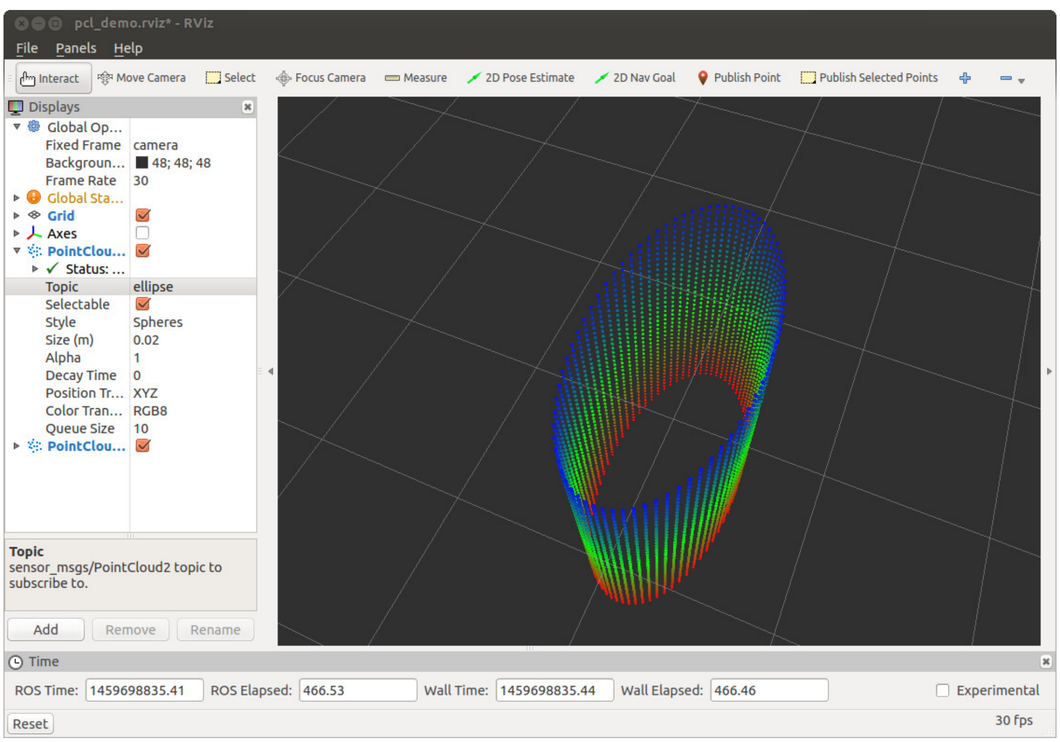


FIGURE 6.1: Rviz view of pointcloud generated and published by `display_ellipse`

As expected, the image consists of sample points of an extruded ellipse with color variation in the extrusion (z) direction. This image can be rotated, translated and zoomed in rviz, which helps one to perceive its 3-D spatial properties.

The function "savePCDFileASCII" invoked on line 34 causes the file to be saved in ASCII format, whereas an alternative command "savePCDFile" would save the point cloud in binary format. The ASCII representation should be used sparingly. ASCII files are typically more than twice the size of binary files. Further, there is (currently) a bug in the ASCII storage that results in some color corruption. Nonetheless, the ASCII option can be useful for understanding PCD file formats.

The ASCII file generated, "ellipse.pcd", can be viewed with a simple text editor. The beginning of this file is:

```
# .PCD v0.7 - Point Cloud Data file format
VERSION 0.7
FIELDS x y z rgb
SIZE 4 4 4 4
TYPE F F F F
COUNT 1 1 1 1
WIDTH 2920
HEIGHT 1
VIEWPOINT 0 0 0 1 0 0 0
POINTS 2920
DATA ascii
0.5 0 -1 2.3423454e-38
0.49809736 0.087155737 -1 2.3423454e-38
0.49240386 0.17364818 -1 2.3423454e-38
0.48296291 0.25881904 -1 2.3423454e-38
0.46984631 0.34202015 -1 2.3423454e-38
0.45315388 0.4226183 -1 2.3423454e-38
```

The file format contains metadata describing how the point cloud is encoded, followed by the data itself. (Details of the file format can be found at `http://pointclouds.org/documentation/tutorials/pcd_file_format.php`). The field names, in order, are `x y z rgb`. The `WIDTH` of the data is 2920, which is identical to the `POINTS`. The `HEIGHT` is simply "1," and `WIDTH` times `HEIGHT` = `POINTS`. This description means that the point cloud is stored merely as a list of points–an "unordered" point cloud–without any inherent structure (or at least none that is encoded in the file).

Following the description of how the data is encoded, the actual data is listed. Only 6 lines of this data is shown; the entire data includes 2920 points. As can be seen from the first 6 lines, all values of "z" are -1, which is consistent with the outer z-height loop of `make_clouds()` (which starts at z= -1). The x and y values are also consistent with the sin/cos values expected for these. The fourth value is harder to interpret, since the RGB data is encoded as 3 of 4 bytes in the single-precision floating-point value.

In the next section, we will see the alternative of an "ordered" point-cloud encoding.

6.2 LOADING AND DISPLAYING POINT-CLOUD IMAGES FROM DISK

In the previous section, a point cloud was populated with computed data. More typically, one would be working with point-cloud data aquired by some form of depth camera, e.g. from a depth camera such as the Kinect™, a scanning LIDAR, such as the Carnegie Robotics sensor head, or point-clouds generated by stereo vision. Such data is typically acquired and processed on-line during robot operation, but it is also convenient to save "snapshots" of such data to disk, e.g. for use in code development or creation of an image database.

A simple example of reading a PCD file from disk and displaying it to rviz is provided in `display_pcd_file.cpp`, which is shown in Listing 6.3.

Listing 6.3: `display_pcd_file.cpp` C++ code illustrating reading a PCD file from disk and publishing it as a Point Cloud

```
1 ///display_pcd_file.cpp
2 // prompts for a pcd file name, reads the file, and displays to rviz on topic "pcd"
3
4 #include<ros/ros.h> //generic C++ stuff
5 #include <stdlib.h>
6 #include <math.h>
7 #include <sensor_msgs/PointCloud2.h> //useful ROS message types
8 #include <pcl_ros/point_cloud.h> //to convert between PCL a nd ROS
9 #include <pcl/ros/conversions.h>
10 #include <pcl/point_types.h>
11 #include <pcl/point_cloud.h>
12 #include <pcl/common/common_headers.h>
13 #include <pcl-1.7/pcl/point_cloud.h>
14 #include <pcl-1.7/pcl/PCLHeader.h>
15
16 using namespace std;
17
18 int main(int argc, char** argv) {
19     ros::init(argc, argv, "pcd_publisher"); //node name
20     ros::NodeHandle nh;
21     pcl::PointCloud<pcl::PointXYZRGB>::Ptr pcl_clr_ptr(new pcl::PointCloud<pcl::PointXYZRGB>); //pointer for color version of pointcloud
22
23     cout<<"enter pcd file name: ";
24     string fname;
25     cin>>fname;
26
27     if (pcl::io::loadPCDFile<pcl::PointXYZRGB> (fname, *pcl_clr_ptr) == -1) //* load the file
28   {
29     ROS_ERROR ("Couldn't read file \n");
30     return (-1);
31   }
32   std::cout << "Loaded "
33             << pcl_clr_ptr->width * pcl_clr_ptr->height
34             << " data points from file "<<fname<<std::endl;
35
36    //publish the point cloud in a ROS-compatible message; here's a publisher:
37     ros::Publisher pubCloud = nh.advertise<sensor_msgs::PointCloud2> ("/pcd", 1);
38     sensor_msgs::PointCloud2 ros_cloud;  //here is the ROS-compatible message
39     pcl::toROSMsg(*pcl_clr_ptr, ros_cloud); //convert from PCL to ROS type this way
40     ros_cloud.header.frame_id = "camera_depth_optical_frame";
41
42     //publish the ROS-type message on topic "/ellipse"; can view this in rviz
43     while (ros::ok()) {
44
45         pubCloud.publish(ros_cloud);
46         ros::spinOnce();
47         ros::Duration(0.1).sleep();
48     }
49     return 0;
50 }
```

Lines 37-48 of Listing 6.3 are equivalent to corresponding lines in the previous `display_ellipse` program. The topic is changed to "pcd" (line 37) and the frame id is set to `camera_depth_optical_frame` (line 40), and these must be set correspondingly in rviz to view the output.

The primary difference with this example code is that the user is prompted to enter a file name (lines 23-25), and this file name is used to load a PCD file from disk and populate the pointer `pcl_clr_ptr` (line 27):

```
if (pcl::io::loadPCDFile<pcl::PointXYZRGB> (fname, *pcl_clr_ptr) == -1)
```

As a diagnostic, lines 32-34 inspect the "width" and "height" fields of the opened point cloud, multiply these together, and print out the result as the number of points in the point cloud.

Example PCD files reside in the accompanying respository within the ROS package `pcd_images`. A snippet of an example PCD file from a Kinect camera image, stored as ASCII, is:

```
# .PCD v0.7 - Point Cloud Data file format
VERSION 0.7
FIELDS x y z rgb
SIZE 4 4 4 4
TYPE F F F F
COUNT 1 1 1 1
WIDTH 640
HEIGHT 480
VIEWPOINT 0 0 0 1 0 0 0
POINTS 307200
DATA ascii
nan nan nan -1.9552709e+38
nan nan nan -1.9552709e+38
nan nan nan -1.9552746e+38
nan nan nan -1.9553241e+38
```

This example is similar to the ellipse PCD file, with the notable exception that the `HEIGHT` parameter is not unity. The `HEIGHTxWIDTH` is 640x480–which is the sensor resolution of the Kinect (1) color camera. The total number of points is 307,200, which is 640 times 480. In this case, the data is an "ordered" point cloud, since the listing of point data corresponds to a rectangular array.

An ordered point cloud preserves potentially valuable spatial relationships among the individual 3-D points. A Kinect camera, for example, has each 3-D point associated with a corresponding pixel in a 2-D (color) image array. Ideally, an inverse mapping also associates each pixel in the 2-D array with a corresponding 3-D point. In practice, there are many "holes" in this mapping for which no valid 3-D point is associated with a 2-D pixel. Such missing data is represented with "nan" not-a-number codes, as is the case with the initial point values in the PCD file snippet shown above.

Coordinate data within a point-cloud object typically is stored as 4-byte, single-precision floating-point values. Single-point precision is more than adequate for the level of precision of current 3-D sensors, and thus this lower-precision representation helps reduce file size without compromising precision.

Running the display node with `rosrun pcl_utils display_pcd_file` results in the user being prompted to enter a file name. If the node is run from the `pcd_images` directory, then the file name entered will not require including a directory path. Entering the file name `coke_can.pcd` results in the response:

```
Loaded 307200 data points from file coke_can.pcd
```

which is the number of points expected for a Kinect-1 image. In rviz, setting the topic to `pcd` and setting the fixed frame to `camera_depth_optical_frame` produces the display shown in Fig 6.2.

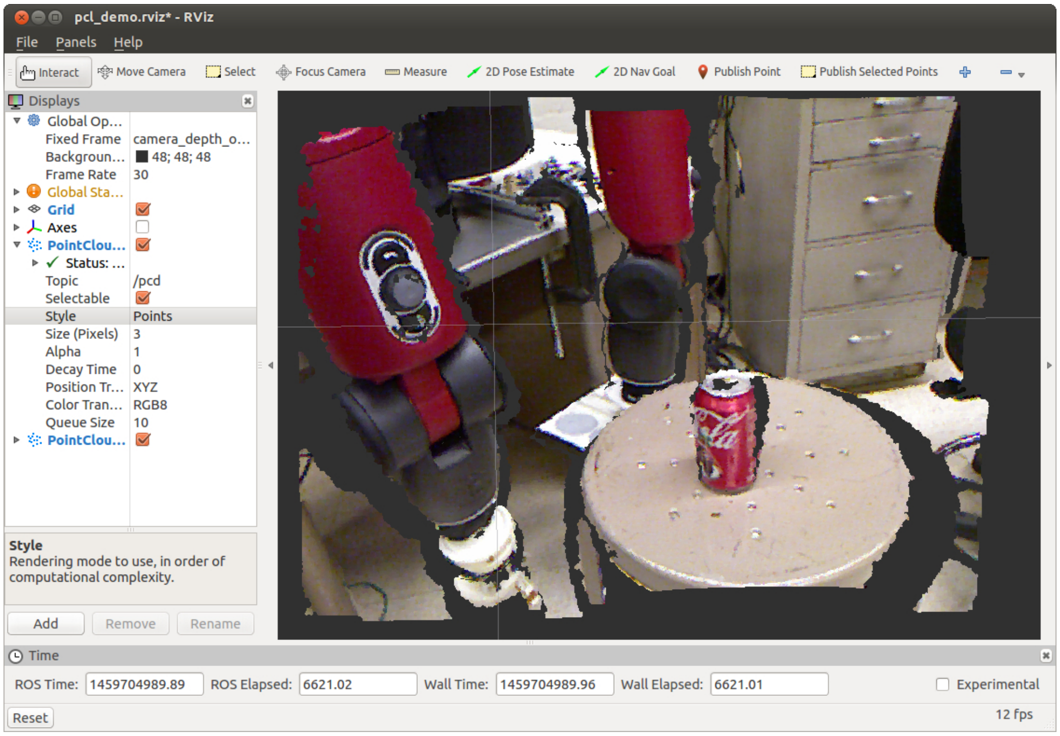


FIGURE 6.2: Rviz view of image read from disk and published by `display_pcd_file`

6.3 SAVING PUBLISHED POINT-CLOUD IMAGES TO DISK

A convenient utility program in the `pcl_utils` package is `pcd_snapshot.cpp`. The contents of this program are shown in Listing 6.4.

Listing 6.4: `pcd_snapshot.cpp` C++ code illustrating subscribing to a point-cloud topic and saving data to disk as a PCD file

```
//pcd_snapshot.cpp
// example of saving a kinect snapshot to a pcd file
// need to connect "Kinect" and start it with: roslaunch freenect_launch freenect.launch

#include <iostream>
#include <ros/ros.h>
#include <sensor_msgs/PointCloud2.h>
#include <pcl_ros/point_cloud.h> //use these to convert between PCL and ROS datatypes
#include <pcl/ros/conversions.h>
#include <pcl-1.7/pcl/point_cloud.h>
#include <pcl-1.7/pcl/PCLHeader.h>

using namespace std;

bool got_kinect_image = false; //snapshot indicator
pcl::PointCloud<pcl::PointXYZRGB>::Ptr pclKinect_clr_ptr(new pcl::PointCloud<pcl::PointXYZRGB>); //pointer for color version of pointcloud

void kinectCB(const sensor_msgs::PointCloud2ConstPtr& cloud) {
    if (!got_kinect_image) { // once only, to keep the data stable
        ROS_INFO("got new selected kinect image");
        pcl::fromROSMsg(*cloud, *pclKinect_clr_ptr);
```

```
22          ROS_INFO("image has  %d * %d points", pclKinect_clr_ptr->width, pclKinect_clr_ptr->height);
23          got_kinect_image = true;
24      }
25  }
26
27  int main(int argc, char** argv) {
28      ros::init(argc, argv, "pcl_snapshot_main"); //node name
29      ros::NodeHandle nh;
30      ros::Subscriber pointcloud_subscriber = nh.subscribe("/camera/depth_registered/points", 1, kinectCB);
31
32      //spin until obtain a snapshot
33      ROS_INFO("waiting for kinect data");
34      while (!got_kinect_image) {
35          ROS_INFO("waiting...");
36          ros::spinOnce();
37          ros::Duration(0.5).sleep();
38      }
39      ROS_INFO("got snapshot; saving to file kinect_snapshot.pcd");
40      pcl::io::savePCDFile("kinect_snapshot.pcd", *pclKinect_clr_ptr, true);
41
42      return 0;
43  }
```

In this program, a ROS subscriber subscribes to topic `camera/depth_registered/points` (line 30) with a callback function, `kinectCB()`, listening for messages of type `sensor_msgs/PointCloud2` (lines 18-25). When a message is received by the callback function, it converts the message to a `pcl::PointCloud` type (line 21). The callback function signals that it has received a new image by setting `got_kinect_image` to "true."

The main program waits for the `got_kinect_image` signal (line 34), and upon confirmation, the main program saves the acquired image as filename `kinect_snapshot.pcd` within the current directory (line 40):

```
pcl::io::savePCDFile("kinect_snapshot.pcd", *pclKinect_clr_ptr, true);
```

To use this program with a Kinect camera, plug in the camera, and with a roscore running, start up the Kinect driver:

```
roslaunch freenect_launch freenect.launch
```

(If the freenect driver is not already installed on your system, it will need to be installed).

Start up rviz, and set the fixed frame to `camera_depth_optical_frame`, add a PointCloud2 item and set its topic to `camera/depth_registered/points`. This will provide a real-time view of the Kinect data, published as a ROS point cloud.

To prepare to capture an image, open a terminal and navigate to a directory relevant for storing the image. When the rviz display shows a scene that is to be captured, run the node:

```
rosrun pcl_utils pcd_snapshot
```

This will capture a single point-cloud transmission and save the captured image to disk (in the current directory) as a PCD file called `kinect_snapshot.pcd`. The saved image will be in binary format, and it will stored the colored, ordered point cloud. To take more snapshots, rename the file `kinect_snapshot.pcd` to something mnemonic, and re-run `pcd_snapshot`. (If you do not rename the previous snapshot, the file `kinect_snapshot.pcd` will be overwritten).

The image in Fig 6.2 was acquired in this manner. The resulting PCD file can be read from disk and displayed, using `display_pcd_file`, or it can be read into another application to perform interpretation of the point-cloud data.

6.4 INTERPRETING POINT-CLOUD IMAGES WITH PCL METHODS

Within the `pcl_utils` package, the program `find_plane_pcd_file.cpp` (displayed in Listing 6.5) illustrates use of a few PCL methods. The functions are illustrated operating on a PCD file. Lines 56-65 are similar to `display_pcd_file`, in which the user is prompted for a PCD file name, which is then read in from disk. The resulting point cloud is published on topic "pcd" (lines 68, 73 and 108), making it viewable via rviz.

Listing 6.5: `find_plane_pcd_file.cpp` C++ code illustrating use of PCL methods

```
1  //find_plane_pcd_file.cpp
2  // prompts for a pcd file name, reads the file, and displays to rviz on topic "pcd"
3  // can select a patch; then computes a plane containing that patch, which is published on topic "planar_pts"
4  // illustrates use of PCL methods: computePointNormal(), transformPointCloud(),
5  // pcl::PassThrough methods setInputCloud(), setFilterFieldName(), setFilterLimits, filter()
6  // pcl::io::loadPCDFile()
7  // pcl::toROSMsg() for converting PCL pointcloud to ROS message
8  // voxel-grid filtering: pcl::VoxelGrid,  setInputCloud(), setLeafSize(), filter()
9  //wsn March 2016
10
11 #include<ros/ros.h>
12 #include <stdlib.h>
13 #include <math.h>
14
15 #include <sensor_msgs/PointCloud2.h>
16 #include <pcl_ros/point_cloud.h> //to convert between PCL and ROS
17 #include <pcl/ros/conversions.h>
18
19 #include <pcl/point_types.h>
20 #include <pcl/point_cloud.h>
21 //#include <pcl/PCLPointCloud2.h> //PCL is migrating to PointCloud2
22
23 #include <pcl/common/common_headers.h>
24 #include <pcl-1.7/pcl/point_cloud.h>
25 #include <pcl-1.7/pcl/PCLHeader.h>
26
27 //will use filter objects "passthrough" and "voxel_grid" in this example
28 #include <pcl/filters/passthrough.h>
29 #include <pcl/filters/voxel_grid.h>
30
31 #include <pcl_utils/pcl_utils.h>  //a local library with some utility fncs
32
33
34 using namespace std;
35 extern PclUtils *g_pcl_utils_ptr;
36
37 //this fnc is defined in a separate module, find_indices_of_plane_from_patch.cpp
38 extern void find_indices_of_plane_from_patch(pcl::PointCloud<pcl::PointXYZRGB>::Ptr input_cloud_ptr,
39         pcl::PointCloud<pcl::PointXYZ>::Ptr patch_cloud_ptr, vector<int> &indices);
40
41 int main(int argc, char** argv) {
42     ros::init(argc, argv, "plane_finder"); //node name
43     ros::NodeHandle nh;
44     //pointer for color version of pointcloud
45     pcl::PointCloud<pcl::PointXYZRGB>::Ptr pclKinect_clr_ptr(new pcl::PointCloud<pcl::PointXYZRGB>);
46     //pointer for pointcloud of planar points found
47     pcl::PointCloud<pcl::PointXYZRGB>::Ptr plane_pts_ptr(new pcl::PointCloud<pcl::PointXYZRGB>);
48     //ptr to selected pts from Rvis tool
```

```
    pcl::PointCloud<pcl::PointXYZ>::Ptr selected_pts_cloud_ptr(new pcl::PointCloud<pcl::PointXYZ>);
    //ptr to hold filtered Kinect image
    pcl::PointCloud<pcl::PointXYZRGB>::Ptr downsampled_kinect_ptr(new pcl::PointCloud<pcl::PointXYZRGB>);

    vector<int> indices;

    //load a PCD file using pcl::io function; alternatively, could subscribe to Kinect messages
    string fname;
    cout << "enter pcd file name: "; //prompt to enter file name
    cin >> fname;
    if (pcl::io::loadPCDFile<pcl::PointXYZRGB> (fname, *pclKinect_clr_ptr) == -1) //* load the file
    {
        ROS_ERROR("Couldn't read file \n");
        return (-1);
    }
    //PCD file does not seem to record the reference frame;  set frame_id manually
    pclKinect_clr_ptr->header.frame_id = "camera_depth_optical_frame";

    //will publish  pointClouds as ROS-compatible messages; create publishers; note topics for rviz viewing
    ros::Publisher pubCloud = nh.advertise<sensor_msgs::PointCloud2> ("/pcd", 1);
    ros::Publisher pubPlane = nh.advertise<sensor_msgs::PointCloud2> ("planar_pts", 1);
    ros::Publisher pubDnSamp = nh.advertise<sensor_msgs::PointCloud2> ("downsampled_pcd", 1);

    sensor_msgs::PointCloud2 ros_cloud, ros_planar_cloud, downsampled_cloud; //here are ROS-compatible messages
    pcl::toROSMsg(*pclKinect_clr_ptr, ros_cloud); //convert from PCL cloud to ROS message this way

    //use voxel filtering to downsample the original cloud:
    cout << "starting voxel filtering" << endl;
    pcl::VoxelGrid<pcl::PointXYZRGB> vox;
    vox.setInputCloud(pclKinect_clr_ptr);

    vox.setLeafSize(0.02f, 0.02f, 0.02f);
    vox.filter(*downsampled_kinect_ptr);
    cout << "done voxel filtering" << endl;

    cout << "num bytes in original cloud data = " << pclKinect_clr_ptr->points.size() << endl;
    cout << "num bytes in filtered cloud data = " << downsampled_kinect_ptr->points.size() << endl; //
    pcl::toROSMsg(*downsampled_kinect_ptr, downsampled_cloud); //convert to ros message for publication and display

    PclUtils pclUtils(&nh); //instantiate a PclUtils object--a local library w/ some handy fncs
    g_pcl_utils_ptr = &pclUtils; // make this object shared globally, so above fnc can use it too

    cout << " select a patch of points to find corresponding plane..." << endl; //prompt user action
    //loop to test for new selected-points inputs and compute and display corresponding planar fits
    while (ros::ok()) {
        if (pclUtils.got_selected_points()) { //here if user selected a new patch of points
            pclUtils.reset_got_selected_points(); // reset for a future trigger
            pclUtils.get_copy_selected_points(selected_pts_cloud_ptr); //get a copy of the selected points
            cout << "got new patch with number of selected pts = " << selected_pts_cloud_ptr->points.size() << endl;

            //find pts coplanar w/ selected patch, using PCL methods in above-defined function
            //"indices" will get filled with indices of points that are approx co-planar with the selected patch
            // can extract indices from original cloud, or from voxel-filtered (down-sampled) cloud
```

```
102             //find_indices_of_plane_from_patch(pclKinect_clr_ptr, selected_pts_cloud_ptr, indices);
103             find_indices_of_plane_from_patch(downsampled_kinect_ptr, selected_pts_cloud_ptr, indices);
104             pcl::copyPointCloud(*downsampled_kinect_ptr, indices, *plane_pts_ptr); //extract these pts into new cloud
105             //the new cloud is a set of points from original cloud, coplanar with selected patch; display the result
106             pcl::toROSMsg(*plane_pts_ptr, ros_planar_cloud); //convert to ros message for publication and display
107         }
108         pubCloud.publish(ros_cloud); // will not need to keep republishing if display setting is persistent
109         pubPlane.publish(ros_planar_cloud); // display the set of points computed to be coplanar w/ selection
110         pubDnSamp.publish(downsampled_cloud); //can directly publish a pcl::PointCloud2!!
111         ros::spinOnce(); //pclUtils needs some spin cycles to invoke callbacks for new selected points
112         ros::Duration(0.1).sleep();
113     }
114
115     return 0;
116 }
```

A new feature introduced in this program, voxel filtering, appears in lines 78-81:

```
pcl::VoxelGrid<pcl::PointXYZRGB> vox;
vox.setInputCloud(pclKinect_clr_ptr);
vox.setLeafSize(0.02f, 0.02f, 0.02f);
vox.filter(*downsampled_kinect_ptr);
```

These instructions instantiate a PCL-library object of type "VoxelGrid" that operates on data of type `pcl::PointXYZRGB` (colored point clouds). The VoxelGrid object, "vox", has member functions that perform spatial filtering for down-sampling. (For greater detail, see `http://pointclouds.org/documentation/tutorials/voxel_grid.php#voxelgrid`). Typical of PCL methods, the first operation with "vox" is to associate point-cloud data with it (on which the object will operate). PCL attempts to avoid making copies of data, since point-cloud data can be voluminous. Pointers or reference variables are typically used to direct the methods to the data already in memory. After affiliating "vox" with the point cloud that has been read in from disk, parameters are set for "leaf size." The x, y and z values have all been set to 0.02 (corresponding to a 2cm cube).

With the instruction "filter", the "vox" object filters the input data and produces a downsampled output (which is populated in a cloud via the pointer `downsampled_kinect_ptr`). Conceptually, downsampling by this method is equivalent to subdividing a volume into small cubes and assigning each point of the original point cloud to a single cube. After all points are assigned, each non-empty cube is represented by a single point, which is the average of all points assigned to that cube. Depending on the size of the cubes, the data reduction can be dramatic, while the loss of resolution may be acceptable.

In the example code, lines 86 and 110 convert the down-sampled cloud to a ROS message and publish the new cloud to the topic `downsampled_pcd`. The sizes of the original and the downsampled clouds are printed out (lines 84-85).

Running `rosrun pcl_utils find_plane_pcd_file` prompts for a file name. Using the same file as in Fig 6.2 (`coke_can.pcd`), downsampling at the chosen resolution (2cm cubes) results in printed output:

```
done voxel filtering
num bytes in original cloud data = 307200
num bytes in filtered cloud data = 6334
```

showing that downsampling reduced the number of points by nearly a factor of 50 (in this case). Bringing up rviz, and setting the fixed frame to `camera_depth_optical_frame` and choosing the topic `downsampled_pcd` in a PointCloud2 display item produces the display in Fig 6.3. In this example, the severity of the downsampling is obvious, yet the image is still recognizable as corresonding to Fig 6.2. With the data reduction, algorithms applied to point-cloud interpretation can run significantly faster. (The choice of downsampling resolution would need to be tuned for any specific application).

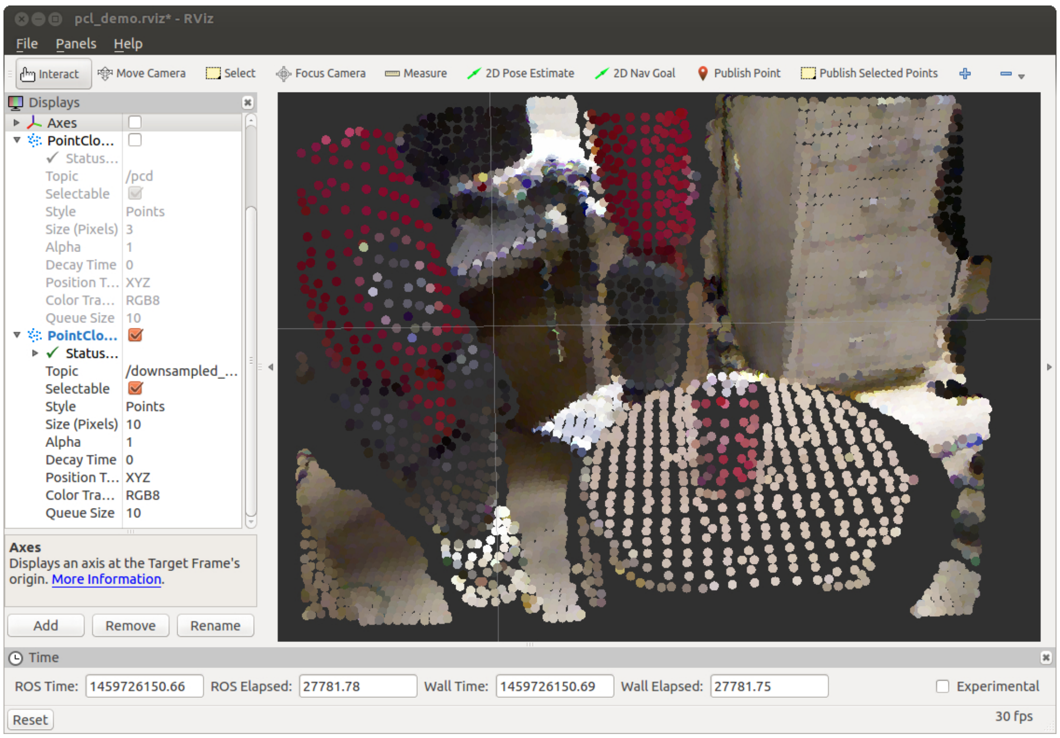


FIGURE 6.3: Rviz view of image read from disk, downsampled and published by `find_plane_pcd_file`

In addition to demonstrating downsampling, the `find_plane_pcd_file` illustrates a means of interacting with the point-cloud data to assist interpretation of the data. The example code is linked with an example library that contains additional point-cloud processing examples, `pcl_utils.cpp` (in the `pcl_utils` package). This library defines a class, `PclUtils`, an object of which is instantiated on line 88. In addition, this object is shared with external functions via a global pointer, which is defined on line 35 and assigned to point to the PclUtils object on line 89.

One of the features of PclUtils is a subscriber that subscribes to the topic `selected_points`. The publisher on this topic is an rviz tool, `publish_selected_points`, which allows a user to select points from an rviz display via click-drag of a mouse. The (selectable) points lying within the rectangle thus defined are published to the topic `selected_points` as a message type `sensor_msgs::PointCloud2`. When the user selects a region with this tool, the corresponding (underlying) 3-D data points are published, which wakes up a callback function that receives and stores the corresponding message.

Within `find_plane_pcd_file`, the value `pclUtils.got_selected_points()` is tested (line 94) to see if the user has selected a new patch of points. If so, a copy of the corresponding selected-points data is obtained (line 96) using the PclUtils method:

```
pclUtils.get_copy_selected_points(selected_pts_cloud_ptr); //get a copy of the selected points
```

When a new copy of selected points is obtained, function is called to analyze the down-sampled image in terms of a cue from the selected points (line 103):

```
find_indices_of_plane_from_patch(downsampled_kinect_ptr, selected_pts_cloud_ptr, indices);
```

The function `find_indices_of_plane_from_patch()` is implemented in a separate module, `find_indices_of_plane_from_patch.cpp`, which resides in package `pcl_utils`. A copy of the contents of this module is shown in Listing 6.6

Listing 6.6: `find_indices_of_plane_from_patch.cpp` C++ module showing PCL functions for interpretation of point-cloud data

```
//find_indices_of_plane_from_patch.cpp

#include<ros/ros.h>
#include <stdlib.h>
#include <math.h>

#include <sensor_msgs/PointCloud2.h>
#include <pcl_ros/point_cloud.h> //to convert between PCL and ROS
#include <pcl/ros/conversions.h>

#include <pcl/point_types.h>
#include <pcl/point_cloud.h>
//#include <pcl/PCLPointCloud2.h> //PCL is migrating to PointCloud2

#include <pcl/common/common_headers.h>
#include <pcl-1.7/pcl/point_cloud.h>
#include <pcl-1.7/pcl/PCLHeader.h>

//will use filter objects "passthrough" and "voxel_grid" in this example
#include <pcl/filters/passthrough.h>
#include <pcl/filters/voxel_grid.h>

#include <pcl_utils/pcl_utils.h>  //a local library with some utility fncs


using namespace std;
PclUtils *g_pcl_utils_ptr;

void find_indices_of_plane_from_patch(pcl::PointCloud<pcl::PointXYZRGB>::Ptr input_cloud_ptr,
        pcl::PointCloud<pcl::PointXYZ>::Ptr patch_cloud_ptr, vector<int> &indices) {

    float curvature;
    Eigen::Vector4f plane_parameters;

    pcl::PointCloud<pcl::PointXYZRGB>::Ptr transformed_cloud_ptr(new pcl::PointCloud<pcl::PointXYZRGB>); //pointer for color version of pointcloud

    pcl::computePointNormal(*patch_cloud_ptr, plane_parameters, curvature); //pcl fnc to compute plane fit to point cloud
    cout << "PCL: plane params of patch: " << plane_parameters.transpose() << endl;

    //next, define a coordinate frame on the plane fitted to the patch.
    // choose the z-axis of this frame to be the plane normal--but enforce that the normal
    // must point towards the camera
```

```
43     Eigen::Affine3f A_plane_wrt_camera;
44     // here, use a utility function in pclUtils to construct a frame on the computed plane
45     A_plane_wrt_camera = g_pcl_utils_ptr->make_affine_from_plane_params(plane_parameters);
46     cout << "A_plane_wrt_camera rotation:" << endl;
47     cout << A_plane_wrt_camera.linear() << endl;
48     cout << "origin: " << A_plane_wrt_camera.translation().transpose() << endl;
49 
50     //next, transform all points in input_cloud into the plane frame.
51     //the result of this is, points that are members of the plane of interest should have z-coordinates
52     // nearly 0, and thus these points will be easy to find
53     cout << "transforming all points to plane coordinates..." << endl;
54     //Transform each point in the given point cloud according to the given transformation.
55     //pcl fnc: pass in ptrs to input cloud, holder for transformed cloud, and desired transform
56     //note that if A contains a description of the frame on the plane, we want to xform with inverse(A)
57     pcl::transformPointCloud(*input_cloud_ptr, *transformed_cloud_ptr, A_plane_wrt_camera.inverse());
58 
59     //now we'll use some functions from the pcl filter library;
60     pcl::PassThrough<pcl::PointXYZRGB> pass; //create a pass-through object
61     pass.setInputCloud(transformed_cloud_ptr); //set the cloud we want to operate on--pass via a pointer
62     pass.setFilterFieldName("z"); // we will "filter" based on points that lie within some range of z-value
63     pass.setFilterLimits(-0.02, 0.02); //here is the range: z value near zero, -0.02<z<0.02
64     pass.filter(indices); //  this will return the indices of the points in   transformed_cloud_ptr that pass our test
65     cout << "number of points passing the filter = " << indices.size() << endl;
66     //This fnc populates the reference arg "indices", so the calling fnc gets the list of interesting points
67 }
```

The objective of this function is: given a patch of point-cloud points, `patch_cloud`, that are presumably nearly coplanar find all other points in the cloud `input_cloud` that are nearly co-planar with the input patch. The result is returned in a vector of point indices, specifying which points in input cloud qualify as co-planar. This allows the operator to interactively select points displayed in rviz to induce the node to find all points co-planar with the selection. This may be useful, e.g., for defining walls, doors, floor and table surfaces corresponding to the operator's focus of attention. The example function illustrates use of PCL methods `computePointNormal()`, `transformPointCloud()`, and `pcl::PassThrough` filter methods `setInputCloud()`, `setFilterFieldName()`, `setFilterLimits()`, and `filter()`.

On line 37 of `find_indices_of_plane_from_patch()`, the input cloud (a presumed nearly-coplanar patch of points) is analyzed to find a least-squares fit of a plane to the points.

```
pcl::computePointNormal(*patch_cloud_ptr, plane_parameters, curvature); //pcl fnc to compute plane fit to point cloud
```

The fitted plane is described in terms of four parameters: the components (nx,ny,nz) of the plane's surface normal, and the minimum distance of the plane from the camera's origin (focal point). These parameters are returned in an Eigen-type 4x1 vector via a the reference argument `plane_parameters`. Additionally, the (scalar) curvature is returned in the argument "curvature." The function `computePointNormal()` uses an eigenvalue/eigenvector approach to provide a robust best-estimate to a plane fit through the provided points. (See `http://pointclouds.org/documentation/tutorials/normal_estimation.php` for an explanation of the mathematical theory behind this implementation).

When fitting a plane to points, there is an ambiguity regarding direction of the plane normal. The data alone cannot distinguish positive vs. negative normal. Physically, solid objects have surfaces that distinguish “inside” from “outside”, and by convention the surface normal is defined to point towards the exterior of the object.

In the current example, the point-cloud data is presumed to be expressed with respect to the camera’s viewpoint. Consequently, all points observable by the camera must correspond to surfaces that have a surface normal pointing (at least partially) towards the camera. When fitting a plane to points, if the computed surface normal has a postive z component, then the normal vector must be negated to be logically consistent.

Additionally, the distance of a plane from a point (the focal point, in this case) is unambiguously defined. However, a signed distance is more useful, where the distance is defined is the displacement of the plane from the reference point along the plane’s normal. As such, since all surface normals must have a negative z component, all surfaces also must have a negative displacement from the camera-frame origin.

These corrections are applied, as necessary, within the function `make_affine_from_plane_params()` (invoked on line 45). This function is a method within the `pcl_utils` library. The implementation of this method is displayed in Listing 6.7.

Listing 6.7: Method `make_affine_from_plane_params()` from the library `Pcl_utils`

```
// given plane parameters of normal vec and distance to plane, construct and return an Eigen Affine object
// suitable for transforming points to a frame defined on the plane

Eigen::Affine3f PclUtils::make_affine_from_plane_params(Eigen::Vector4f plane_parameters) {
    Eigen::Vector3f plane_normal;
    double plane_dist;
    plane_normal(0) = plane_parameters(0);
    plane_normal(1) = plane_parameters(1);
    plane_normal(2) = plane_parameters(2);
    plane_dist = plane_parameters(3);
    return (make_affine_from_plane_params(plane_normal, plane_dist));
}
//this version takes separate args for plane normal and plane distance
Eigen::Affine3f PclUtils::make_affine_from_plane_params(Eigen::Vector3f plane_normal, double plane_dist) {
    Eigen::Vector3f xvec,yvec,zvec;
    Eigen::Matrix3f R_transform;
    Eigen::Affine3f A_transform;
    Eigen::Vector3f plane_origin;
    // define a frame on the plane, with zvec parallel to the plane normal
    zvec = plane_normal;
    if (zvec(2)>0) zvec*= -1.0; //insist that plane normal points towards camera
    // this assumes that reference frame of points corresponds to camera w/ z axis pointing out from camera
    xvec<< 1,0,0; // this is arbitrary, but should be valid for images taken w/ zvec= optical axis
    xvec = xvec - zvec * (zvec.dot(xvec)); // force definition of xvec to be orthogonal to plane normal
    xvec /= xvec.norm(); // want this to be unit length as well
    yvec = zvec.cross(xvec);
    R_transform.col(0) = xvec;
    R_transform.col(1) = yvec;
    R_transform.col(2) = zvec;
    //cout<<"R_transform = :"<<endl;
    //cout<<R_transform<<endl;
    if (plane_dist>0) plane_dist*=-1.0; // all planes are a negative distance from the camera, to be consistent w/ normal
    A_transform.linear() = R_transform; // directions of the x,y,z axes of the plane's frame, expressed w/rt camera frame
    plane_origin = zvec*plane_dist; //define the plane-frame origin here
    A_transform.translation() = plane_origin;
    return A_transform;
}
```

Lines 20-21 test and correct (as necessary) the surface normal of the provided patch of points. Similarly, line 32 assures that the signed distance of the plane from the camera's origin is negative.

This function constructs a coordinate system on the plane defined by the plane parameters. The coordinate system has its z-axis equal to the plane normal. The x and y axes are constructed (lines 23-26) to be mutually orthogonal and to form a right-hand coordinate system together with the z axis. These axes are used to populate the columns of a 3x3 orientation matrix (lines 27-29), which constitutes the linear() field of an Eigen::Affine object (line 33). The origin of the constructed coordinate system is defined to be the point on the plane closest to the camera-frame origin. This is computed as a displacement from the camera origin, in the direction opposite the plane normal, by (signed) distance equal to the plane-distance parameter (line 34). The resulting vector is stored as the translation() field of the Eigen::Affine object. The resulting populated Affine object is returned.

After function `find_indices_of_plane_from_patch()` calls function `make_affine_from_plane_params()` (line 45 of Listing 6.6), the resulting Affine is used to transform all of the input-cloud data. Noting that object `A_plane_wrt_camera` specifies the origin and orientation of a coordinate frame defined on the plane, the input data can be transformed into the plane frame using the inverse of this Affine object. A PCL function, `transformPointCloud()` can be used to transform a complete point cloud. This is invokved on line 57 of `find_indices_of_plane_from_patch()`, specifying the input cloud (`*input_cloud_ptr`), an object to hold the transformed cloud (`*transformed_cloud_ptr`) and the desired transformation to apply to the input (`A_plane_wrt_camera.inverse()`).

```
pcl::transformPointCloud(*input_cloud_ptr, *transformed_cloud_ptr, A_plane_wrt_camera.↩
    inverse());
```

After being transformed into the coordinate frame defined on the plane, the data becomes easier to interpret. Ideally, all points co-planar with the plane's coordinate frame will have a z component of zero. In dealing with noisy data, one must specify a tolerance to accept points that have nearly-zero z components. This is done using PCL filters in lines 60-64:

```
pcl::PassThrough<pcl::PointXYZRGB> pass; //create a pass-through object
pass.setInputCloud(transformed_cloud_ptr); //set the cloud we want to operate on-
pass.setFilterFieldName("z"); // "filter" based on some range of z-value
pass.setFilterLimits(-0.02, 0.02); //set range: z value near zero, -0.02<z<0.02
pass.filter(indices); //  returns indices of the points that pass our test
```

A "PassThrough" filter object is instantiated from the PCL library that operates on colored point clouds. This object is directed to operate on the point cloud specifed using the `setInputCloud()` method. The desired filter operation is specified by the `setFilterFieldName()` method, which in this case is set to accept (pass through) points based on inspection of their z coordinate. The `setFilterLimits()` method is called to specify the tolerance; here, points with z values between -2cm and 2cm will be considered sufficiently close to zero. Finally, the method `filter(indices)` is called. This method performs the filtering operation. All points within the specified input cloud that have z values between -0.02 and 0.02 are accepted, and these accepted points are identified in terms of their indices in the point data of the input cloud. These indices are returned within the vector of integers "indices." The result is identification of points within the original point cloud that are (nearly) co-planar with the user's selected patch.

The calling function, the main program within `find_plane_pcd_file` given in Listing 6.5, invokes the function `find_indices_of_plane_from_patch()` (line 103) whenever the user selects a new set of points in rviz. This function returns with a list of points (in "indices") that qualify as co-planar with the selected patch. Although these indices were selected from the transformed input cloud (expressed in the constructed plane's frame), the same indices apply to the points in the original cloud (expressed with respect to the camera frame). The corresponding points are extracted from the original point cloud in line 104 using the PCL method `copyPointCloud()`:

```
pcl::copyPointCloud(*downsampled_kinect_ptr, indices, *plane_pts_ptr); //extract these↩
    pts into new cloud
```

With this operation, the points referenced by index values in "indices" are copied from the down-sampled point cloud into a new cloud called `plane_pts`. This cloud is converted to a ROS message (line 106) and published (line 109) using a publisher object (line 69) defined to publish to topic `planar_pts`.

Running the node `find_plane_pcd_file` on the same input file as before, points were selected from the rviz display as shown in Fig 6.4. A rectangle of points on the stool has

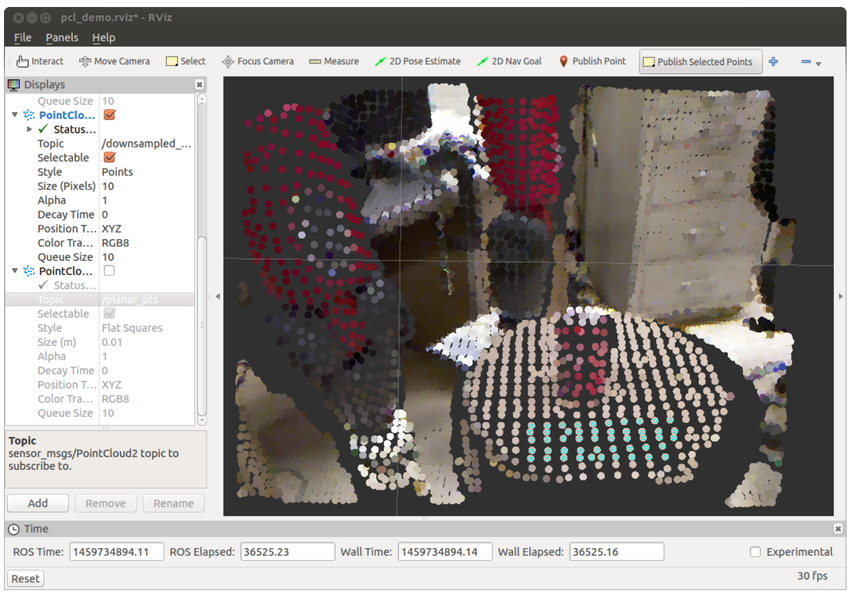


FIGURE 6.4: Scene with down-sampled point cloud and patch of selected points (in cyan)

been selected, as indicated by light cyan coloring of these points. The act of selecting these points causes the `find_plane_pcd_file` node to compute a plane through these points, then find all points from the (down-sampled) point cloud that are (nearly) co-planar, and publish the corresponding cloud on topic `planar_pts`.

Figure 6.5 shows the points chosen from the down-sampled point cloud that were determined to be co-planar with the selected patch. The points on the surface of the stool are correctly identified. However, there are additional points that lie in the same plane, including points on the furniture and the robot in the background, that are not part of the surface of the stool. The selected points could be further filtered to limit consideration to x and y values as well (e.g., based on the computed centroid of the selected patch), thus limiting point extraction to only points on the surface of the stool. Points above the stool may also be extracted, yielding identification of points corresponding to objects on a surface of interest.

The illustrations here are a mere introduction to the Point-Cloud Library. In addition to the existing methods available in this library, it typically will be necessary to compose

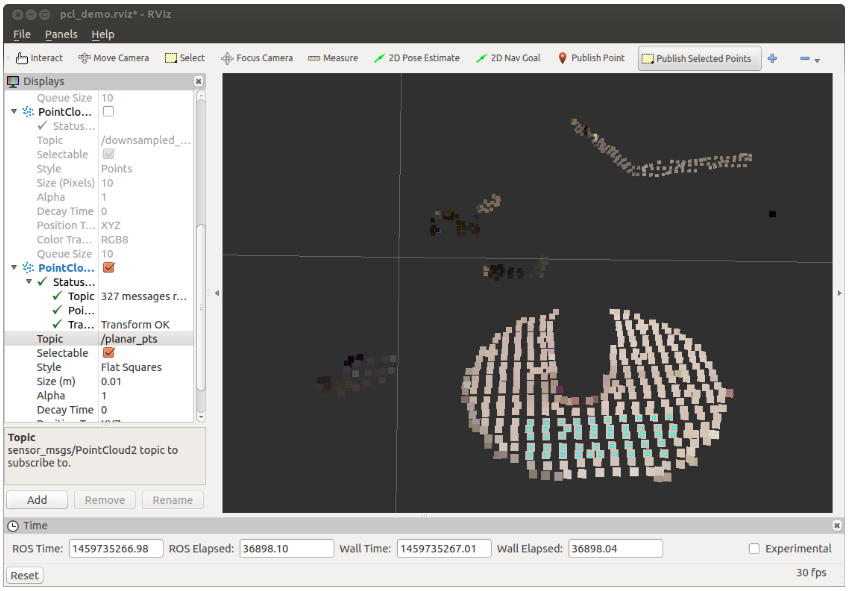


FIGURE 6.5: Points computed to be co-planar with the selected points

custom code for applications. Examples of custom code are contained within `pcl_utils.cpp` within the package `pcl_utils`. Drilling down into the PCL source code will also examples to emulate for developing custom code.

IV

Mobile Robots in ROS

INTRODUCTION
Mobile-robot navigation can be subdivided roughly into the subjects of path-planning and driving. A path planner is responsible for proposing a collision-free, efficient and navigable path to be followed. A driving sub-system is responsible for controlling the robot to execute such a plan safely and precisely. Achieving good navigation requires development of multiple, inter-related subsystems.

Global path planning requires a model (map) of the environment, and thus map-making and representation are important topics.

Global path planning typically is performed with simplifying assumptions on dynamics, resulting in coarse plans. These plans are converted to efficient, achievable trajectories through trajectory planning.

Steering algorithms are used to achieve precise following of plans, but this requires knowledge of path-following errors to be corrected. Localization is thus an crucial module used to obtain robot coordinates from which one can compute path-following errors.

The subject of path planning branches into a variety of cases, including a priori specification of a path to be followed, autonomous path planning based on knowledge of a map, planning with uncertainty or partial map information, local path planning in unknown environments with global positioning available (e.g., GPS), or exploration behaviors without benefit of maps or global positioning information. These topics are broad and are subjects of ongoing research. It is not presumed that this treatment will cover the field of mobile robotics. Rather, use of ROS for mobile platforms will be described with respect to illustrative algorithms. Further, the examples here will be limited to differential-drive ground vehicles, although the concepts are extensible to more complex systems, including holonomic vehicles, walking or crawling robots, aerial vehicles, or underwater vehicles.

CHAPTER 7

Mobile-Robot Motion Control

CONTENTS

The driving sub-system of a mobile robot is responsible for making the robot follow a desired path with sufficient precision and safety. This typically involves three sub-systems: a desired-state generator that specifies a viable sequence of states to lead the robot through a desired path; a robot state estimator that can be used to estimate deviations of the robot from the desired trajectory; and a steering algorithm that evaluates estimated states relative to desired states to compute and execute corrective actions. These three topics will be introduced in this chapter. For the purposes of this chapter, it will be assumed that a desired path has been specified. The subject of path planning (means to compute a desired path) will be deferred to Chapter **??**.

7.1 DESIRED STATE GENERATION

The purpose of a desired-state generator is to compute and stream a sequence of robot states that corresponds to following a specified path while conforming to dynamic constraints (limitations on angular and translational velocities and accelerations). The resulting desired states can be used either for open-loop control or closed-loop control.

7.1.1 From Paths to Trajectories

The "state" of a mobile platform corresponds to its pose and its twist. To be explicit, one must declare a reference frame on the vehicle as well as some reference frame of interest (e.g. coordinates of a map, absolute GPS coordinates, or simply coordinates relative to wherever the robot starts up). The vehicle frame's pose can be specified as its position (x, y, z coordinates of the origin of the vehicle frame with respect to the chosen reference frame) and its orientation (e.g. a quaternion expressing the orientation of the vehicle frame with respect to the reference frame). Additionally, the vehicle's "twist" is expressed as x, y and z velocities, as well as x, y and z rotational velocities with respect to the reference frame.

For a wheeled vehicle navigating on a plane, the pose can be expressed more compactly in terms of x, y and heading (ψ) coordinates. In the present treatment, orientation sometimes will be referred to informally as "heading" and will use the variable ψ. Use of "heading" in place of "orientation" for vehicles confined to a plane implies a scalar angle, which is simpler and more easily visualized than a full quaternion. The term "heading" can be ambiguous, since traditional navigation defines heading relative to North with angle measured clockwise. Here, heading will be defined in conventional engineering terms: the angle of the vehicle's x axis relative to the reference-frame x axis, measured from reference frame to vehicle frame as a positive rotation about the reference-frame z axis. In the plane (where the z axis points "up," normal to the plane), this is a counter-clockwise rotation from the reference-frame x axis to the vehicle-frame x axis.

For navigation in the plane, one can convert orientation to heading as follows:

Listing 7.1: Function to convert orientation quaternion to scalar heading angle in a plane

```
double  convertPlanarQuat2Psi(geometry_msgs::Quaternion quaternion) {
    double quat_z = quaternion.z;
    double quat_w = quaternion.w;
    double psi = 2.0 * atan2(quat_z, quat_w); // onversion from quaternion to heading for planar motion
    return psi;
}
```

This function takes a quaternion (expressed as a ROS message type, `geometry_msgs::Quaternion`) and converts this to a simple rotation in the x-y plane. The x and y components of a quaternion corresponding to a pure rotation about z are simply zero.

The inverse function–converting a heading angle into a quaterion–can be performed as:

Listing 7.2: Function to convert heading of planar-motion vehicle to quaternion

```
geometry_msgs::Quaternion convertPlanarPsi2Quaternion(double psi) {
    geometry_msgs::Quaternion quaternion;
    quaternion.x = 0.0;
    quaternion.y = 0.0;
    quaternion.z = sin(psi / 2.0);
    quaternion.w = cos(psi / 2.0);
    return (quaternion);
}
```

This function accepts a scalar heading angle and converts it to a ROS quaternion. Note that these conversions only apply to motion confined to a plane, for which only one degree of freedom of rotation is unconstrained. In this context, "heading" and "orientation" can be used interchangeably.

In addition to pose, a robot "state" requires specification of the "twist." A twist is a vector of 6 velocities: three translational velocities and three rotational velocities. For motion in a plane, only 3 of these 6 components are relevant: v_x, v_y and ω_z. The remaining components should be set to zero.

A ROS message type that is conventionally used to express a robot's state is `nav_msgs/Odometry`, which is comprised of the following components:

```
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
string child_frame_id
geometry_msgs/PoseWithCovariance pose
  geometry_msgs/Pose pose
    geometry_msgs/Point position
      float64 x
      float64 y
      float64 z
    geometry_msgs/Quaternion orientation
      float64 x
      float64 y
      float64 z
      float64 w
  float64[36] covariance
geometry_msgs/TwistWithCovariance twist
  geometry_msgs/Twist twist
    geometry_msgs/Vector3 linear
      float64 x
      float64 y
      float64 z
    geometry_msgs/Vector3 angular
      float64 x
      float64 y
      float64 z
  float64[36] covariance
```

This message type can be used to publish desired states, since it contains fields for both pose and twist. It is also useful to specify the reference frame in which the states are specified, which can be done via the `frame_id` component of the header. The "covariance" fields (which are useful in the context of state estimation) will not be needed for specifying desired states. Their existence makes the Odometry data structure more cumbersome than necessary, but use of the Odometry message is conventional in ROS, and using this datatype for desired states will make these messages consistent with state-estimation publications.

It is also conventional to send a path request to a mobile robot using the `nav_msgs/Path` message type, which is defined as:

```
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
geometry_msgs/PoseStamped[] poses
  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
```

Note that specification of a path contains a vector (variable-length array) of (time-stamped) poses. This specification is exclusively geometric. It does not contain any velocity nor timing information. It may also be the case that a path is defined very coarsely, e.g. in terms of via points. In order to follow the path with consideration of speed and acceleration limits, the path should be augmented to create a virtually continuous stream of states (poses and twists). Such translation converts a "path" into a "trajectory." The resulting stream of outputs constitutes desired states, and the corresponding node is a desired-state generator.

Desired state messages should be updated and published at a relatively high frequency (e.g. 50 Hz would be suitable for slowly-moving mobile robots). The sequence of states should be smooth and should conform to constraints on achievable speeds and accelerations.

To illustrate the design of a desired-state generator, we will consider a simple path description: a polyline. A polyline is simply a sequence of line segments connected at vertices. A polyline path is easily specified in a `nav_msgs/Path` message. Here, a Path message will be interpreted literally to mean the robot should follow the corresponding polyline, which requires a spin-in-place rotation at each vertex. Given non-zero mass and rotational inertia of the robot, and given non-infinite translational and rotational acceleration limits, it follows that a polyline path requires that the robot come to a full halt at each vertex of the path. Following the path corresponds to pure forward translation along each line segment followed by pure rotation at each vertex. As a consequence, polyline paths are inefficient.

A more sophisticated trajectory generator would consider fitting higher-order path segments through the specified via points, e.g. including circular arcs or splines. Execution of motion along such paths would coordinate simultaneous translational and rotational velocities to sweep through a continuous trajectory without halting. While polyline paths are assumed here for simplicity, the approach presented here is extensible to more sophisticated trajectories.

The process of converting the crude Path specification to fine-grained sequence of states is sometimes referred to as "filtering." This process can be performed in real time while the robot is running, or it can be performed in advance as a pre-computation. In order to assign viable desired states, some degree of look-ahead is required, where the look-ahead distance must exceed the braking distance required to come to a full halt. Polyline paths are the simplest case, since the robot must come to a full halt at each vertex of the path. Thus the required look-ahead distance corresponds to the next subgoal (vertex) in the vector of path poses.

With the polyline simplification, there are only two behaviors to consider: rotate from an initial heading to a desired heading, and move forward for some desired distance. In both cases, the robot should come to a complete halt at the end of the motion. Doing so requires that the velocity (linear or angular) ramp up from zero to some peak velocity, then ramp back down to zero velocity just as the desired goal pose is reached. A common technique for profiling the velocity is the "trapezoidal velocity profile."

A trapezoidal velocity profile provides a simple means to construct dynamically-feasible trajectories. If a robot's trajectory must induce a net displacement of d_{travel} and must conform to a maximum velocity and a maximum acceleration (translational or rotational), then the trapezoidal velocity profile approach involves three phases:

1. Ramp up the velocity from initial velocity (zero, in our case) to some peak, v_{cruise}, at constant ramp-up acceleration, a_{RU}.

2. Maintain constant velocity, v_{cruise}, for some distance, d_{cruise} (equivalently, some time, $t_{cruise} = d_{cruise}/v_{cruise}$).

3. Ramp down the velocity from v_{cruise} to zero at constant ramp-down deceleration, a_{RD}.

The values of v_{cruise} and d_{cruise} must be chosen to satisfy the velocity constraint $|v_{cruise}| < v_{speedLimit}$ and to result in achieving the desired total travel distance, d_{travel}. In the above, the speed $v_{speedLimit}$ could correspond to the maximum achievable velocity of the robot, or it could be defined to be some more conservative speed limit (e.g. for navigating cautiously through constricted or risky areas). Similarly, the values of ramp-up and ramp-down accelerations, a_{RU} and a_{RD}, could be set to correspond to the physical acceleration/deceleration limits of the robot, or these accelerations could be set to lower values for gentler motion (although a_{RD} might be set to physical limits, if an emergency-braking condition is called for).

The trapezoidal-velocity profile constraints can be satisfied as follows. Define a maximum ramp-up distance as: $d_{RUmax} = \frac{1}{2}v^2_{speedLimit}/a_{RU}$. Define a maximum ramp-down distance as: $d_{RDmax} = \frac{1}{2}v^2_{speedLimit}/a_{RD}$. These values define two cases to consider for computing the velocity profile: short moves and long moves. A short move corresponds to $d_{travel} \leq d_{RUmax} + d_{RDmax}$, and long moves are $d_{travel} > d_{RUmax} + d_{RDmax}$.

For short moves, the feasible velocity profile will not be able to reach $v_{speedLimit} = v_{cruise}$. Rather, the peak velocity will be $v_{peak} < v_{speedLimit}$, and the velocity profile will be triangular ($d_{cruise} = 0$, i.e. a degenerate trapezoid) instead of trapezoidal. The triangular profile will consist of a ramp-up phase covering distance $d_{RU} = \frac{1}{2}v^2_{peak}/a_{RU}$, and a ramp-down phase that covers an additional ramp-down distance, $d_{RD} = \frac{1}{2}v^2_{peak}/a_{RD}$. These two phases must cover the desired distance, so: $d_{travel} = d_{RU} + d_{RD}$. Given the desired travel distance and the ramp-up and ramp-down accelerations, one can solve for the corresponding v_{peak} as:

$$v_{peak} = \sqrt{2d_{travel}a_{RD}a_{RU}/(a_{RD} + a_{RU})} \tag{7.1}$$

For the common case of $a_{RD} = a_{RU} = a_{ramp}$, this simplifies to $v_{peak} = \sqrt{d_{travel}a_{ramp}}$. Given v_{peak}, the ramp-up and ramp-down times are $\Delta t_{RU} = v_{peak}/a_{RU}$ and $\Delta t_{RD} = v_{peak}/a_{RD}$. The total move time is $\Delta t_{move} = \Delta t_{RU} + \Delta t_{RD}$.

For the short-move case (i.e., triangular velocity profile case), the corresponding distance covered as a function of time, starting from time t_0, is:

$$d(t) = \begin{cases} \frac{1}{2}a_{RU}(t - t_0)^2 & \text{for } 0 \leq t - t_0 \leq \Delta t_{RU} \\ d_{travel} - \frac{1}{2}|a_{RD}|(\Delta t_{move} - (t - t_0))^2 & \text{for } t_{RU} \leq t - t_0 \leq \Delta t_{move} \end{cases} \tag{7.2}$$

For the long-move case, $d_{travel} > d_{RUmax} + d_{RDmax}$, the velocity profile will be a trapezoid consisting of three phases. The first phase is velocity ramp-up, during which the acceleration will be constant at a_{RU}. The velocity will ramp up to v_{cruise} (which may be set to $v_{speedLimit}$, or to some more conservative value). This phase will cover a distance d_{RUmax} in ramp-up time $\Delta t_{RU} = v_{cruise}/a_{RU}$. In the second phase, the velocity will be held constant at v_{cruise}. This phase will cover a distance $d_{cruise} = d_{travel} - d_{RUmax} - d_{RDmax}$ in time $\Delta t_{cruise} = d_{cruise}/v_{cruise}$. The third phase will be a velocity ramp down, which will cover distance d_{RDmax} in time $\Delta t_{RD} = v_{cruise}/a_{RD}$. The total move time is $t_{move} = \Delta t_{RU} + \Delta t_{cruise} + \Delta t_{RD}$.

For the trapezoidal velocity profile case, the corresponding distance covered as a function of time, starting from time t_0, is:

$$d(t) = \begin{cases} \frac{1}{2}a_{RU}(t-t_0)^2 & \text{for } 0 \le t-t_0 \le \Delta t_{RU} \\ d_{RUmax} + v_{cruise}(t-\Delta t_{RU}) & \text{for } \Delta t_{RU} \le t-t_0 < \Delta t_{RU} + \Delta t_{cruise} \\ d_{travel} - \frac{1}{2}|a_{RD}|(\Delta t_{move} - (t-t_0))^2 & \text{for } \Delta t_{RU} + \Delta t_{cruise} \le t-t_0 \le \Delta t_{move} \end{cases} \tag{7.3}$$

Implementation of these equations for generating feasible trajectories from path segments is illustrated in example software, described next.

7.1.2 A Trajectory-Builder Library

The package `traj_builder` (in `Part_4` of the `learning_ros` repository) contains a library defining a class for constructing triangular or trapezoidal velocity profiles. The code is lengthy, and thus it is not detailed here in entirety.

The class definition for the `traj_builder` library is in `traj_builder.h`. The header includes default values for maximum angular and linear accelerations (assumed to be identical for ramp-up and ramp-down), for maximum angular and linear velocities, and for the time-step resolution of the trajectories to be constructed. A default minimum-distance path-segment tolerance is also defined. Polyline line segments shorter than this tolerance are ignored, in which case the trajectory only defines reorienting to a desired heading. All of these parameters can be changed by respective "set" functions that are defined in-line in the header file.

Several utility functions are defined as member functions. The function `min_dang(psi)` evaluates periodic alternatives of a desired delta-angle and returns the option of smallest magnitude. The function `sat(x)` returns saturated value of x, saturated at +1 or -1. The function `sgn(x)` returns the sign of x (+1, -1 or 0). Conversion from a quaternion to heading is performed by the function `convertPlanarQuat2Psi()`, and the reverse-direction conversion is computed by `convertPlanarPsi2Quaternion()`. The function `xyPsi2PoseStamped(x,y,psi)` accepts a pose defined within a plane (described by coordinates x and y and heading ψ) and returns a `geometry_msgs::PoseStamped` object populated with the corresponding 6-D components. The remaining member functions perform computations to convert a path segment into a corresponding trajectory.

The main function in the `traj_builder` library is `build_point_and_go_traj()`. The contents of this function are provided in Listing 7.3.

Listing 7.3: Top-level function to convert a path segment to a trajectory

```
void TrajBuilder::build_point_and_go_traj(geometry_msgs::PoseStamped start_pose,
        geometry_msgs::PoseStamped end_pose,
        std::vector<nav_msgs::Odometry> &vec_of_states) {
    ROS_INFO("building point-and-go trajectory");
    nav_msgs::Odometry bridge_state;
    geometry_msgs::PoseStamped bridge_pose; //bridge end of prev traj to start of new traj
    vec_of_states.clear(); //get ready to build a new trajectory of desired states
    ROS_INFO("building rotational trajectory");
    double x_start = start_pose.pose.position.x;
    double y_start = start_pose.pose.position.y;
    double x_end = end_pose.pose.position.x;
    double y_end = end_pose.pose.position.y;
    double dx = x_end - x_start;
    double dy = y_end - y_start;
    double des_psi = atan2(dy, dx); //heading to point towards goal pose
    ROS_INFO("desired heading to subgoal = %f", des_psi);
    //bridge pose: state of robot with start_x, start_y, but pointing at next subgoal
```

```
18      //  achieve this pose with a spin move before proceeding to subgoal with translational
19      //  motion
20      bridge_pose = start_pose;
21      bridge_pose.pose.orientation = convertPlanarPsi2Quaternion(des_psi);
22      ROS_INFO("building reorientation trajectory");
23      build_spin_traj(start_pose, bridge_pose, vec_of_states); //build trajectory to reorient
24      //start next segment where previous segment left off
25      ROS_INFO("building translational trajectory");
26      build_travel_traj(bridge_pose, end_pose, vec_of_states);
27  }
```

This function constructs trajectories one path segment at a time, which is useful for polyline paths. It takes arguments of a start pose, a goal pose, and a reference to a vector of `nav_msgs::Odom` objects, which it will fill with a sequence of states that correspond to a smooth and executable trajectory from start to goal. This function builds a trajectory that does the following:

- find heading from start to goal coordinates
- compute a spin-in-place trajectory to point the robot towards the goal pose
- compute a forward-motion trajectory to move in a straight line, stopping at the origin of the goal pose

The computed trajectories are stored in a vector of Odometry objects, sampled every time step `dt_` (set by default to 20ms, but changeable via the function `set_dt(double dt)`). Each Odom object contains a state description corresponding to incremental subgoals. The pose components are populated based on x, y, and orientation ψ values for planar motion. The twist components are populated from corresponding forward velocity and rotational velocity. For polyline paths, the sequence of desired states starts at rest and ends at rest, first performing reorientation, then performing straight-line motion. Whether rotating or translating, the computed trajectories correspond to either triangular velocity profiles or trapezoidal velocity profiles, depending on the move distance and the acceleration and speed-limit parameters.

For the initial spin-in-place behavior, the robot must reorient to point towards the origin of the goal pose. Typically, this means the orientation of the goal pose will be ignored. However, if the start and goal poses have nearly identical (x, y) values, (as determined by the parameter `path_move_tol_`), then the goal pose is interpreted to contain the desired goal heading, and only reorientation (no translation) is computed. Thus, if one wants to travel to a pose and reach both the specified coordinates and specified orientation, one may repeat the last pose value as an additional subgoal, and this will lead to a computed trajectory that concludes with reorienting to the desired final heading.

In Listing ??, the vector of states to be computed is initially cleared (line 7). Lines 9-15 compute the heading required to point from the origin of the start pose to the origin of the goal pose. This heading, `des_psi`, becomes the goal of the initial part of the trajectory– a spin-in-place move. For this reorientation move, a "bridge" pose, (an intermediate goal pose) is constructed by copying the (x, y) coordinates of the start pose but changing the bridge-pose heading to point towards the destination coordinates. Line 21 accomplishes this by setting the orientation of `bridge_pose` to a quaternion corresponding to the computed desired heading, `des_psi`.

On line 23, the function `build_spin_traj()` is called with arguments of the start pose, the bridge pose, and a reference to the vector of desired states. This function computes a profiled trajectory with ramp-up, possible constant velocity, and ramp-down of the angular velocity that is consistent with ending up at rest with heading `des_psi`.

On line 26, the function `build_travel_traj()` is called with arguments to specify starting at the bridge pose and terminating at the (x, y) coordinates of the goal pose, as well as a reference to the partially-constructed vector of desired states. The `build_travel_traj()` function appends additional states to this vector to specify a triangular or trapezoidal velocity profile corresponding to straight-line motion from the initial (x, y) coordinates to the goal (x, y) coordinates. As noted, the robot will not necessarily end up oriented consistent with the orientation of the goal pose. To enforce reorienting to a specified pose, the goal pose should be repeated as a goal, resulting in a spin-in-place motion with no translation motion.

The functions `build_spin_traj()` and `build_travel_traj()` evaluate the move distance corresponding to the arguments for start and goal pose, and these functions will call appropriate helper functions to build either a triangular or a trapezoidal trajectory. One of these four functions, `build_triangular_travel_traj()`, is shown in Listing 7.4.

Listing 7.4: Low-level function to compute a triangular velocity profile trajectory for a straight-line path

```
// constructs straight-line trajectory with triangular velocity profile,
// respective limits of velocity and accel
void TrajBuilder::build_triangular_travel_traj(geometry_msgs::PoseStamped start_pose,
        geometry_msgs::PoseStamped end_pose,
        std::vector<nav_msgs::Odometry> &vec_of_states) {
    double x_start = start_pose.pose.position.x;
    double y_start = start_pose.pose.position.y;
    double x_end = end_pose.pose.position.x;
    double y_end = end_pose.pose.position.y;
    double dx = x_end - x_start;
    double dy = y_end - y_start;
    double psi_des = atan2(dy, dx);
    nav_msgs::Odometry des_state;
    des_state.header = start_pose.header; //really, want to copy the frame_id
    des_state.pose.pose = start_pose.pose; //start from here
    des_state.twist.twist = halt_twist_; // insist on starting from rest
    double trip_len = sqrt(dx * dx + dy * dy);
    double t_ramp = sqrt(trip_len / accel_max_);
    int npts_ramp = round(t_ramp / dt_);
    double v_peak = accel_max_*t_ramp; // could consider special cases for reverse motion
    double d_vel = alpha_max_*dt_; // incremental velocity changes for ramp-up

    double x_des = x_start; //start from here
    double y_des = y_start;
    double speed_des = 0.0;
    des_state.twist.twist.angular.z = 0.0; //omega_des; will not change
    des_state.pose.pose.orientation = convertPlanarPsi2Quaternion(psi_des); //constant
    // orientation of des_state will not change; only position and twist
    double t = 0.0;
    //ramp up;
    for (int i = 0; i < npts_ramp; i++) {
        t += dt_;
        speed_des = accel_max_*t;
        des_state.twist.twist.linear.x = speed_des; //update speed
        //update positions
        x_des = x_start + 0.5 * accel_max_ * t * t * cos(psi_des);
        y_des = y_start + 0.5 * accel_max_ * t * t * sin(psi_des);
        des_state.pose.pose.position.x = x_des;
        des_state.pose.pose.position.y = y_des;
        vec_of_states.push_back(des_state);
    }
    //ramp down:
    for (int i = 0; i < npts_ramp; i++) {
        speed_des -= accel_max_*dt_; //Euler one-step integration
        des_state.twist.twist.linear.x = speed_des;
        x_des += speed_des * dt_ * cos(psi_des); //Euler one-step integration
        y_des += speed_des * dt_ * sin(psi_des); //Euler one-step integration
        des_state.pose.pose.position.x = x_des;
```

```
49          des_state.pose.pose.position.y = y_des;
50          vec_of_states.push_back(des_state);
51      }
52      //make sure the last state is precisely where requested, and at rest:
53      des_state.pose.pose = end_pose.pose;
54      //but final orientation will follow from point-and-go direction
55      des_state.pose.pose.orientation = convertPlanarPsi2Quaternion(psi_des);
56      des_state.twist.twist = halt_twist_; // insist on starting from rest
57      vec_of_states.push_back(des_state);
58  }
```

The function `build_triangular_travel_traj()` takes three arguments: a start pose, and end pose and a reference to a vector of desired states. This function assumes that the robot is already pointing from the start pose towards the goal pose. Lines 6-12 extract the (x, y) coordinates from the start and end poses and computes the heading angle from start to pose, `psi_des`. An object of `nav_msgs::Odometry`, called `des_state`, is instantiated, and its header is populated with a copy of the start-pose header (line 15). Primarily, the value of this is to retain the `frame_id` of the input poses. On line 16, all components of twist are set to zero in the `des_state`. While building the trajectory, only the value of x velocity will be altered; all of the other 5 components will remain 0.

The trip length (in meters) and the ramp-up time (in seconds) are computed on lines 17-18. the trajectory will be computed as a sequence of samples of state. The number of samples for the ramp-up phase is computed on line 19. The peak velocity that will be achieved in the triangular trajectory is computed on line 20.

The robot's initial state is prescribed by the (x, y) coordinates of the start pose, the required heading, `des_psi`, and initial velocities of 0 (lines 23-27). A loop to compute the ramp-up trajectory begins on line 31. In lines 33-39, the desired state is updated corresponding to time t (which is incremented by `dt_` each iteration of the loop). The updates are consistent with the equations presented in Section 7.1.1. These updates are populated in `des_state` (lines 38-39) and each such state is appended to the variable-length array `vec_of_states` (line 40).

Lines 43-51 repeat the computations of state samples, but for the ramp-down phase. Instead of using an analytic expression in this loop, an alternative computation is performed using numerical integration (lines 44, 46 and 47). Because numerical integration can result in rounding errors, the final state of the trajectory is coerced to identically match the desired goal coordinates (lines 53-57).

When this function returns, the vector of states will contain the newly-computed trajectory, appended to whatever contents the vector originally contained. If this vector is to be cleared to start computing a new trajectory, it must be cleared by the parent function that calls `build_triangular_travel_traj()`. (This is true of all 4 lower-level trajectory-builder functions).

The function `build_triangular_spin_traj()` is virtually identical to `build_triangular_travel_traj()`. The triangular spin trajectory builder ramps up angular velocity then ramps down again. One important difference is that one must take into account the sign of the spin direction, which may be positive or negative. Thus, "ramping up" may consist of ramping to more negative angular velocities. The translational-trajectory builder function could be augmented to consider negative motions as well, which would be necessary to perform backing-up (reverse) motions.

The functions `build_trapezoidal_travel_traj()` and `build_trapezoidal_spin_traj()` perform equivalent logic, but for long moves that include a middle phase of travel at constant velocity. These functions behave similarly, taking arguments of start pose, end pose and reference to a vector of states and returning a computed trajectory in the vector of states.

A function `build_braking_traj()` is also declared. However, its implementation is empty. The intent of this function is to provide computation of a trajectory for a graceful halt (a ramp-down trajectory with desirable deceleration). Such a trajectory would be valuable for a robot to perform unanticipated braking, e.g. due to map errors or unexpected obstacles (e.g. debris or pedestrians). Planning a halt trajectory on the fly in an emergency situation might seem like unnecessary delay. However, such planning only requires on the order of 1ms to compute on an unexceptional computer, and thus this presents no significant delay. Implementation of this function is left as an exercise.

Illustration of use of the trajectory-builder library is provided by the node `traj_builder_example_main` in the `traj_builder` package. This node hard-codes start and goal poses, and it iterates computing trajectories from start to goal, then back from goal to start. Each computed trajectory is published, in sequence, every 20ms to the topic `desState`. Running this function, the desired state values can be plotted with `rqt_plot`. An example result is shown in Fig 7.1, where the start coordinates are $(0, 0$ and the goal coordinates are $(2, -4)$. The velocity limits were set to large values to produce triangular velocity profiles. It can be seen from Fig 7.1 that the angular (blue) and translational (brown) velocities alternately ramp up and down triangularly. The corresponding heading (magenta), x-position (green) and y-position (orange) transition smoothly between the goal values as blended quadratics with an inflection point.

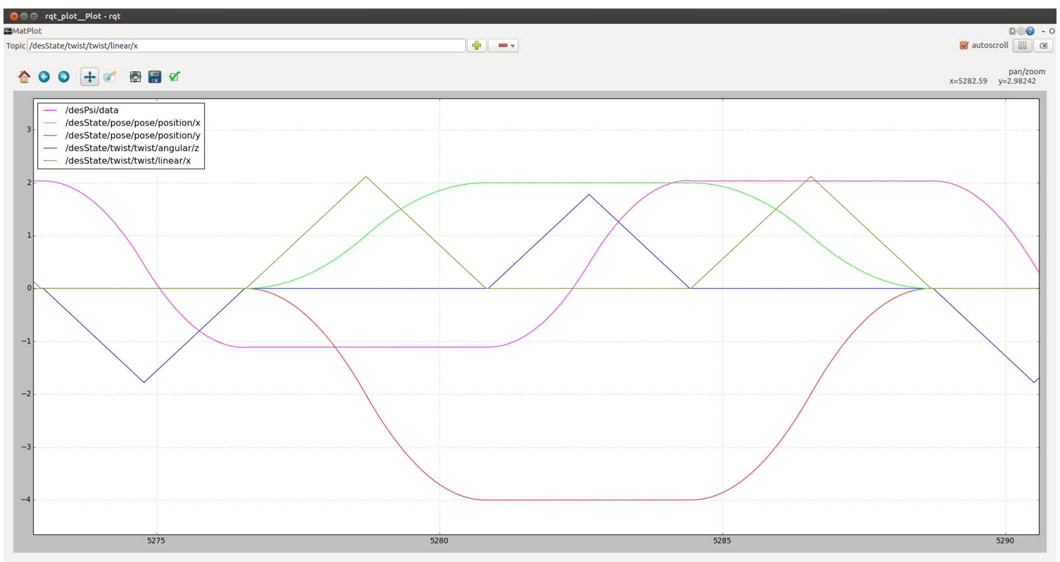


FIGURE 7.1: Example computed triangular velocity profile trajectories

A second example, shown in Fig 7.2 has specified start coordinates of $(0, 0)$ and goal coordinates of $(2, -4)$ with speed limits of 1.0 m/sec and 1.0 rad/sec. The resulting angular (blue) and translational (magenta) velocities have trapezoidal profiles, as expected. The corresponding x, y and ψ values are smooth, reach the desired goals, and satisfy the acceleration and velocity constraints.

7.1.3 Open-Loop Control

A simple approach to using computed trajectories is open-loop control. To do so, merely use the twist values of speed and spin rate from each desired state to command motion of a mobile robot via the `cmd_vel` topic. From `traj_builder_example_main`, the following

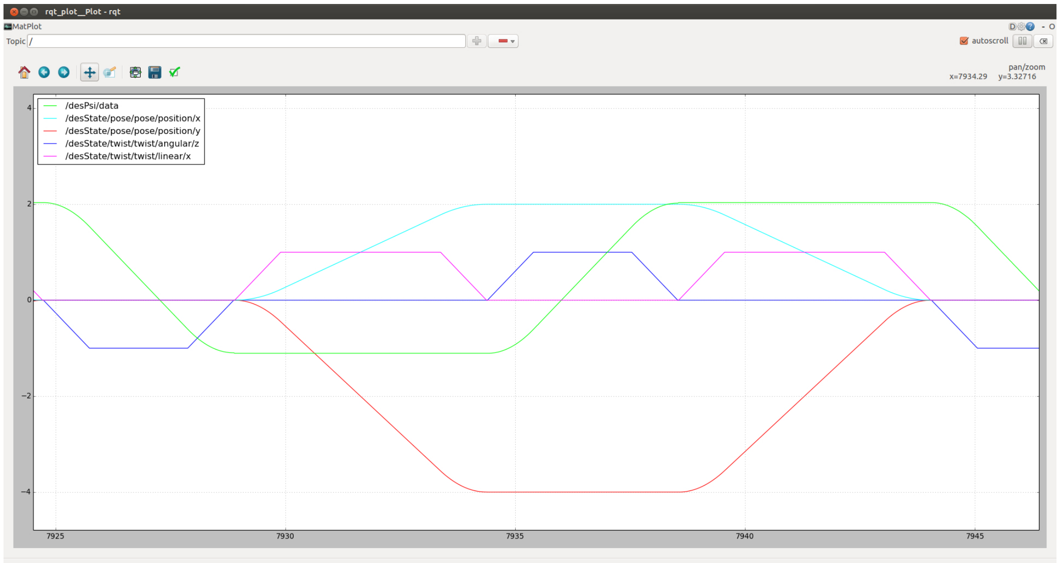


FIGURE 7.2: Example computed trapezoidal velocity profile trajectories

lines build a trajectory, step through the trajectory (at 20ms intervals), strip out the twist component from each desired state, and publish that twist to the `cmd_vel` topic.

```
trajBuilder.build_point_and_go_traj(g_start_pose, g_end_pose, vec_of_states);
 ROS_INFO("publishing desired states and open-loop cmd_vel");
 for (int i = 0; i < vec_of_states.size(); i++) {
     des_state = vec_of_states[i];
     des_state.header.stamp = ros::Time::now();
     des_state_publisher.publish(des_state);
     des_psi = trajBuilder.convertPlanarQuat2Psi(des_state.pose.pose.↩
         orientation);
     psi_msg.data = des_psi;
     des_psi_publisher.publish(psi_msg);
     twist_commander.publish(des_state.twist.twist); //FOR OPEN-LOOP CTL ONLY!
     looprate.sleep(); //sleep for defined sample period, then do loop again
 }
```

Note that publication of the twist values to `cmd_vel` is a special case. Commonly, the desired states should be published at the sample rate of the computed trajectories (as is done in the above code snippet), and these published states may be used for control. However, interpreting the desired states as open-loop commands and publishing to `cmd_vel` (via `twist_commander.publish(des_state.twist.twist);`) invokes open-loop control, which is seldom adequate.

Figure 7.3 shows an example result of commanding our model mobile robot (mobot) with open-loop commands. In this case, `traj_builder_example_main` was used to command the robot to go 5m back and forth along the x axis. The angular acceleration was set to a low value (0.1 rad/sec^2) to help improve pointing accuracy with slow turns. The computed trajectories include a trapezoidal velocity profile for forward motion (green) and triangular angular-velocity profiles for turning (blue). The corresponding desired y value remains 0 throughout the desired trajectory, and the desired x value has a smooth trajectory from 0 to 5m, then (after a 180-deg turn) back again from 5m to 0. An important feature of Fig 7.3 is the black trace, corresponding to the actual x value of the robot. Although the robot starts with perfect initial conditions ($x = 0$, $y = 0$, $\psi = 0$), it drifts off course

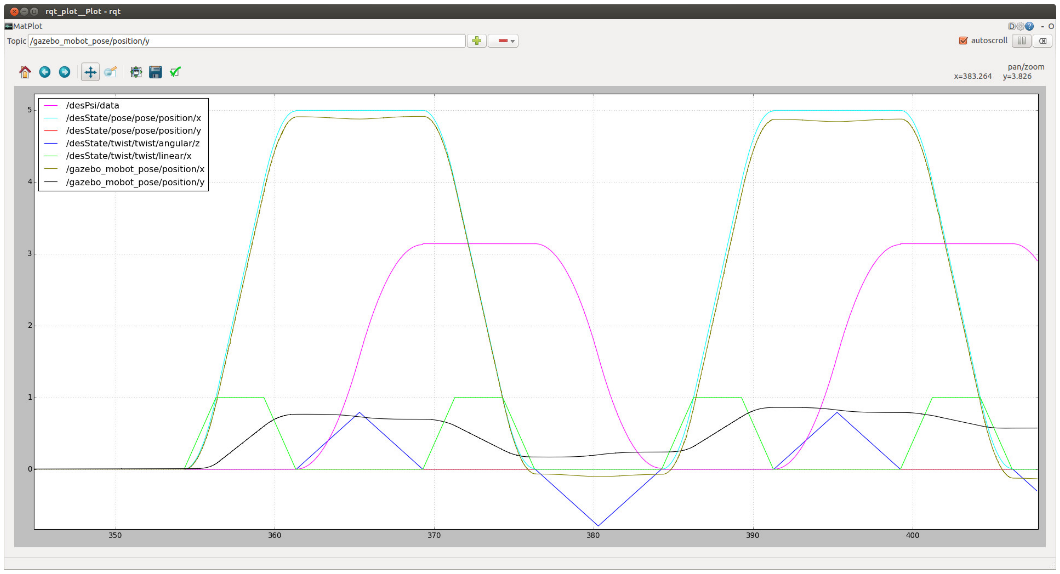


FIGURE 7.3: Example open-loop control of mobot. Intended motion is along x axis.

fairly quickly. Initially, the robot is commanded to move straight ahead, but the robot drifts to its left by approximately 0.8m. Although the robot model has no asymmetry, it still tends to drift to its left in simulation–presumably due to numerical imprecision in the gazebo physics simulation. Although such asymmetry was not intentionally modeled, such behavior is commonplace. A mobile robot will not steer perfectly straight with open-loop motion commands. The degree of drift will vary with robot properties (e.g. the robot mechanics, its electronic motor driver, its speed controller, uneven wear of the drive wheels and influence of the uncontrolled casters) as well as terrain properties (e.g. irregular or slippery surfaces).

Figure 7.4 shows the result of open-loop control under the same conditions, except that the maximum angular acceleration has been increased to 1.0 rad/sec^2. In Fig 7.4, the blue trace shows that the robot has drifted off course by roughly 3m. This emphasizes that use of open-loop control should be restricted to short travel distances and low accelerations, unless large path deviations are permissible (e.g. for random exploration).

7.1.4 Desired state publishing

The example node `traj_builder_example_main` illustrated use of the `traj_builder` library and how to publish desired states at a fixed rate. This example, however, was restrictive in that goal poses were hard coded. Further, there was no provision for responding to sensor or e-stop commands.

A more flexible node for publishing desired states is contained in the package `mobot_pub_des_state`. This package contains a class definition for `DesStatePublisher`, which is described in the header file `pub_des_state.h` and implemented in the file `pub_des_state.cpp`. In this example, `DesStatePublisher` is not a library (although it could be). Instead, the CMakeLists file compiles `pub_des_state.cpp` as a module together with `pub_des_state_main.cpp` with the instruction:

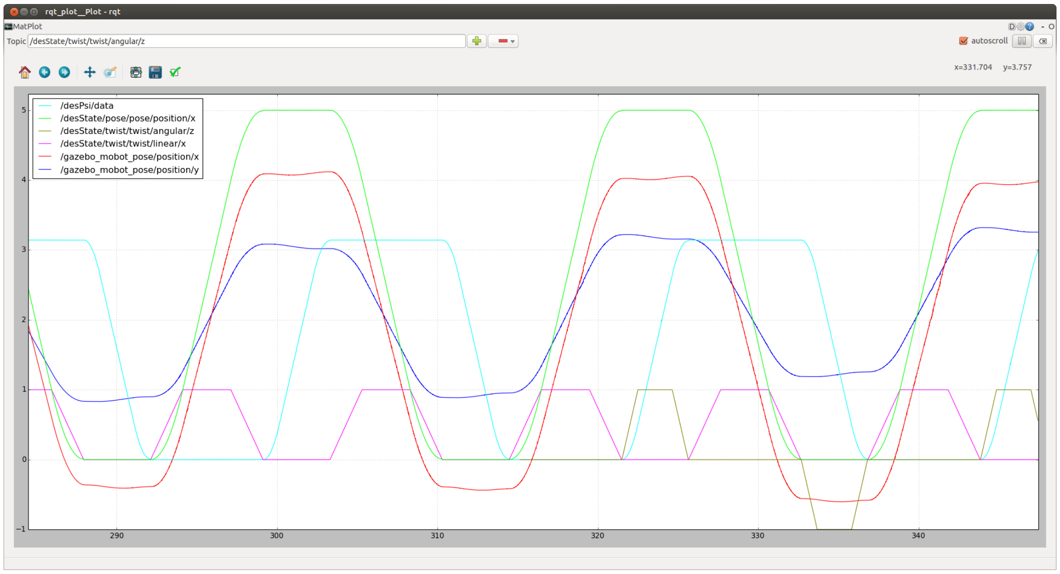


FIGURE 7.4: Example open-loop control of mobot with higher angular acceleration limit. Intended motion is along x axis.

```
cs_add_executable(mobot_pub_des_state src/pub_des_state_main.cpp src/pub_des_state.cpp)
```

The main program, `pub_des_state_main.cpp`, shown in Listing 7.5, is very brief.

Listing 7.5: Desired state publisher, main program

```
1  #include "pub_des_state.h"
2
3  int main(int argc, char **argv) {
4      ros::init(argc, argv, "des_state_publisher");
5      ros::NodeHandle nh;
6      //instantiate a desired-state publisher object
7      DesStatePublisher desStatePublisher(nh);
8      //dt is set in header file pub_des_state.h
9      ros::Rate looprate(1 / dt); //timer for fixed publication rate
10     desStatePublisher.set_init_pose(0,0,0); //x=0, y=0, psi=0
11     //put some points in the path queue--hard coded here
12     desStatePublisher.append_path_queue(5.0,0.0,0.0);
13     desStatePublisher.append_path_queue(0.0,0.0,0.0);
14
15     // main loop; publish a desired state every iteration
16     while (ros::ok()) {
17         desStatePublisher.pub_next_state();
18         ros::spinOnce();
19         looprate.sleep(); //sleep for defined sample period, then do loop again
20     }
21 }
```

This program includes the header file for the `DesStatePublisher` class, which contains all other necessary header files. This header also sets a value for "dt", which is the sample period that will be used to generate trajectory arrays of desired states. Line 10 sets the initial pose of the robot. In practice, this should be based on sensors that estimate the robot's initial pose in some reference frame (e.g. a map frame).

Lines 12-13 show how one can append goal poses to a path queue. The function `append_path_queue()` should be used rarely in practice. Rather, points should be added to the path queue via a service client of the service `append_path_queue_service`.

Lines 16-20 are the main loop of this node. Each iteration, the member method `pub_next_state()` of the `desStatePublisher` object is called, which causes this object to advance one step within its state machine. Typically, this results in accessing the next state from a computed trajectory and publishing this state to the topic "/desState," but several other behaviors can be invoked.

An object of class `DesStatePublisher` has four services. The `append_path_queue_service` expects a service request message as defined in the `mobot_pub_des_state` package. This service message, `path.srv`, contains a field "path" of type `nav_msgs/Path`. This message type contains a vector of poses. A service client can populate this vector of poses with any desired sequence of subgoals, then send it to the service `append_path_queue_service` to add the listed points to the end of the current path queue. An example program to do this is `pub_des_state_path_client.cpp`. This node creates a client of `append_path_queue_service`, defines 5 stamped poses, appends them one at a time to the vector of poses within the `nav_msgs/Path` message, then sends this message as a request to the append-path service.

With the `mobot_pub_des_state` node running, invoking `pub_des_state_path_client` sends 5 stamped poses to the desired-state publisher, which adds these poses as new subgoals to a C++ "queue" object of subgoals. An example result is shown in Figure 7.5. In this

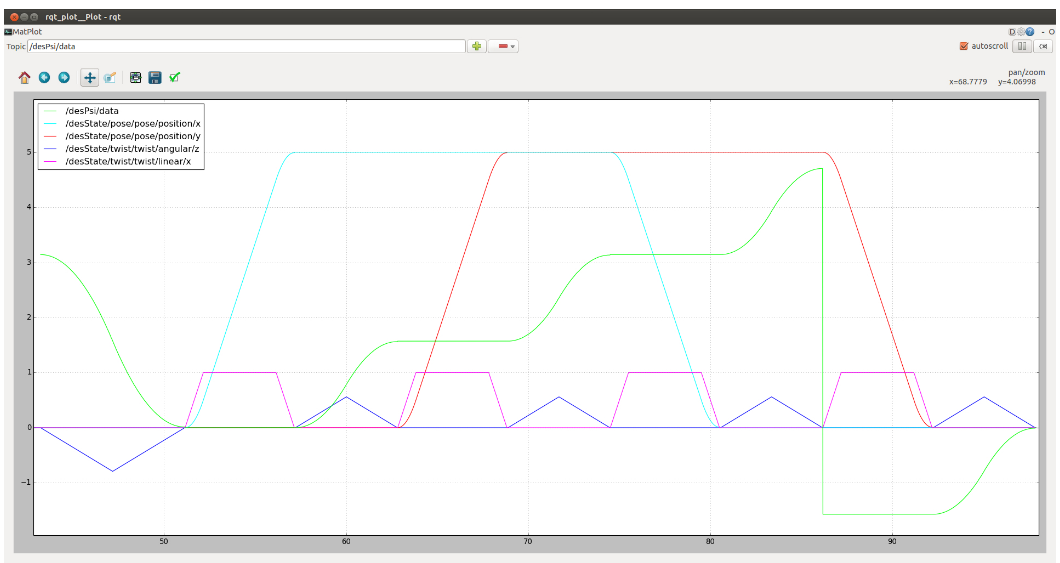


FIGURE 7.5: Trajectory generation for 5m x 5m square path.

figure, the desired effect can be seen. Starting from $(x, y) = (0, 0)$ and with orientation $\psi = 0$, the desired x velocity has a trapezoidal profile (magenta trace) resulting in a smooth trajectory of x values advancing from 0 to 5m (cyan trace). Once this position goal is reached, the desired states correspond to reorientation via a triangular angular-velocity profile (blue trace), resulting in the heading changing from 0 to $\pi/2$ (green trace). At this heading, the desired y coordinate is increased smoothly to 5m using a trapezoidal velocity profile. After reaching this subgoal, the heading is smoothly changed to π. The next advance brings the desired state to $(x, y) = (0, 5)$. This is followed by smooth motion to $(x, y) = (0, 0)$, then reorientation to $\psi = 0$. This shows how the desired-state publisher can receive pose subgoals and sequence through them with smooth trajectories that conform to dynamic limitations.

When the path queue within the desired-state publisher is empty, the desired-state publisher continues to publish the last command sent (which should be a zero-twist command). With the addition of new subgoals (poses) in the queue, the `DesStatePublisher` object will process each new pose as a vertex in a polyline using the `traj_builder` library. While this behavior is convenient, a few additional features are needed.

An important addition is the ability to respond to an emergency-stop signal. A service is provided for this: `estop_service`. One can send a trigger (message type `std_srvs::Trigger`) to this service, which will invoke an e-stop. Emergency-stop logic can be complex, as memory is required. Upon initiating an e-stop, the desired-state node should: compute a feasible deceleration profile; publish the states of this trajectory to completion; then hold the final state until the e-stop is reset. For e-stop reset, the desired-state publisher should plan a trajectory from the halted state to the next unattained subgoal, then invoke publications of this recovery trajectory.

Another important capability is the ability to clear out the current path queue. If the robot is halted, e.g. due to an e-stop, it is often the case that the original plan is no longer valid. Subgoals in the path queue may need to be flushed before new subgoals can be specified. The service `flush_path_queue_service` accepts a trigger (message type `std_srvs::Trigger`) that causes the current path queue to be cleared out.

The main function of the `DesStatePublisher` class is `pub_next_state()`, the contents of which are shown in Listing **??** (which is extracted from `pub_des_state.cpp`.

Listing 7.6: Key function, `pub_next_state()` from `pub_des_state.cpp`

```
void DesStatePublisher::pub_next_state() {
    // first test if an e-stop has been triggered
    if (e_stop_trigger_) {
        e_stop_trigger_ = false; //reset trigger
        //compute a halt trajectory
        trajBuilder_.build_braking_traj(current_pose_, des_state_vec_);
        motion_mode_ = HALTING;
        traj_pt_i_ = 0;
        npts_traj_ = des_state_vec_.size();
    }
    //or if an e-stop has been cleared
    if (e_stop_reset_) {
        e_stop_reset_ = false; //reset trigger
        if (motion_mode_ != E_STOPPED) {
            ROS_WARN("e-stop reset while not in e-stop mode");
        }
        //OK...want to resume motion from e-stopped mode;
        else {
            motion_mode_ = DONE_W_SUBGOAL; //this will pick up where left off
        }
    }

    //state machine; results in publishing a new desired state
    switch (motion_mode_) {
        case E_STOPPED: //this state must be reset by a service
            desired_state_publisher_.publish(halt_state_);
            break;

        case HALTING: //e-stop service callback sets this mode
            //if need to brake from e-stop, service will have computed
            // new des_state_vec_, set indices and set motion mode;
            current_des_state_ = des_state_vec_[traj_pt_i_];
            current_des_state_.header.stamp = ros::Time::now();
            desired_state_publisher_.publish(current_des_state_);
            current_pose_.pose = current_des_state_.pose.pose;
            current_pose_.header = current_des_state_.header;
```

```
            des_psi_ = trajBuilder_.convertPlanarQuat2Psi(current_pose_.pose.orientation);
            float_msg_.data = des_psi_;
            des_psi_publisher_.publish(float_msg_);

            traj_pt_i_++;
            //segue from braking to halted e-stop state;
            if (traj_pt_i_ >= npts_traj_) { //here if completed all pts of braking traj
                halt_state_ = des_state_vec_.back(); //last point of halting traj
                // make sure it has 0 twist
                halt_state_.twist.twist = halt_twist_;
                seg_end_state_ = halt_state_;
                current_des_state_ = seg_end_state_;
                motion_mode_ = E_STOPPED; //change state to remain halted
            }
            break;

        case PURSUING_SUBGOAL: //if have remaining pts in computed traj, send them
            //extract the i'th point of our plan:
            current_des_state_ = des_state_vec_[traj_pt_i_];
            current_pose_.pose = current_des_state_.pose.pose;
            current_des_state_.header.stamp = ros::Time::now();
            desired_state_publisher_.publish(current_des_state_);
            //next three lines just for convenience--convert to heading and publish
            // for rqt_plot visualization
            des_psi_ = trajBuilder_.convertPlanarQuat2Psi(current_pose_.pose.orientation);
            float_msg_.data = des_psi_;
            des_psi_publisher_.publish(float_msg_);
            traj_pt_i_++; // increment counter to prep for next point of plan
            //check if we have clocked out all of our planned states:
            if (traj_pt_i_ >= npts_traj_) {
                motion_mode_ = DONE_W_SUBGOAL; //if so, indicate we are done
                seg_end_state_ = des_state_vec_.back(); // last state of traj
                path_queue_.pop(); // done w/ this subgoal; remove from the queue
                ROS_INFO("reached a subgoal: x = %f, y= %f",current_pose_.pose.position.x,
                        current_pose_.pose.position.y);
            }
            break;

        case DONE_W_SUBGOAL: //suspended, pending a new subgoal
            //see if there is another subgoal is in queue; if so, use
            //it to compute a new trajectory and change motion mode

            if (!path_queue_.empty()) {
                int n_path_pts = path_queue_.size();
                ROS_INFO("%d points in path queue",n_path_pts);
                start_pose_ = current_pose_;
                end_pose_ = path_queue_.front();
                trajBuilder_.build_point_and_go_traj(start_pose_, end_pose_,des_state_vec_);
                traj_pt_i_ = 0;
                npts_traj_ = des_state_vec_.size();
                motion_mode_ = PURSUING_SUBGOAL; // got a new plan; change mode to pursue it
                ROS_INFO("computed new trajectory to pursue");
            } else { //no new goal? stay halted in this mode
                // by simply reiterating the last state sent (should have zero vel)
                desired_state_publisher_.publish(seg_end_state_);
            }
            break;

        default: //this should not happen
            ROS_WARN("motion mode not recognized!");
            desired_state_publisher_.publish(current_des_state_);
            break;
    }
```

The state-machine logic of function `pub_next_state()` is based on setting the `motion_mode_` member variable to one of the states: `E_STOPPED`, `DONE_W_SUBGOAL`, `PURSUING_SUBGOAL` or `HALTING`. The state transition possibilities are as follows.

If an e-stop trigger is received (via the service `estop_service`), the the flag e_stop_trigger_ is set to "true." As a result, lines 3-10 of pub_next_state() do: reset the e-stop trigger; use the trajectory-builder object to compute a braking trajectory and put this trajectory in des_state_vec_; set motion_mode_ to mode HALTING; set the member variable npts_traj_ to the number of states in the braking trajectory; and initialize the state index traj_pt_i_ to 0. The state machine will then be in mode HALTING, prepared to perform planned braking.

A complement to triggering an e-stop is recovering from an e-stop. An e-stop is cleared via the service clear_estop_service, which sets the value of e_stop_reset_ to true. Lines 12-21 deal with this case. The e_stop_reset_ trigger is reset, and the motion mode is set to mode DONE_W_SUBGOAL. The effect of this state change is that the pub_next_state() function will treat this condition as though the robot had been halted because it was out of goals. Subsequently, if there is at least one goal in the queue, it will be processed according to the DONE_W_SUBGOAL logic. Note that an e-stop does not remove the current subgoal from the queue, so resetting an e-stop can resume with a plan to reach this unattained pose. However, it might also be the case that the service flush_path_queue_service has emptied the queue during the e-stop condition (if higher-level code finds this appropriate), in which case the most recent subgoal will be forgotten.

After checking the status of reset triggers, pub_next_state() enters a switch/case block to handle the various motion-mode cases.

The case (motion-mode state) HALTING (lines 29-51) provides the logic for ramping down velocities to a halt. The member variables current_des_state_ and current_pose_ are updated by using index traj_pt_i_ into the array (vector) des_state_vec_. Note that the trajectory in this vector will have been computed for braking due to an e-stop trigger. The extracted desired state is published to /desState. In addition, for debug and visualization purposes, lines 37-39 compute the desired scalar heading and publish this to topic desPsi. The index into the vector of desired states is then incremented and tested. When this index reaches the last value in the vector of desired states, lines 43-50 set current_des_state_ to the last state in the braking trajectory (for use in e-stop recovery), set an equivalent halt_state_ (with enformed zero twist), and set the motion mode to E_STOPPED.

For case E_STOPPED, the halt_state_ is repeatedly published. The E_STOPPED mode does not advance the state machine. This is a terminal state, until/unless an e-stop reset condition is received (which results in the motion mode changing to DONE_W_SUBGOAL).

The case PURSUING_SUBGOAL is similar to HALTING. This mode steps through states from des_state_vec_, publishing them to the desState topic, updating current_des_state_, computing and publishing the scalar heading, and incrementing the index into current_des_state_ (lines 53-64). When the last point in the plan is reached, lines 66-71 perform the following. The motion mode is changed to DONE_W_SUBGOAL, the last state commanded is saved in seg_end_state_, and the current (achieved) subgoal is removed (popped) from the path queue.

Finally, the motion-mode state DONE_W_SUBGOAL prepares a plan for achieving the next subgoal (lines 75-93). If there is at least one subgoal in the path queue (line 79), then a new trajectory plan is computed using the trajectory-builder library (lines 82-84), and the trajectory index is reset to zero, number of trajectory points in the plan is set, and the motion mode is changed to PURSUING_SUBGOAL for execution of the plan.

If there are no points in the path queue, the DONE_W_SUBGOAL case republishes the end state from the previously-achieved subgoal. (This is initialized to the start state within the constructor of DesStatePublisher.

This code provides a minimal set of capabilities for executing plans and responding to unexpected conditions. It is up to higher-level code to construct intelligent plans and to invoke halts as appropriate (e.g. based on sensation of potential collision or unsafe terrain).

At present, this code is limited to polyline paths. A useful extension would be to handle curved or spline paths to achieve more graceful and more efficient navigation.

An additional node, `open_loop_controller`, invokes open-loop control from publications of desired states. This node simply subscribes to the `desState` topic, strips off the twist term, and republishes this to `cmd_vel`.

These nodes can be run as follows:

- `roslaunch gazebo_ros empty_world.launch` starts gazebo
- `roslaunch mobot_urdf mobot.launch` loads the mobot model
- `rosrun mobot_pub_des_state open_loop_controller` starts the open-loop controller
- `rosrun mobot_pub_des_state mobot_pub_des_state` starts the desired-state publisher
- `rosrun mobot_pub_des_state pub_des_state_path_client` sends a square path request to the desired-state publisher

In addition to running prescribed paths, the following services can be invoked:

- `rosservice call estop_service`
- `rosservice call clear_estop_service`
- `rosservice call clear_estop_service`

Note, though, that the e-stop behavior is not fully functional, since the braking-trajectory function within TrajBuilder is not implemented.

The code presented here can be used for open-loop control to execute prescribed paths. However, open-loop control is seldom adequate. Often, a robot must navigate with precision, e.g. to pass through doorways, to dock with a charger or to approach a workstation within a pose tolerance, or to stay within a lane on a highway. Open-loop control only uses twist commands an makes no reference to the pose within each desired state. To take advantage of the pose component of desired state, the robot needs to know its actual pose in space. By comparing actual pose to desired pose, a steering algorithm can improve navigation precision. The subject of state estimation is introduced next.

7.2 ROBOT STATE ESTIMATION

Use of open-loop control results in drift from the intended path, which accumulates with distance traveled. Open-loop control is thus unsuitable for traveling significant distances or when there are demanding tolerances on path following (e.g., navigating through a narrow doorway). For more precise and reliable path following, a feedback steering algorithm is used, which requires reference to both desired states and to estimated actual states of the system. State estimation is introduced in this section, starting with simple odometry and extending to use of GPS and LIDAR.

7.2.1 Getting Model State from Gazebo

For development and analysis purposes only, it can be useful to get model states from gazebo. One should be careful, however, not to rely on this information for code to be deployed on real robots. Gazebo has omniscient awareness of system states, since gazebo itself computes these states. On a physical system, however, the state of a system with respect to the world is unknown. Considerable effort is often required to estimate the state of a system in the world.

For the purpose of evaluating system performance, one can consult the gazebo-published system state and use this information to compute the difference between estimated system state and the "actual" system state (as computed by gazebo). One can use gazebo's published model states for developing steering algorithms, although it is important that the algorithms be tolerant of the non-idealities of state estimators. The ideal system state from Gazebo can also be sampled, modified and republished to emulate realistic absolute sensors, such as Global Positioning Systems. By adding appropriate levels of noise to the gazebo states, one can create a virtual sensor that is useful for development of localization algorithms.

Starting our mobot simulation with:

```
roslaunch gazebo_ros empty_world.launch
```

```
roslaunch mobot_urdf mobot.launch
```

we can see the topics published, which includes `gazebo/model_states`. This topic carries messages of type `gazebo_msgs/ModelStates`. A `rosmsg show gazebo_msgs/ModelStates` displays the format of this message type:

```
string[] name
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
geometry_msgs/Twist[] twist
  geometry_msgs/Vector3 linear
    float64 x
    float64 y
```

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

This message type contains a vector of names, a vector of poses and a vector of twists. The message is interpreted as follows: the order in which a model name appears in the `name` array is the same as the order in which the respective pose and twist appear in the pose and twist arrays. A `rostopic echo gazebo/model_states` example output is:

```
name: ['ground_plane', 'mobot']
pose:
  -
    position:
      x: 0.0
      y: 0.0
      z: 0.0
    orientation:
      x: 0.0
      y: 0.0
      z: 0.0
      w: 1.0
  -
    position:
      x: 0.00080270286594
      y: 0.170554714089
      z: -1.95963022377e-05
    orientation:
      x: -1.33139494634e-05
      y: 0.0020008448071
      z: 0.0033807500955
      w: 0.999992283456
twist:
  -
    linear:
      x: 0.0
      y: 0.0
      z: 0.0
    angular:
      x: 0.0
      y: 0.0
      z: 0.0
  -
    linear:
      x: 2.66486083024e-05
      y: 0.000452450930556
      z: -0.00510658997482
    angular:
      x: 5.29981190091e-05
      y: -0.00334270636901
      z: -3.15214058467e-06
```

From the echo, we see that gazebo is publishing the states of only two models: `ground_plane` and `mobot`. The ground plane simply has (identically) zero pose and zero velocity, and these properties will remain constant, since the ground plane is statically joined to the world frame. The second model on the list, mobot, is the model we care about. By accessing the second pose and the second twist in the respective arrays, we can obtain gazebo's claim of the robot's state.

Inconveniently, one cannot assume that the mobot will always be the second model within the arrays of models states published by gazebo. One must identify the index location of the desired model to access the corresponding state information.

Listing 7.7 shows the code in `mobot_gazebo_state.cpp`, which is contained in a package of the same name within the accompanying learning-ros repository. Lines 21-29 search through the list of names for a match to "mobot". If a match is found, the corresponding array index is contained in `imodel`. Lines 31-32 access the corresponding pose and republish this on the topic `gazebo_mobot_pose`. Lines 33-37 copy the ideal pose to the object `g_noisy_mobot_pose`. The x and y components of the pose are corrupted with Gaussian random noise, where the mean (0) and standard deviation (1.0) are set on line 14. The heading of `g_noisy_mobot_pose` is suppressed (line 34), and the resulting corrupted (partial) pose is published to topic `gazebo_mobot_noisy_pose`. By adding noise to the ideal pose information, one can make a more realistic emulation of a GPS source. Such sources do not have drift, but position information does contain noise. Further, heading information from GPS is generally not trustworthy. For the emulated GPS, the heading information has been suppressed, which enforces that code developed based on this simulated signal cannot accidentally rely on information that is expected to be poor in practice. The noisy GPS emulator signal from this node will be used later to show how to compute localization that uses GPS.

The topic `gazebo_mobot_pose` contains ideal position and orientation data. It can be used to develop steering algorithms and to compare localization results with the "correct" answer (which is typically unknown in practice).

Listing 7.7: Code to get mobot model states from gazebo and republish both as ideal states and as noisy states. Can be used to emulate GPS

```
1  #include <ros/ros.h>
2  #include <gazebo_msgs/ModelStates.h>
3  #include <geometry_msgs/Pose.h>
4  #include <string.h>
5  #include <stdio.h>
6  #include <math.h>
7  #include <random>
8
9  geometry_msgs::Pose g_mobot_pose; //this is the pose of the robot in the world, ↩
       according to Gazebo
10 geometry_msgs::Pose g_noisy_mobot_pose; //added noise to x,y, and suppress orientation
11 geometry_msgs::Quaternion g_quat;
12 ros::Publisher g_pose_publisher;
13 ros::Publisher g_gps_publisher;
14 std::normal_distribution<double> distribution(0.0,1.0); //args: mean, std_dev
15 std::default_random_engine generator;
16 void model_state_CB(const gazebo_msgs::ModelStates& model_states)
17 {
18   int n_models = model_states.name.size();
19   int imodel;
20   //ROS_INFO("there are %d models in the transmission",n_models);
21   bool found_name=false;
22   for (imodel=0;imodel<n_models;imodel++) {
23     std::string model_name(model_states.name[imodel]);
24     if (model_name.compare("mobot")==0) {
25       //ROS_INFO("found match: mobot is model %d",imodel);
26       found_name=true;
27       break;
28     }
29   }
30   if(found_name) {
```

```
    g_mobot_pose= model_states.pose[imodel];
    g_pose_publisher.publish(g_mobot_pose);
    g_noisy_mobot_pose = g_mobot_pose;
    g_noisy_mobot_pose.orientation = g_quat;
    g_noisy_mobot_pose.position.x += distribution(generator);
    g_noisy_mobot_pose.position.y += distribution(generator);
    g_gps_publisher.publish(g_noisy_mobot_pose); //publish noisy values
    //double randval = distribution(generator);
    //ROS_INFO("randval =%f",randval);
    }
  else
    {
      ROS_WARN("state of mobot model not found");
    }
}

int main(int argc, char **argv) {
    ros::init(argc, argv, "gazebo_model_publisher");
    ros::NodeHandle nh;

    g_pose_publisher= nh.advertise<geometry_msgs::Pose>("gazebo_mobot_pose", 1);
    g_gps_publisher = nh.advertise<geometry_msgs::Pose>("gazebo_mobot_noisy_pose", 1);
    ros::Subscriber state_sub = nh.subscribe("gazebo/model_states",1,model_state_CB);
    //suppress the orientation output for noisy state; fill out a legal, constant quaternion
    g_quat.x=0;
    g_quat.y=0;
    g_quat.z=0;
    g_quat.w=1;
    ros::spin();
}
```

The gazebo-state publisher node can be started with:

```
rosrun mobot_gazebo_state mobot_gazebo_state
```

For the mobot following a 5x5m desired trajectory under open-loop control, the resulting republished gazebo-model state is shown in Fig 7.6.

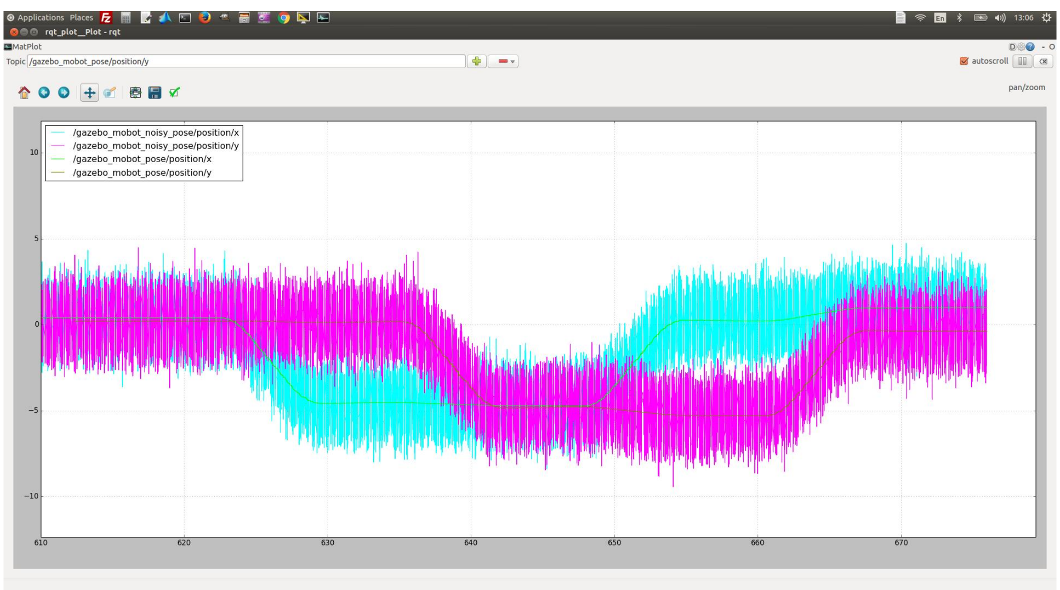


FIGURE 7.6: Republished mobot gazebo state and noisy state with robot executing a square trajectory under open-loop control

7.2.2 Odometry

When creating a ROS interace to a vehicle, there are two essential needs: a means to command motion (e.g. via `cmd_vel`) and a means to report back estimated state (typically via the topic `odom`). For our differential-drive vehicle, we assume that the two wheels are independently controllable via left and right-wheel velocity servos, and that these wheels each have some type of encoder capable of providing incremental wheel angles (or, alternatively, wheel velocities). From speed and spin commands, we need to derive corresponding left and right-wheel joint commands, and from measured wheel motions, we need to derive and publish updates of estimated pose and twist of the robot.

A differential mapping between incremental wheel motions, $d\theta_l$ and $d\theta_r$, and corresponding incremental pose updates can be derived with the help of Fig 7.7.

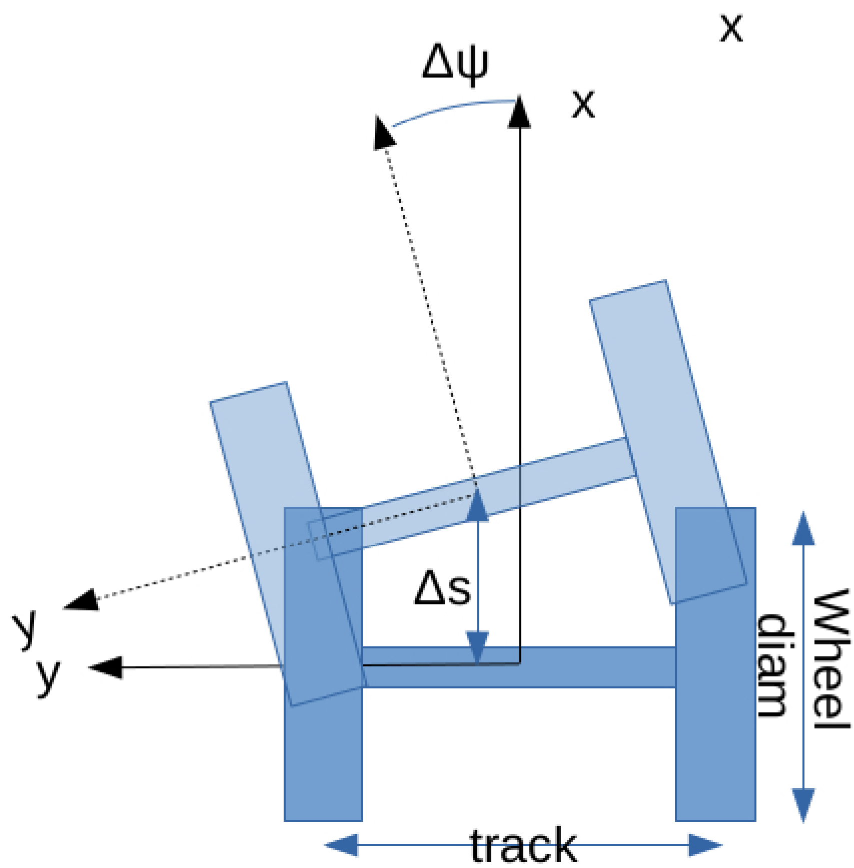


FIGURE 7.7: Differential-drive kinematics: incremental wheel rotations yield incremental pose changes

With reference to Fig 7.7, we specify a coordinate frame on the vehicle with origin between the drive wheels, x axis pointing forward and y axis pointing to the left. With incremental motions of the left and right wheels, the robot will advance its frame origin forward by amount Δs and will change its heading by amount $\Delta\psi$. The relation depends on the wheel diameter, D, and the "track" (distance between the wheels), T. (It is implicitly assumed that the left and right wheel diameters are equal, although this will not be precisely true). The relations are:

$$\Delta s = \frac{1}{4}D(d\theta_l + d\theta_r) = K_s(d\theta_l + d\theta_r) \tag{7.4}$$

where $d\theta_l$ and $d\theta_r$ are in radians, and s and D are in meters. The change in heading is

$$\Delta\psi = \frac{1}{2}D(d\theta_r - d\theta_l)/T = K_\psi(d\theta_r - d\theta_l) \tag{7.5}$$

If at time t the estimate of pose is $(\tilde{x}(t), \tilde{y}(t), \tilde{\psi}(t))$, and if at time $t + dt$ incremental motions of the left and right wheels are $d\theta_l$ and $d\theta_r$, then the updated pose at time $t + dt$ is approximately:

$$\begin{bmatrix} \tilde{x}(t+dt) \\ \tilde{y}(t+dt) \\ \tilde{\psi}(t+dt) \end{bmatrix} = \begin{bmatrix} \tilde{x}(t) \\ \tilde{y}(t) \\ \tilde{\psi}(t) \end{bmatrix} + \begin{bmatrix} K_s(d\theta_l + d\theta_r)cos(\tilde{\psi}) \\ K_s(d\theta_l + d\theta_r)sin(\tilde{\psi}) \\ K_\psi(d\theta_r - d\theta_l) \end{bmatrix} \tag{7.6}$$

The robot's twist can be computed as well, which is:

$$\begin{bmatrix} \tilde{v_x} \\ \tilde{v_y} \\ \tilde{\omega_z} \end{bmatrix} = \begin{bmatrix} K_s(d\theta_l + d\theta_r)cos(\tilde{\psi})/dt \\ K_s(d\theta_l + d\theta_r)sin(\tilde{\psi})/dt \\ K_\psi(d\theta_r - d\theta_l)/dt \end{bmatrix} \tag{7.7}$$

An implementation of these equations is in `mobot_drifty_odom.cpp`, which is in a package of the same name. The source code appears in Listing 7.8 (as well as in the accompanying code repository).

This node uses the parameters `TRACK` (line 13) and two wheel radii (lines 10-11) in the computation of state updates. These values agree with the mobot URDF specifications, but they can be altered to simulate the effects of imperfect odometry computations.

Wheel-joint increments used in pose and twist updates are based on subscription to the topic `joint_states` (line 125). The joint-state data (specifically, the rotation angles of the left and right wheels) is checked every 10ms (as set by the loop timer on line 128).

Lines 53-71 of the joint-state subscriber callback test that the data is good and that both wheel-joint topics are present. If the wheel-joint data is determined to be valid, the pose and twist estimates are updated via lines 73-86.

Listing 7.8: Code to compute and publish odom from incremental wheel-joint rotations. Can introduce errors to simulate larger drift effects.

```
1  #include <ros/ros.h>
2  #include <geometry_msgs/Pose.h>
3  #include <sensor_msgs/JointState.h>
4  #include <nav_msgs/Odometry.h>
5  #include <string.h>
6  #include <stdio.h>
7  #include <math.h>
8
9  //from URDF:  <xacro:property name="tirediam" value="0.3302" />
10 const double R_LEFT_WHEEL = 0.3302/2.0;
11 const double R_RIGHT_WHEEL = R_LEFT_WHEEL; //+0.005; //introduce error--tire diam diff
12 //from URDF: <xacro:property name="track" value=".56515" />
13 const double TRACK = 0.56515; //0.560; // track error
14
15 const double wheel_ang_sham_init= -1000000.0;
16 bool joints_states_good=false;
17
18 nav_msgs::Odometry g_drifty_odom;
```

```
sensor_msgs::JointState g_joint_state;
ros::Publisher g_drifty_odom_pub;
ros::Subscriber g_joint_state_subscriber;

double g_new_left_wheel_ang,g_old_left_wheel_ang;
double g_new_right_wheel_ang, g_old_right_wheel_ang;
double g_t_new,g_t_old,g_dt;
ros::Time g_cur_time;
double g_odom_psi;

geometry_msgs::Quaternion convertPlanarPsi2Quaternion(double psi) {
    geometry_msgs::Quaternion quaternion;
    quaternion.x = 0.0;
    quaternion.y = 0.0;
    quaternion.z = sin(psi / 2.0);
    quaternion.w = cos(psi / 2.0);
    return (quaternion);
}


void joint_state_CB(const sensor_msgs::JointState& joint_states)
{
  double dtheta_right, dtheta_left,ds, dpsi;
  int n_joints = joint_states.name.size();
  int ijnt;
  int njnts_found=0;
  bool found_name=false;
  g_old_left_wheel_ang = g_new_left_wheel_ang;
  g_old_right_wheel_ang = g_new_right_wheel_ang;
  g_t_old = g_t_new;
  g_cur_time = ros::Time::now();
  g_t_new = g_cur_time.toSec();
  g_dt = g_t_new-g_t_old;

  for (ijnt=0;ijnt<n_joints;ijnt++) {
    std::string joint_name(joint_states.name[ijnt]);
    if (joint_name.compare("left_wheel_joint")==0) {
        g_new_left_wheel_ang = joint_states.position[ijnt];
        njnts_found++;
    }
    if (joint_name.compare("right_wheel_joint")==0) {
        g_new_right_wheel_ang = joint_states.position[ijnt];
        njnts_found++;
    }
  }
  if (njnts_found<2) ROS_WARN("did not find both wheel joint angles!");
  if (!joints_states_good) {
      if (g_new_left_wheel_ang > wheel_ang_sham_init/2.0) {
          joints_states_good=true; //passed the test
          g_old_left_wheel_ang = g_new_left_wheel_ang;
          g_old_right_wheel_ang = g_new_right_wheel_ang; //assume right is good now as well
      }
  }
  if (joints_states_good) { //only compute odom if wheel angles are valid
        dtheta_left = g_new_left_wheel_ang - g_old_left_wheel_ang;
        dtheta_right = g_new_right_wheel_ang - g_old_right_wheel_ang;
        ds = 0.5*(dtheta_left*R_LEFT_WHEEL + dtheta_right*R_RIGHT_WHEEL);
        dpsi = dtheta_right*R_RIGHT_WHEEL/TRACK - dtheta_left*R_LEFT_WHEEL/TRACK;
        g_drifty_odom.pose.pose.position.x += ds*cos(g_odom_psi);
        g_drifty_odom.pose.pose.position.y += ds*sin(g_odom_psi);
        g_odom_psi+= dpsi;
        //ROS_INFO("dthetal, dthetar, dpsi, g_odom_psi = %f, %f %f, %f",dtheta_left,dtheta_right,dpsi,g_odom_psi);
        g_drifty_odom.pose.pose.orientation = convertPlanarPsi2Quaternion(g_odom_psi);

        g_drifty_odom.twist.twist.linear.x = ds/g_dt;
        g_drifty_odom.twist.twist.angular.z = dpsi/g_dt;
        g_drifty_odom.header.stamp=g_cur_time;
        g_drifty_odom_pub.publish(g_drifty_odom);
  }
}



int main(int argc, char **argv) {
```

```
93      ros::init(argc, argv, "drifty_odom_publisher");
94      ros::NodeHandle nh;
95      //inits:
96      g_new_left_wheel_ang=wheel_ang_sham_init;
97      g_old_left_wheel_ang=wheel_ang_sham_init;
98      g_new_right_wheel_ang=wheel_ang_sham_init;
99      g_old_right_wheel_ang=wheel_ang_sham_init;
100     g_cur_time=ros::Time::now();
101     g_t_new=g_cur_time.toSec();
102     g_t_old=g_t_new;
103 
104     //initialize odom with pose and twist defined as zero at start-up location
105     g_drifty_odom.child_frame_id="base_link";
106     g_drifty_odom.header.frame_id="odom";
107     g_drifty_odom.header.stamp = g_cur_time;
108     g_drifty_odom.pose.pose.position.x = 0.0;
109     g_drifty_odom.pose.pose.position.y = 0.0;
110     g_drifty_odom.pose.pose.position.z = 0.0;
111     g_drifty_odom.pose.pose.orientation.x = 0.0;
112     g_drifty_odom.pose.pose.orientation.y = 0.0;
113     g_drifty_odom.pose.pose.orientation.z = 0.0;
114     g_drifty_odom.pose.pose.orientation.w = 1.0;
115 
116     g_drifty_odom.twist.twist.linear.x = 0.0;
117     g_drifty_odom.twist.twist.linear.y = 0.0;
118     g_drifty_odom.twist.twist.linear.z = 0.0;
119     g_drifty_odom.twist.twist.angular.x = 0.0;
120     g_drifty_odom.twist.twist.angular.y = 0.0;
121     g_drifty_odom.twist.twist.angular.z = 0.0;
122     ros::Rate timer(100.0); // a 100Hz timer
123 
124     g_drifty_odom_pub = nh.advertise<nav_msgs::Odometry>("drifty_odom", 1);
125     g_joint_state_subscriber = nh.subscribe("joint_states",1,joint_state_CB);
126     while(ros::ok()) {
127         ros::spinOnce();
128         timer.sleep();
129     }
130 }
```

The `mobot_drifty_odom` can be started: The

```
rosrun mobot_drifty_odom mobot_drifty_odom
```

together with the `mobot_gazebo_state` node to test the performance of the odometry state estimator. If the robot is started from the home pose, (0,0,0), then it keeps track of robot state quite well, as shown in Figure 7.8. In Fig 7.8, the odometry values for x and y closely follow the gazebo (ideal) states for x and y. The path tracking is not very good, since the trapezoids of x and y should reach and track horizontal values identically equal to 0m or 5m. The problem with the path following is that the differential-drive controller (subject to friction, torque saturation, inertial effects and controller bandwidth limitations) is imperfect, and the commanded speeds are not achieved identically. To improve on the path following, it is necessary to use a feedback steering controller, which will be introduced in Section 7.3.

More realistic odometry can be simulated by altering parameters of the kinematic model. As an example, line 11 of Listing **??** was changed to: The

```
const double R_RIGHT_WHEEL = R_LEFT_WHEEL+0.005;
```

Adding 5mm to the assumed wheel diameter is equivalent to having a modeling error that fails to recognize a tire wear differential of 5mm. When `mobot_drifty_odom` is run with the same 5mx5m square path command, the odometry estimates of pose diverge badly from the gazebo-reported poses, as shown in Fig 7.9. Odometry drift is a significant problem that can only be corrected with reference to some additional sensing, e.g. LIDAR sensing of the environment or GPS signals. Nonetheless, the odometry signal is very useful, particularly when combined with additional sensors.

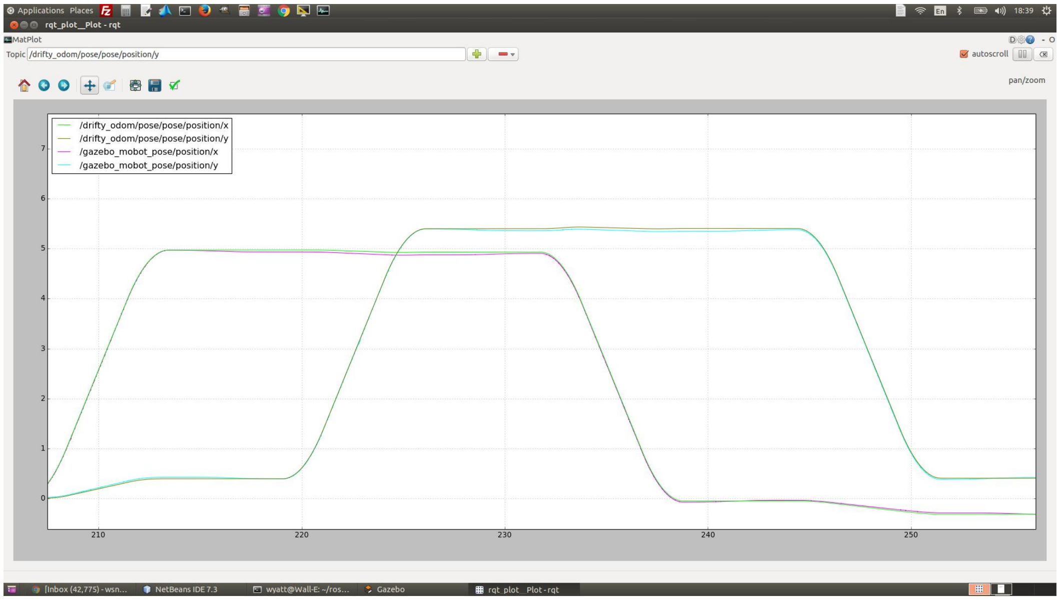


FIGURE 7.8: Comparison of ideal state, per gazebo, vs estimated state, per odom, while the simulated robot is following a 5mx5m square path

---

Often, the robot will have an inertial measurement unit (IMU), and this additional sensory information can be folded into the odometry estimate. An IMU is insensitive to some sources of error in estimating pose from wheel kinematics alone, including effects of slipping or skidding, uneven terrain, imprecision of kinematic parameters (e.g. wheel diameters and track), and unmeasured backlash in the drivetrain. On the other hand, one must be careful with bias offsets in IMU signals, which can cause odometry estimates to integrate to infinity, even when the robot is standing still.

When creating a ROS interface to a new mobile platform, code similar to `mobot_drifty_odom` should be written (unless such an "driver" already exists, which is increasingly common). The result of this node is publication of estimated state to the topic `odom`. For the STDR robot, odom messages are published, although they are merely identical to the equivalent `cmd_vel` inputs, without consideration of realistic dynamics. The mobot simulation in gazebo uses a differential-drive plug-in, which includes the equivalent of Listing **??** within the plug-in.

A second node that is needed to interface to a mobile platform is one that acts on input commands on the `vel_cmd` topic. This node should convert speed/spin commands into right-wheel and left-wheel velocity commands. Conversion from speed/spin to wheel joint commands can be computed as follows. For known sample period, dt, equations 7.4 and 7.5 can be re-written as:

$$\begin{bmatrix} ds/dt \\ d\psi/dt \end{bmatrix} = \begin{bmatrix} K_s & K_s \\ K_\psi & -K_\psi \end{bmatrix} \begin{bmatrix} d\theta_r/dt \\ d\theta_l/dt \end{bmatrix} \tag{7.8}$$

which can be inverted to yield:

$$\begin{bmatrix} d\theta_r/dt \\ d\theta_l/dt \end{bmatrix} = \frac{2}{K_s K_\psi} \begin{bmatrix} K_\psi & K_s \\ K_\psi & -K_s \end{bmatrix} \begin{bmatrix} ds/dt \\ d\psi/dt \end{bmatrix} \tag{7.9}$$

Equation may be used to convert incoming speed (ds/dt) and spin ($d\psi/dt$) commands from

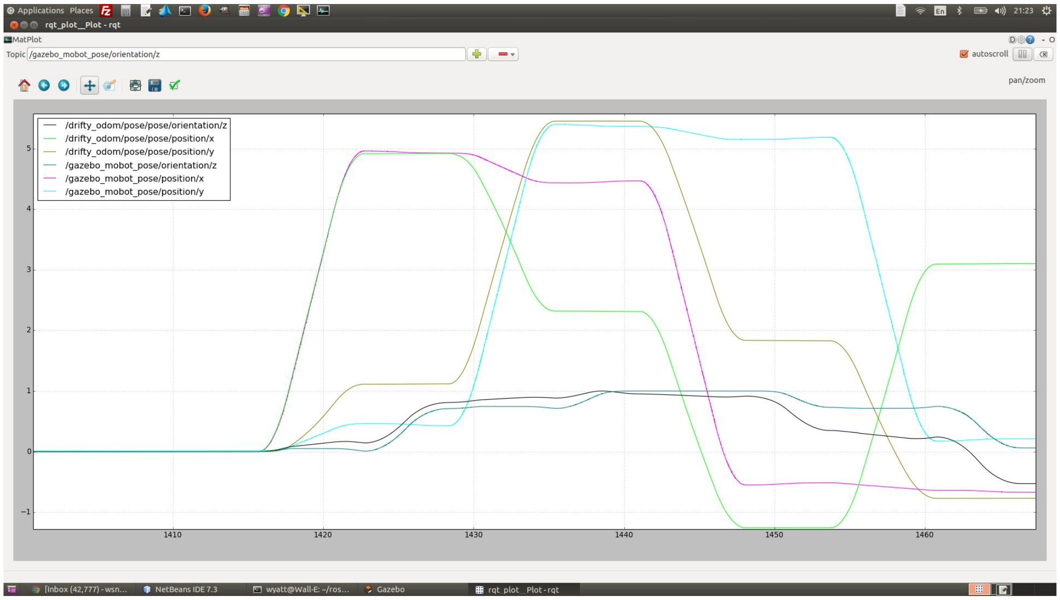


FIGURE 7.9: Ideal state vs odom state estimates for 5mx5m path following. Odom as 5mm wheel diameter error of right wheel vs left wheel

---

the `cmd_vel` topic into corresponding right and left-wheel speed commands, $d\theta_r/dt$ and $d\theta_l/dt$. A local velocity controller should enforce that the commanded wheel speeds are achieved. Inevitably, there will be errors between the commanded wheel velocities and the realized wheel velocities. Odometry thus should be based on measured wheel increments rather than the assumption that the wheel-speed commands are achieved precisely. (Although the latter approach can be taken if necessary, as is done with the STDR simulator).

As noted, computation of odometry suffers from modeling imperfections, leading to accumulation of errors that can be arbitrarily large. Addressing this issue requires use of additional sensors. A second problem with odometry is its choice of reference frame.

For both the pose and the twist, the estimated values correspond to the robot's base frame expressed with respect to some ground reference frame. The reference frame, however is ambiguous. By default, for odometry computations, the reference frame pose of $(0, 0, 0)$ corresponds to whatever pose the robot is in when the odometry node begins its computations. This pose (which is different every time the robot's odometry node starts up) is known as the "odometry frame."

The approach in ROS to addressing odometry drift is somewhat odd but pragmatic. Two common absolute reference frames are "world" (e.g., lattitude, longitude and heading) and "map", for which map coordinates can be stated with respect to defined reference frame on the map (a definition of $(x, y) = (0, 0)$). Often, the robot base that is computing and publishing its odom information does not have access to other sensors that could help establish the robot's absolute pose. Instead, the base continues to publish its odom signals that are increasingly bad over time and distance travelled. Soon, the pose information from this topic is seemingly worthless. However, it the odom signal is still valuable, if combined properly with additional sensors. In ROS, this is done with the computation of a transform from the "odom" frame to the "world" (or "map") frame.

Consider a world frame. There is a "correct" value of state, (x, y, ψ), in the world (whether or not these values are known). The pose of the robot with respect to the world can be represented as a transform, as introduced in Section . A transform $T_{odom/world}$ can be used to express the position and orientation of the "odom" frame with respect to the world frame. At start-up, the odom publisher node assumes that its initial pose in the world is $(x, y, \psi) = (0, 0, 0)$. In world coordinates, these values may be referred to as $(x_{world}, y_{world}, \psi_{world})$. Equivalently, these values may be expressed as equivalent components within $T_{odom/world}$. As the robot moves and odometry information is updated, the state estimates may be expressed as $T_{robot/odom}$. The robot's pose in world coordinates can be computed as: $T_{robot/world} = T_{odom/world}T_{robot/odom}$. As noted, the odometry-based state estimates accumulate drift errors as the robot moves. The ROS approach for accounting for this drift is to express the error using variable transformation $T_{odom/world}$. That is, one could say the odometry state estimate is "correct", but the odom reference frame has drifted from its original pose in the world. To reconcile the odom state with the world frame, one must update $T_{odom/world}$.

This odd perspective has some benefits. As noted, the mobile base responsible for publishing odom may be unaware of additional sensors, and thus all it may be able to do is to continue publishing its increasingly bad state estimates. However, if the transform $T_{odom/world}$ can be found that reconciles the odometry values with world-frame states, the odometry signal is still useful. One benefit of the odometry signal is that it is quite smooth and it is updated frequently. These are important virtues for use in feedback for steering. In contrast, additional sensors, such as GPS, may be noisy and update at a much lower rate, making them unsuitable for steering feedback. World-frame sensors (e.g. GPS) do have the benefit of having zero drift. Such signals thus can be used to help update $T_{odom/world}$, reconciling odom publications with world-frame coordinates. Although odom state estimates drift to arbitrarily large errors, such drift is typically slow. Therefore, absolute (world-frame) sensors do not have to have fast updates nor must they be low noise in order to provide ongoing corrections to $T_{odom/world}$. Combining odometry and GPS is described next.

7.2.3 Combining Odometry and LIDAR

One of types of direct environmental sensing, particularly applicable to indoor environments, is LIDAR. Use of LIDAR for localization is most convenient when one has a pre-determined map of the environment. Fortunately, creating maps with LIDAR is relatively simple and convenient using existing ROS packages, as described in Section 8.1. Here, we assume that a map is available, and we wish to establish the robot's pose within the map with the help of LIDAR signals.

An useful package for LIDAR-based localization is Adaptive Monte-Carlo Localization (AMCL). From the package wiki at `http://wiki.ros.org/amcl`:

> amcl is a probabilistic localization system for a robot moving in 2D. It implements the adaptive (or KLD-sampling) Monte Carlo localization approach (as described by Dieter Fox), which uses a particle filter to track the pose of a robot against a known map.

To illustrate use of this package, start up gazebo with the mobot model, add a model environment (the OSRF starting pen, in this example), along with supporting nodes to publish desired states and to perform open-loop control based on desired states. These steps are combined the launch file `mobot_startup_open_loop.launch`. The system can be started with:

```
roslaunch gazebo_ros empty_world.launch
```

```
roslaunch mobot_urdf mobot_startup_open_loop.launch
```

(The above steps could also be combined within a single launch file, as well as the steps to follow).

A map is loaded into a "map server" using the ROS `map_server` package, described at `http://wiki.ros.org/map_server`. An example (partial) map of the starting-pen environment has been created and saved in the directory `maps/starting_pen`. The `maps` directory is created as a ROS package, by vitue of it containing a "package.xml" file and a "CMakeLists.txt" file. However, this directory does not contain any actual code. By creating it as a ROS package, however, one can conveniently navigate to this directory with `roscd`, or refer to this directory in launch files with the expression `$(find maps)`. The starting pen map can be loaded by running:

```
roscd maps
```

followed by:

```
rosrun map_server map_server starting_pen/starting_pen_map.yaml
```

This node publishes the specified map on topic "map" with message type `nav_msgs/OccupancyGrid`. The example map is comprised of 4000 X 4000 cells, with each cell 5cm x 5cm. Each cell contains a gray-scale value indicating if that cell is occupied (black), empty (white) or uncertain (implied by shade of gray between white and black). This would be a fairly large message to send out repeatedly, when it is, in fact, updated only rarely. The map server thus uses a special publication option, "latched", that avoids republications except when the message (map) value is changed.

With the LIDAR signal available (on topic "scan" for our mobot model) and with a map loaded (the starting-pen map, appropriate for the mobot placed within the starting pen), the AMCL localization algorithm can be run. The amcl node is started with:

```
rosrun amcl amcl
```

The amcl node expects to find a map on topic "map" and expects to find LIDAR data on topic "scan." If the LIDAR data is being published to a different topic, this can be accommodated with topic remapping. For example, if the LIDAR data is published to topic `/laser/scan`, amcl should be started with topic remapping as follows:

```
rosrun amcl amcl scan:=/laser/scan
```

With amcl running, this node attempts to discover the pose of the robot with respect to the map. As a particle-filter based method, there is actually a large collection of candidate poses with varying probabilities. The distribution of particles under consideration can be visualized by displaying an item "PoseArray" in rviz, and having it subscribe to the topic `particlecloud` (which is published by amcl). For our example, this display on start-up of amcl appears as in Fig 7.11.

The robot is commanded to move in a 3m x 3m square (open-loop) using a client of the desired-state publisher by running:

```
rosrun mobot_pub_des_state pub_des_state_path_client_3x3
```

After the robot has moved a short distance, the multiple clues provided by LIDAR from

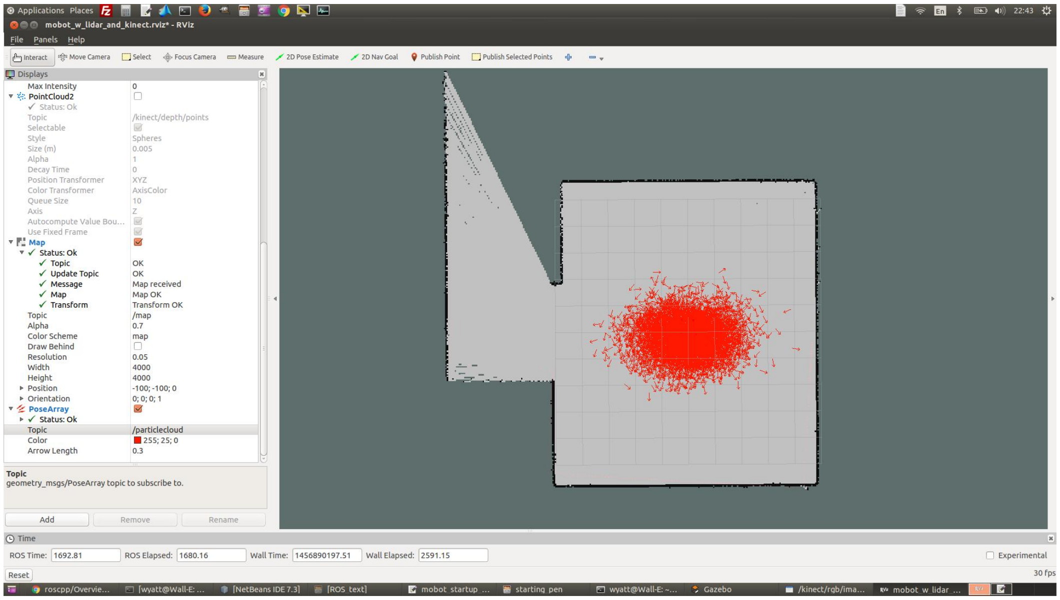


FIGURE 7.10: Distribution of candidate poses for the mobot within the map upon start-up. The candidate poses have a large initial large variance.

changing viewpoints reduces the pose uncertainty. This is shown in Fig **??**, where the candidate poses are more tightly bundled.

The amcl node can be confused by poor initial conditions. The initial pose estimate can be assisted interactively with the rviz tool “2D pose estimate”, which enables the user to click and drag an vector, implying the initial position and heading amcl should assume.

With AMCL running, pose estimates (with respect to the map frame) to the topic `amcl_pose` with message type `geometry_msgs/PoseWithCovarianceStamped`. These updates are relatively infrequent (on the order of seconds), and thus they are unsuitable for steering feedback. Further, these are only poses; twist information is not included.

At the same time, a pose estimate (along with twist) is being published to the “odom” topic that is smooth and much higher bandwidth. The relation between the odom frame and the map frame can be expressed as a transform. Transforms between odom and map are recomputed and published (to the “tf” topic) by amcl (with updates of map pose estimates from amcl). The resulting transforms can be observed using a tool within the “tf” package. Running:

```
rosrun tf tf_echo odom map
```

requests that the transform from “odom” frame to the “map” frame be printed out. An example display from this command is:

```
At time 516.100
- Translation: [0.002, 0.024, 0.002]
- Rotation: in Quaternion [0.002, 0.001, -0.003, 1.000]
            in RPY (radian) [0.004, 0.003, -0.006]
            in RPY (degree) [0.214, 0.145, -0.332]
At time 517.058
- Translation: [0.002, 0.024, 0.002]
```

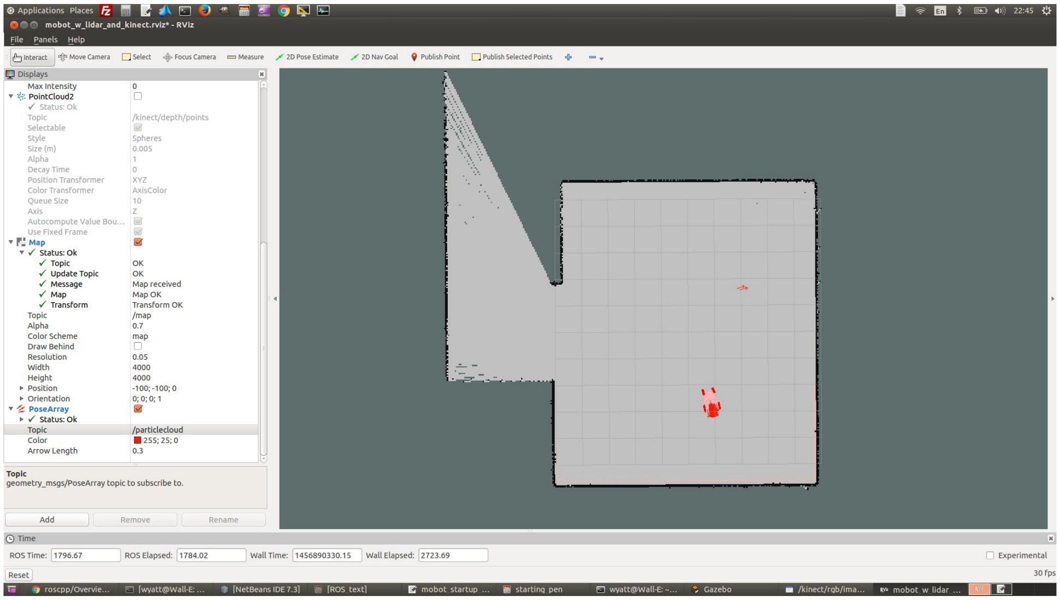


FIGURE 7.11: Distribution of candidate poses for the mobot within the map upon start-up. The candidate poses have a large initial large variance.

```
- Rotation: in Quaternion [0.002, 0.001, -0.003, 1.000]
            in RPY (radian) [0.004, 0.003, -0.006]
            in RPY (degree) [0.214, 0.145, -0.332]
At time 518.059
- Translation: [0.002, 0.024, 0.002]
- Rotation: in Quaternion [0.002, 0.001, -0.003, 1.000]
            in RPY (radian) [0.004, 0.003, -0.006]
            in RPY (degree) [0.214, 0.145, -0.332]
```

This transform can be obtained programmatically within a node and used to correct for odometry drift.

7.2.4 Combining Odometry and GPS

emulate noisy GPS combine (Kalman) for improved estimate; express as a transform; publish xform world to odom (world for GPS)

7.2.5 Transform Listener

7.3 DIFFERENTIAL-DRIVE STEERING ALGORITHMS

Vehicle steering algorithms provide feedback that attempts to make a mobile robot follow a desired path. Many variations exist, depending on how the desired trajectories are specified, what signals are available for feedback and the kinematics of the vehicle's steering mechanism.

Some vehicles have omnidirectional steering, such as the PR2 robot [**?**]. In this case, one can command three degrees of freedom of twist (velocity): a forward speed, a sideways speed and a spin rate. Steering algorithms for omnidirectional motion capability are relatively straightforward.

A more difficult and more common variation involves a steering mechanism as used in road vehicles, called "Ackermann" steering ([**?**]). In this case, one has control over forward speed and over a steering angle (heading of the front wheels, e.g. as imposed by a steering wheel). The angular velocity (yaw rate) of the vehicle depends on both the steering angle and the vehicle's forward speed. The vehicle cannot spin independent of forward motion. Further, the vehicle is kinematically constrained such that it cannot (or should not) slip sideways. A simple approach to steering for such vehicles is the "wagon-handle" algorithm (see, e.g., **??**).

A third case, which will be the focus of this Chapter, is differential-drive kinematics. Examples include skid-steered vehicles, such as iRobot's tracked PackBot[1] , and Clearpath's 4-wheeled Husky ground vehicle[2]. Alternatively (and more gracefully), some differential-drive robots have two drive wheels and passive casters. This design type includes powered wheelchairs, the Roomba [**?**] or Turtlebot[3], and the Pioneer 3, [4]. The simple "mobot" model introduced in Section 3.7 is an example of this class of mobile robots. For such designs, one can control two degrees of freedom independently: speed and yaw rate (spin). We have seen control of this type of robot in simulation, both for the Simple Two-Dimensional Robot (STDR) and the "mobot" model, using speed and spin commands via the `cmd_vel` topic. We begin with a kinematic analysis of differential-drive vehicles.

7.3.1 Robot motion model

Consider a simple vehicle that travels in the x-y plane and which can be commanded with a speed, v, and an angular-rotation rate, ω. The robot has a 3-D pose that can be specified as its x and y coordinates plus its heading, ψ. The heading will be defined as measured CCW from the world frame (or chosen reference frame) x axis. The motion of the robot is described by the following differential equations:

$$\frac{d}{dt}\begin{bmatrix} x \\ y \\ \psi \end{bmatrix} = \begin{bmatrix} v\cos(\psi) \\ v\sin(\psi) \\ \omega \end{bmatrix} \tag{7.10}$$

It will be useful to define path-following error coordinates d_{err} , a lateral offset error, and ψ_{err}, a heading error, relative to a desired path. An example desired path segment is a directed line segment starting from (x_0, y_0) , and with tangent $\mathbf{t} = [\cos(\psi_{des}), \sin(\psi_{des})]^T$ and with edge normal $\mathbf{n} = [-\sin(\psi_{des}), \cos(\psi_{des})]^T$, where ψ_{des} is the angle of the segment relative to the x-axis of the reference frame (measured counter-clockwise). The edge normal is a positive 90-deg rotation of $\mathbf{t}$, where positive rotation is defined as "up" (rotation about

[1] `http://www.irobot.com/For-Defense-and-Security/Robots/510-PackBot.aspx#Military`
[2] `http://www.clearpathrobotics.com/husky-unmanned-ground-vehicle-robot/`
[3] `http://www.turtlebot.com/`
[4] `http://www.mobilerobots.com/ResearchRobots/PioneerP3DX.aspx`

z, where the z-axis is defined normal to the x-y plane, consistent with forming a right-hand coordinate system x-y-z).

With respect to this directed line segment, we can compute a lateral offset error of the robot at position (x, y) as:

$$d_{err} = \left(\begin{bmatrix} x \\ y \end{bmatrix} - \begin{bmatrix} x_0 \\ y_0 \end{bmatrix} \right) \cdot \mathbf{n} \tag{7.11}$$

The heading error, ψ_{err}, is computed as the actual heading vs the desired heading: $\psi_{err} = \psi - \psi_{des}$. With rotational variables, it is important to consider periodicity. For example, consider a desired heading of $\psi_{des} = 0$ (i.e., pointing parallel to the x axis) and an actual heading of $\psi = 0.1$. This corresponds to a heading error of 0.1 rad, which is close to the desired heading. However, for a heading of $\psi = 6.0$, the formula $\psi_{err} = \psi - \psi_{des}$ would seem to imply a heading error of 6.0 rad–a large heading error. Instead, the periodic solutions of this result should be checked to find the smallest error interpretation. The value of $\psi_{err} - 2\pi$ corresponds to a negative heading error of approximately -0.28 rad. That is, the heading is more easily corrected by rotating a postive 0.28 rad than by rotating negative 6 rad. This periodic condition must be checked within iterations of a control algorithm, else it can lead to gross instability. We will choose to define the heading error as the smallest-magnitude option among periodic alternatives.

We will define a 2-D path-following error vector in terms of our two error components as:

$$\mathbf{e} = \begin{bmatrix} d_{err} \\ \psi_{err} \end{bmatrix} \tag{7.12}$$

The components may be interpreted as a lateral offset, which is positive to the "left" of the path, and a heading error, which is measured positive as a CCW rotation of heading away from the path tangent.

In the following, we will assume that the robot is moving forward at speed v, but that we can command a superimposed rotation rate, ω, with the objective of driving both components of the error vector to zero. If the error vector is zero, then the robot will have zero offset (i.e. will be positioned on top of the path segment) and zero heading error (i.e. will be pointing consistent with the desired heading). For a differential-drive robot, an objective of a steering algorithm is to compute an appropriate spin command, ω, to drive the error components to zero, and to do so with desirable dynamics (e.g., rapidly and stably).

7.3.2 Linear steering of a linear robot

We first consider a simplified system: a linear approximation of the robot dynamics controlled by a linear controller. For simplicity, assume that the desired path is the positive x axis, for which the error vector is simply $\mathbf{e} = [y, \psi]^T$ (assuming ψ is expressed as the smallest-magnitude option among periodic alternatives).

Assuming a linear controller, we may command $\omega = \mathbf{K}\mathbf{e}$, where $\mathbf{K} = [K_d, K_\psi]$. The two components of $\mathbf{K}$ are control gains, which should be chosen to result in desirable dynamics of the robot approaching the desired path.

Consider simplified (linearized) vehicle dynamics of:

$$\frac{d}{dt} \begin{bmatrix} x \\ y \\ \psi \end{bmatrix} = \begin{bmatrix} v \\ v\psi \\ \omega \end{bmatrix} \tag{7.13}$$

corresponding to a small-angle approximation for small values of ψ.

Equivalently, the error dynamics can be expressed as:

$$\frac{d}{dt}\mathbf{e} = \begin{bmatrix} 0 & v \\ 0 & 0 \end{bmatrix} \mathbf{e} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} \omega \tag{7.14}$$

Substituting in our control policy for ω yields:

$$\frac{d}{dt}\mathbf{e} = \begin{bmatrix} 0 & v \\ 0 & 0 \end{bmatrix} \mathbf{e} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} \mathbf{K}\mathbf{e} = \begin{bmatrix} 0 & v \\ K_d & K_\psi \end{bmatrix} \mathbf{e} \tag{7.15}$$

In the LaPlace domain, this corresponds to:

$$\begin{bmatrix} 0 \\ 0 \end{bmatrix} = \begin{bmatrix} s & -v \\ -K_d & s - K_\psi \end{bmatrix} \mathbf{e} \tag{7.16}$$

which has a characteristic equation of:

$$0 = s^2 - K_\psi s - vK_d \tag{7.17}$$

Any combination of negative values for the control gains theoretically results in a stably-controlled system (for this linear approximation). We can choose gains intelligently by choosing values interpreted in terms of the generic second-order system response: $0 = s^2 + 2\zeta\omega_n s + \omega_n^2$. E.g., if we choose $\omega_n = 6$ (roughly 2π, or about 1Hz) and $\zeta = 1$ (i.e., critical damping), we would expect convergence to the desired path with a time constant of approximately 1 second with zero overshoot.

Figures 7.12, 7.13, and 7.14 show a simulation of the linearized system with the chosen controller. The initial offset error and the heading error both converge to zero within about 1 second, as expected, and the offset error does not overshoot, consistent with a critically-damped system. The control effort, i.e. the ω command, is shown in Fig 7.14. The spin command starts with a strongly negative ω, which changes sign before converging on zero as the robot converges on zero offset error and zero heading error.

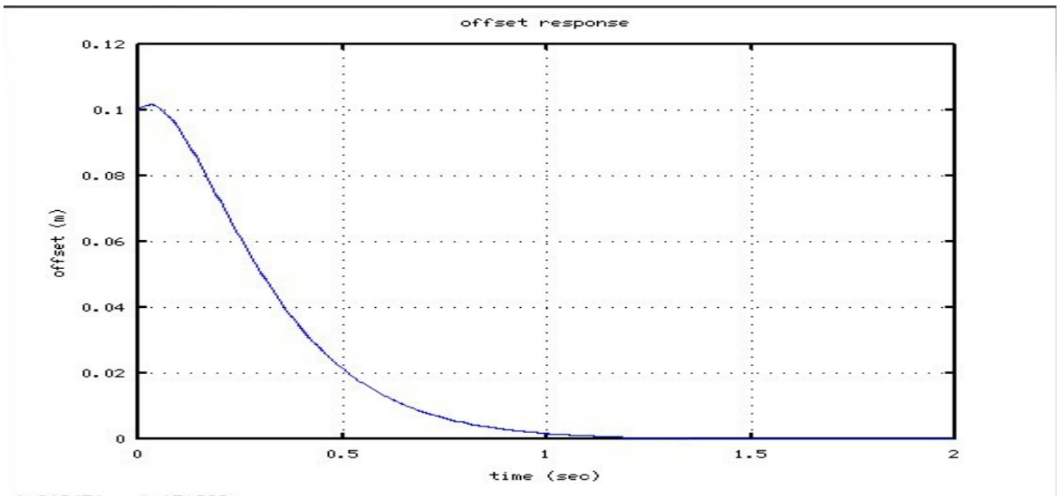


FIGURE 7.12: Offset response, linear model, linear controller, 1Hz controller, critically damped

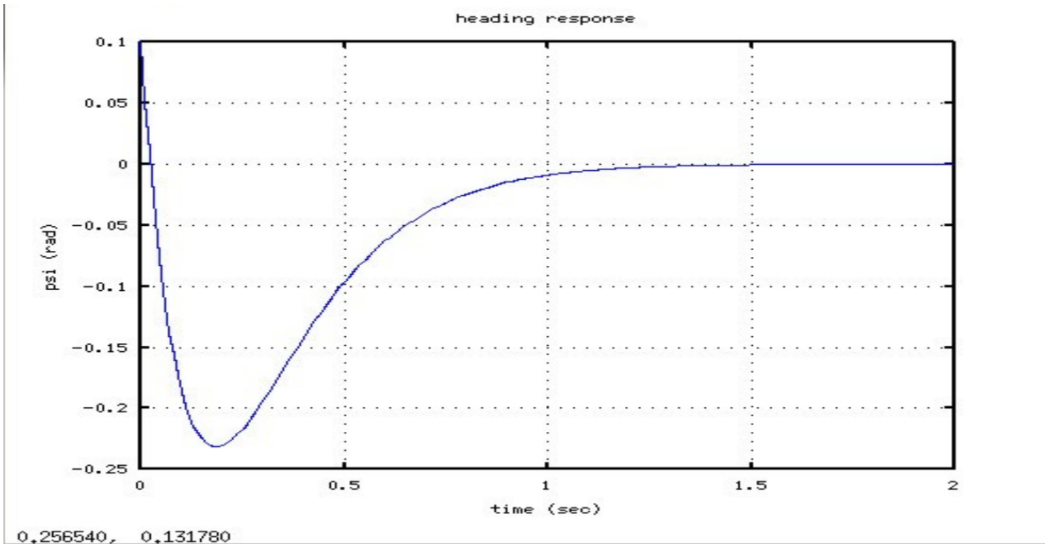


FIGURE 7.13: Heading response of linear system

7.3.3 Linear steering of a non-linear robot

In the preceding section, we considered a linear controller acting on a hypothetical linear system (a small-angle approximation of our robot dynamics). We should evaluate the consequences of attempting use of a linear controller on a more realistic model of the robot. With the same gains as in Section 7.3.2, but with the dynamic model of Section 7.3.1 (not linearized), we consider a directed line-segment desired path, specified as follows. The start of the line segment is at $(x, y) = (1, 0)$, and the slope of the desired path is 45-deg. Path heading error and offset error are computed as described in Section 7.3.1. The angular-rotation command is computed based on the linear control algorithm and control gains of Section 7.3.2.

Figures 7.15 and 7.16 show the response to relatively small initial errors, displayed as lateral-offset error, Fig 7.15, and path vs desired path in the x-y plane 7.16. The response is good, similar to that of the linear analysis.

Figure 7.17 shows the response of the linear controller on the nonlinear robot for a variety of initial conditions for which convergence is well behaved. In this view, the delta-heading is plotted as a function of displacement, which is a “phase space” plot. It can be observed that as the robot approaches convergence to the desired path segment, the delta-heading (robot heading minus specified path heading) is approximately linearly proportional to the lateral-offset error. This observation will be used to help design a nonlinear controller that that shares characteristics with the linear controller.

Unlike linear systems, the stability of a nonlinear system can depend on initial conditions. Tuning the controller for a desirable response may give the impression that it is well behaved, but different initial conditions can result in a wildly unstable response. Figures 7.18 and 7.19 show an example with a large initial offset error. The robot spins in circles, potentially dangerously, and it fails to converge on the desired path.

In short, a linear controller cannot be trusted on the actual robot. A nonlinear controller is considered next, which blends strategies for large offset errors with the behavior of a linear controller for small path-following errors.

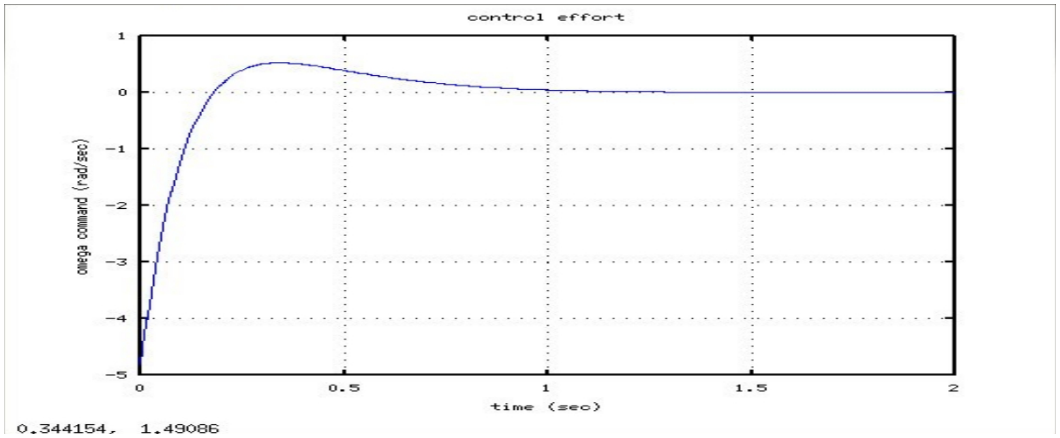


FIGURE 7.14: Control effort history. Note that spin command may exceed actual physical limitations

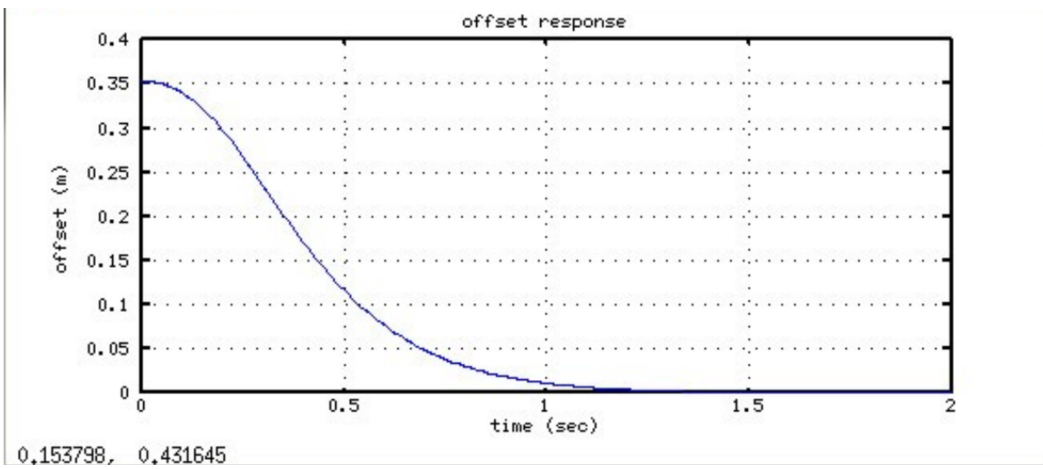


FIGURE 7.15: Offset response, nonlinear model, 1Hz linear controller. Response to small initial error is good, similar to linear model response

7.3.4 Nonlinear steering of a non-linear robot

In the previous sections, it was shown that a linear controller works well as a steering algorithm, provided the path-following errors are sufficiently small. The linear control algorithm computes a spin- rate command as a weighted sum of heading and displacement errors. However, when the robot is far from the goal (or has a large heading error), this linear mapping is inappropriate.

If the lateral displacement is large, the robot should first attempt to head directly towards the desired path. To do so, the best heading is to orient the robot towards the path, i.e. with robot heading orthogonal to path heading. In this case, a heading error of $\pm\pi/2$ is desirable and should not be penalized. This may be considered an “approach” or a “reaching” phase.

Once the lateral displacement between the path segment and the robot is sufficiently small, the reaching phase should transition to a path-following phase, for which a linear

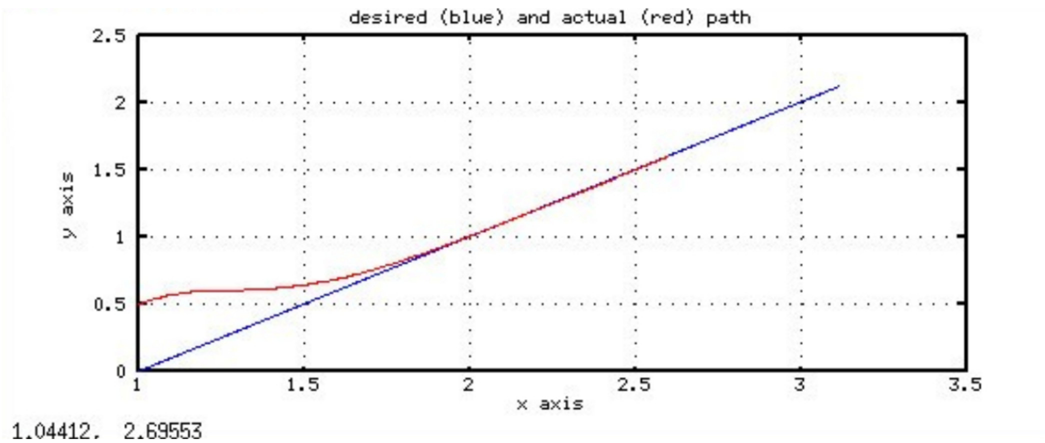


FIGURE 7.16: Path following of nonlinear robot with linear controller and small initial error. Convergence to precise path following is well behaved.

controller is well suited. In the path-following phase, the heading should tend to align with the path tangent (i.e. the heading error should tend towards zero) as the lateral offset decreases.

The above observations can be integrated in a formula that combines the approach behavior and the path-following behavior as a function of lateral offset. To construct this controller, we define four different heading variables: ψ_{path}, the ideal heading along a specified path; ψ_{state}, the current heading of the robot; $\psi_{strategy}$, a computed strategy of desirable heading to achieve convergence to the specified path; and ψ_{cmd}, a heading command to be sent to a low-level heading controller.

The nonlinear steering controller described here depends on a low-level heading controller. The purpose of the heading controller is to cause the robot to change its orientation, ψ_{state}, to point in a commanded direction, ψ_{cmd}. A simple heading controller may be of the form:

$$\omega = K_{\psi}(\psi_{cmd} - \psi_{state}) \tag{7.18}$$

although more sophisticated controllers may be used for better performance.

In implementation, the value of $(\psi_{cmd} - \psi_{state})$ should be checked for periodicity, choosing the direction of rotation that is fastest (i.e. using the smallest rotation angle to reach the commanded heading). The gain value, K_{ψ}, should be tuned to achieve good response from the system.

It is also desirable to have the controller explicitly respect saturation limits. If the speed limit imposed on rotation rate (whether a hard or soft limit) is ω_{max}, this can be respected by saturating the feedback algorithm to put limits on the spin command, ω_{cmd}:

$$\omega_{cmd} = \omega_{max} f_{sat}(K_{\psi}(\psi_{cmd} - \psi_{state})) \tag{7.19}$$

The saturation function, f_{sat}, is defined as:

$$f_{sat} = \begin{cases} -1 & \text{if } x < -1 \\ 1 & \text{if } x > 1 \\ x & \text{otherwise} \end{cases} \tag{7.20}$$

Assuming the heading controller works well, it remains to compute a strategy for heading commands that will lead the robot to converge on the specified path. This mapping should

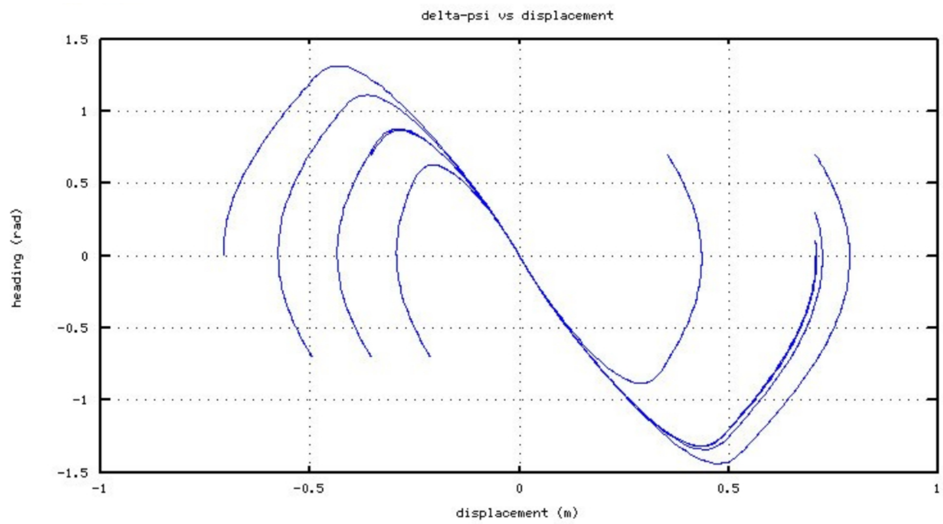


FIGURE 7.17: Phase space of linear control of nonlinear robot. Note linear relation between displacement and delta-heading near the convergence region.

---

result in values of $\psi_{strategy}$ that are continuous and smooth, and which achieve the desired result of good convergence to the specified path regardless of initial conditions. It should also blend approach strategies for large d_{err} (point towards the path, orthogonal to the path) and small d_{err} (behave like a linear controller for path following).

Initially, consider a desired path corresponding to the x axis (in the chosen reference frame). The path heading for this case is ψ_{path}=0, the lateral offset error is $d_{err} = y$, and the heading error is $\psi_{err} = \psi_{state}$. A nonlinear approach strategy can be constructed that specifies $\psi_{strategy}$ as a function of lateral offset, d_{err}, as follows:

$$\psi_{strategy}(d_{err}) = -(\pi/2) f_{sat}(d_{err}/d_{thresh}) \tag{7.21}$$

In this formula, the value d_{thresh} (with units of meters) is a parameter to be tuned. When the lateral offset error approaches d_{thresh}, the heading strategy approaches $\pm\pi/2$, causing the robot to orient perpendicular to (and pointing towards) the current path segment. For values of offset error, d_{err}, that are small compared to d_{thresh}, the heading strategy is proportional to the offset error, which conforms to the behavior of the convergence region of a linear steering algorithm, as seen in Fig 7.17.

To generalize the approach strategy to path segments that are oriented arbitrarily, the specified path heading, ψ_{path}, should be added to the computed-strategy heading, yielding:

$$\psi_{cmd} = \psi_{path} + \psi_{strategy} \tag{7.22}$$

The resulting ψ_{cmd} becomes the input to the lower-level heading controller.

7.3.5 Simulating the nonlinear steering algorithm

The package `mobot_nl_steering` contains an implementation of the nonlinear steering algorithm as a class, SteeringController. The code is lengthy, and thus is not displayed here in full. Some key excerpts are interpreted here.

A member function that accounts for periodicity to find an angular error is:

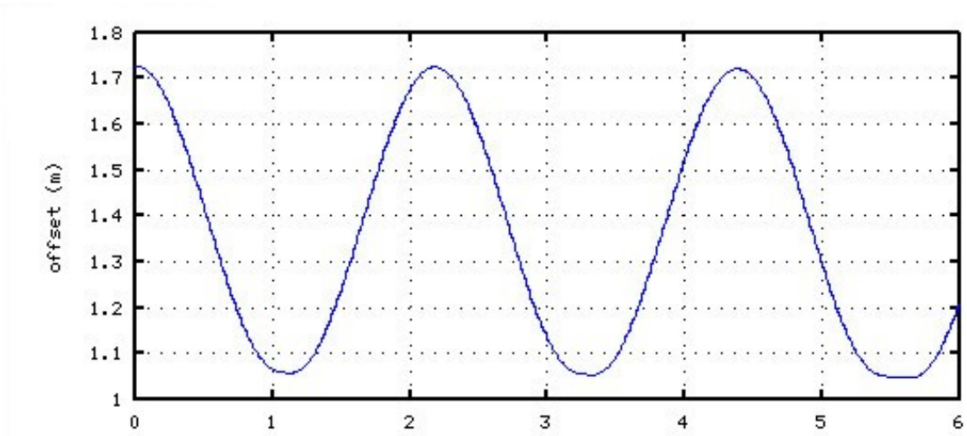


FIGURE 7.18: Linear controller on nonlinear robot with larger initial displacement. Displacement error vs time oscillates.

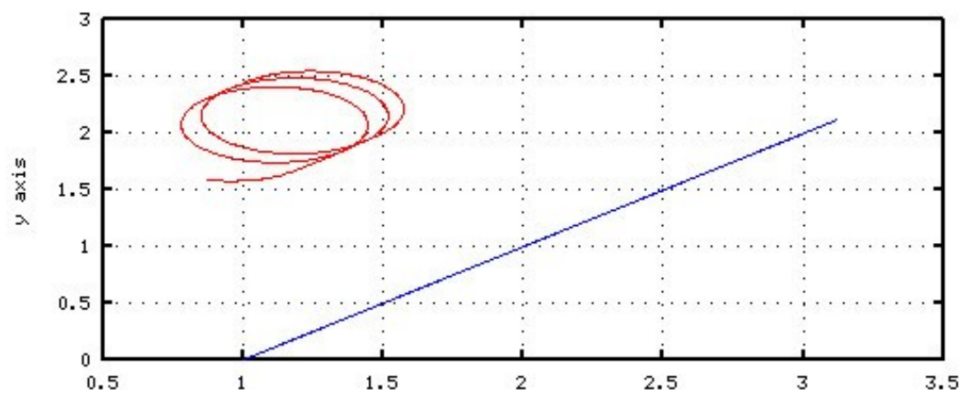


FIGURE 7.19: Path from linear controller on nonlinear robot with large initial error. Robot spins in circles, failing to converge on desired path.

```
double SteeringController::min_dang(double dang) {
    while (dang > M_PI) dang -= 2.0 * M_PI;
    while (dang < -M_PI) dang += 2.0 * M_PI;
    return dang;
}
```

This function is useful for finding the direction and magnitude of the minimum angular distance motion from some initial angle to some desired angle.

Implementation of the saturation function is:

```
double SteeringController::sat(double x) {
    if (x>1.0) {
        return 1.0;
    }
    if (x< -1.0) {
        return -1.0;
    }
    return x;
```

```
}
```

The main steering function is `SteeringController::mobot_nl_steering()`. The following code segment:

```
    double tx = cos(des_state_psi_); // [tx,ty] is tangent of desired path
    double ty = sin(des_state_psi_);
    double nx = -ty; //components [nx, ny] of normal to path, points to left of ↩
        desired heading
    double ny = tx;

    double dx = state_x_ - des_state_x_;  //x-error relative to desired path
    double dy = state_y_ - des_state_y_;  //y-error

    lateral_err_ = dx*nx+dy*ny; //lateral error is error vector dotted with path ↩
        normal
                                     // lateral offset error is positive if robot is ↩
                                         to the left of the path
    double trip_dist_err = dx*tx+dy*ty; // progress error: if positive, then we are ↩
        ahead of schedule
    //heading error: if positive, should rotate -omega to align with desired heading
    double heading_err = min_dang(state_psi_ - des_state_psi_);
    double strategy_psi = psi_strategy(lateral_err_); //heading command, based on NL ↩
        algorithm
    controller_omega = omega_cmd_fnc(strategy_psi, state_psi_, des_state_psi_); //spin↩
         command
```

computes the path tangent, **t**, given a desired path heading `des_state_psi_`, as well as the path normal, **n**, the lateral offset error, `lateral_err_`, and the heading error, `heading_err`, taking periodicity into consideration. The value of `strategy_psi` is computed as a function of the lateral error. This function is given by:

```
double SteeringController::psi_strategy(double offset_err) {
  double psi_strategy = -(M_PI/2)*sat(offset_err/K_LAT_ERR_THRESH);
  return psi_strategy;
}
```

which implements our derived nonlinear expression for heading strategy. This is used in the heading controller:

```
double SteeringController::omega_cmd_fnc(double psi_strategy, double psi_state, double↩
     psi_path) {
  psi_cmd_ = psi_strategy+psi_path;
  double omega_cmd = K_PSI*(psi_cmd_ - psi_state);
  omega_cmd = MAX_OMEGA*sat(omega_cmd/MAX_OMEGA); //saturate the command at specified ↩
      limit
  return omega_cmd;
}
```

The values `K_PSI` and `K_LAT_ERR_THRESH` are parameters to be tuned for desirable response (e.g., fast but with low overshoot).

To invoke the controller on our simulated mobile robot, we need an estimate of the robot's state–specifically, published on topic `gazebo_mobot_pose` as a message of type `geometry_msgs::Pose`. For the purpose of evaluating our controller, this state estimation is taken from the gazebo publication on topic `gazebo/model_states`, parsed for the state of the model "mobot", and republished on topic `gazebo_mobot_pose`. This is performed by the node `mobot_gazebo_state` in the package `mobot_gazebo_state`. Simulation experiments can be run by starting the simulator:

```
roslaunch gazebo_ros empty_world.launch
```

loading the robot model:

```
roslaunch mobot_urdf mobot.launch
```

starting the mobot state publisher:

```
rosrun mobot_gazebo_state mobot_gazebo_state
```

and running the controller:

```
rosrun mobot_nl_steering mobot_nl_steering
```

Desired path states can be published to the topic `/desState` with message type `nav_msgs::Odometry`. However, by default, the desired path is the world x axis (pointing in the positive direction). The robot can be repositioned and reoriented in gazebo, and the nonlinear steering controller will attempt to have the robot converge on the current path segment (e.g., the x axis).

For chosen parameters of:

```
const double K_PSI= 5.0; // control gains for steering
const double K_LAT_ERR_THRESH = 3.0;
// dynamic limitations:
const double MAX_SPEED = 1.0; // m/sec; tune this
const double MAX_OMEGA = 1.0; // rad/sec; tune this
```

an example response is shown in Fig 7.20. In Fig 7.20, the robot starts out with a lateral displacement error of approximately 4m (positive, i.e. shifted to the left of the desired path). Since this is a large displacement error, the heading strategy is computed to be $-\pi/2$. Since the robot's initial heading is zero, the difference between actual and desired heading is large, resulting in saturating the ω command at -1.0 rad/sec. This command persists while the robot reorients and moves forward, closer to the path. After about 2 sec, the heading strategy starts to reorient towards 0 (the path heading). The ω command becomes positive to achieve more positive headings (approaching 0 heading). After about 4 sec, the remainder of the trajectory looks like a linear controller, with both heading error and offset error smoothly converging to zero.

- show results w/ mobot: odom only
- show results w/ drifty odom and noisy GPS
- segue to path planning

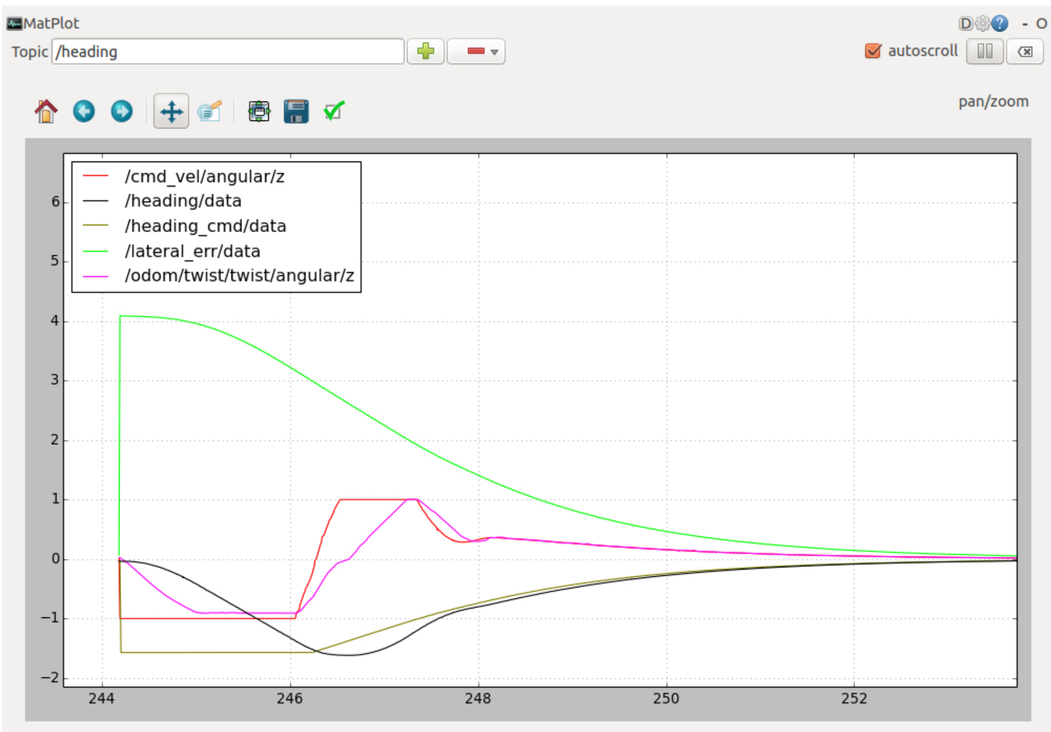


FIGURE 7.20: Phase space of linear control of nonlinear robot. Note linear relation between displacement and delta-heading near the convergence region.

CHAPTER 8

Mobile-Robot Path Planning

CONTENTS

Chapter intro goes here... map makink, ROS path-planning packages, ...

8.1 MAP MAKING

The "gmapping" package can be used to create maps using a mobile robot and a LIDAR sensor. Per the gmapping wikie, `http://wiki.ros.org/gmapping`:

> The gmapping package provides laser-based SLAM (Simultaneous Localization and Mapping), as a ROS node called `slam_gmapping`. Using `slam_gmapping`, you can create a 2-D occupancy grid map (like a building floorplan) from laser and pose data collected by a mobile robot.

To use this package, the robot's LIDAR must be mounted horizontally with a fixed mount, and the robot must publish odometry. This package post-processes LIDAR and "tf" data and attempts to build a map that explains the LIDAR data relative to the robot's movements.

The first step in map making is to acquire (bag) data. As an example of this, start up gazebo:

```
roslaunch gazebo_ros empty_world.launch
```

The launch file `mobot_startup_open_loop.launch` in the package `mobot_urdf` starts up the mobot model, brings in the `starting_pen` model, starts up the desired-state publisher, and starts up an open-loop controller. The robot model used in this launch contains LIDAR, a Kinect sensor and a color camera. Rviz is also launched. Perform these operations with:

```
roslaunch mobot_urdf mobot_startup_open_loop.launch
```

With the robot running, rviz should show that the LIDAR is active and that it can sense the walls of the starting pen. At this point, the robot should wander (slowly) through some path, exploring its environment while bagging data on the topics "tf" and "scan" (where "scan" is the LIDAR topic for the mobot model).

To record the data, navigate to a desired directory. The current example uses the "maps" directory within the `learning_ros` respository. From the chosen directory, start recording data with:

```
rosbag record -O mapData /scan /tf
```

While data is being recorded, have the robot move. An example pre-scripted motion for a 3m x 3m square path may be invoked with:

```
rosrun mobot_pub_des_state pub_des_state_path_client_3x3
```

If desired, this can be re-run to have the robot perform a second (or third) repeat of the path. Due to open-loop control, the robot will drift from the desired path, so repeats of this path request will yield novel viewpoints, which is useful for map-making.

When using a real robot for map-making, one can start the bagging process, then use a joystick to manually drive the robot through the environment of interest. When doing so, one should take care to move the robot slowly–particularly in turning motions. Such data will be easier to post-process to reconcile scans from different poses.

Once enough data has been collected, kill the rosbag process (with control-C). Gazebo can be killed as well. The bagged data will be contained in the file "mapData" within the chosen directory (e.g. "maps").

Given a rosbag recording that contains LIDAR-scan data and transform (tf) data, the rosbag can be post-processed to compute a corresponding map. To do so, start a roscore, then, from the directory in which the rosbag data resides, run:

```
rosrun gmapping slam_gmapping scan:=/scan
```

The remapping option `scan:=/scan` is not necessary in the present example, since the LIDAR scan topic for the mobot model is already called "scan." However, if the robot's scan topic were, e.g., `/robot0/scan`, then one should use topic remapping with the option: `scan:=/robot0/scan`. At this point, the gmapping map-builder is subscribed to the tf topic and to the LIDAR topic. To publish the acquired data to the respective topics, playback the data with:

```
rosbag play mapData.bag
```

The above command assumes that the recorded data is called "mapData.bag" (an option chosen during rosbag recording), and it is assumed that this command is being run from the directory in which "mapData.bag" resides. With both `rosbag play` and `gmapping slam_mapping` running, the map-making node will receive publications of LIDAR and tf data and will attempt to build a map from this data.

The map-making process can be watched dynamically, if desired. To do so, bring up rviz:

```
rosrun rviz rviz
```

Add a "map" display item. In the display panel, expand this new item and choose the topic "/map". While the recording is being played back, one can watch a display of the map that is being built. An example is shown in Fig 8.1. Figure 8.1 shows a 2-D occupancy map, where shades of gray imply the degree of certainty of a cell being occupied. Black cells correspond to high probability of occupancy, and white cells imply a high probability of being vacant. Gray cells imply no information is available. The display in Fig 8.1 is a credible fit to the starting pen. The exit hallway is incompletely mapped, since the robot did not travel to poses from which a line-of-sight LIDAR acquisition could detect the occluded hallway.

When the bag-file playback is complete, the created map needs to be saved to disk. This is done with the command:

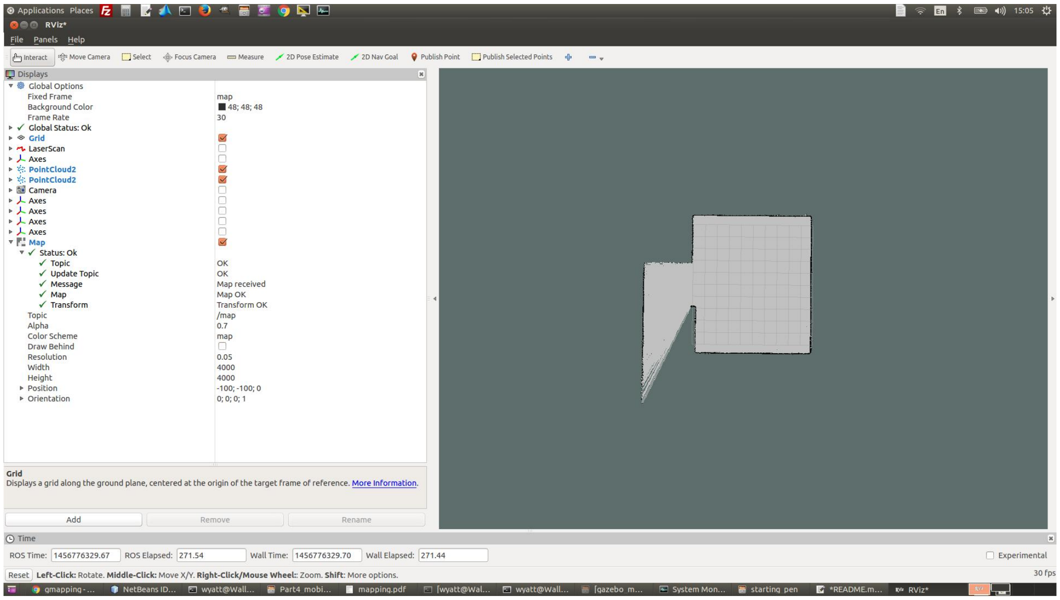


FIGURE 8.1: Rviz view of map constructed from recorded data of mobot moving within the starting pen

```
rosrun map_server map_saver -f newMap
```

where "newMap" is the file name, which should be chosen mnemonically. This will create two files: newMap.yaml and newMap.pgm. For the example map created, the file newMap.yaml contains:

```
image: newMap.pgm
resolution: 0.050000
origin: [-100.000000, -100.000000, 0.000000]
negate: 0
occupied_thresh: 0.65
free_thresh: 0.196
```

These parameters declare that the map data can be found in the file named "newMap.pgm", and that the resolution of cells within this map is 5cm. The threshold values are used by planning algorithms to decide if a cell is to be called "occupied" or "vacant."

The map data itself is contained in "newMap.pgm". This is in an image format, which can be viewed directly just by double-clicking the file within Linux. If desired, the map may be touched up using an image editor program.

V

Robot Arms in ROS

INTRODUCTION
Intro here.

- low-level control
- Fwd Kin
- IK
- trajectories
- kinematic redundancy planning
- collision avoidance planning
- perception-based manipulation

CHAPTER 9

Low-Level Control

CONTENTS

This section will explore variations on low-level joint control in ROS. In Section 3.5, robot-arm control concepts in ROS were introduced using a separate feedback node that interacted with Gazebo via Gazebo topics. This approach, while suitable for illustrative purposes, is not preferred, since sample rates and latencies due to communication via topics limits the achievable performance of such controllers. Instead, as introduced in Section 3.6, it is preferable to interact with Gazebo using plug-ins. Design of Gazebo plug-ins will not be covered here, but the interested designer can learn about plug-ins from on-line tutorials (see `http://gazebosim.org/tutorials?tut=ros_gzplugins`). Existing plug-ins for position controllers and velocity controllers will be discussed.

Using a Gazebo plug-in for joint control requires a fair amount of detailed specification (see `http://gazebosim.org/tutorials/?tut=ros_control`). One specifies joint range-of-motion limits, velocity limits, effort limits and damping. The URDF must specify a transmission and actuator for each controlled joint. The `gazebo_ros_control.so` library is included within a Gazebo tag in the robot model file. Control gains are specified in an associated YAML file. Finally, one specifies in a launch file to spawn individual joint controllers (position or velocity). Having specified the control type to use, one needs to tune feedback parameters, for which ROS offers some helpful tools.

This section will illustrate use of three simple controllers: position control, velocity control and force control (which will require modelling a force sensor). These examples will use a single degree-of-freedom robot with a prismatic joint.

9.1 A ONE-DOF, PRISMATIC-JOINT ROBOT MODEL

The associated package `example_controllers` includes the file `prismatic_1dof_robot_description_w_jnt_ctl.xac` This model file is largely similar to the `minimal_robot_description.urdf` in package `minimal_robot_description`, described in Section 3.2, except that its single degree of freedom is a prismatic joint instead of a revolute joint. Key components of this file include the following lines:

```
<joint name="joint1" type="prismatic">
  <parent link="link1"/>
  <child link="link2"/>
  <origin xyz="0 0 1" rpy="0 0 0"/>
  <axis xyz="0 0 1"/>
  <limit effort="1000.0" lower="-1.0" upper="0.0" velocity="100.0"/>
```

```
    <dynamics damping="10.0"/>
  </joint>
```

which defines “joint1” and specifies upper and lower range of motion, maximum control effort (1000N force), and a maximum velocity (100m/sec).

The following lines, declaring a transmission and actuator, are also required to interface with ROS controllers:

```
  <transmission name="tran1">
    <type>transmission_interface/SimpleTransmission</type>
    <joint name="joint1">
      <hardwareInterface>EffortJointInterface</hardwareInterface>
    </joint>
    <actuator name="motor1">
      <hardwareInterface>EffortJointInterface</hardwareInterface>
      <mechanicalReduction>1</mechanicalReduction>
    </actuator>
  </transmission>
```

The controller plug-in library is included in the URDF with the following lines:

```
  <gazebo>
     <plugin name="gazebo_ros_control" filename="libgazebo_ros_control.so">
       <robotNamespace>/one_DOF_robot</robotNamespace>
     </plugin>
  </gazebo>
```

(The model also includes a simulated force sensor, but discussion of this is deferred for now).

The ROS control plug-in includes options for both position and velocity controllers. The specific controller to be used is established by launching appropriate nodes.

9.2 AN EXAMPLE POSITION CONTROLLER

The launch file `prismatic_1dof_robot_w_jnt_pos_ctl.launch` contains:

```
<launch>
  <!-- Load joint controller configurations from YAML file to parameter server -->
  <rosparam file="$(find example_controllers)/control_config/one_dof_pos_ctl_params.yaml"
     command="load"/>

<!-- Convert xacro model file and put on parameter server -->
<param name="robot_description" command="$(find xacro)/xacro.py
  '$(find example_controllers)/prismatic_1dof_robot_description_w_jnt_ctl.xacro'" />

  <!-- Spawn a robot into Gazebo -->
  <node name="spawn_urdf" pkg="gazebo_ros" type="spawn_model"
     args="-param robot_description -urdf -model one_DOF_robot" />

  <!--start up the controller plug-ins via the controller manager -->
  <node name="controller_spawner" pkg="controller_manager" type="spawner" respawn="false"
    output="screen" ns="/one_DOF_robot" args="joint_state_controller
    joint1_position_controller"/>

</launch>
```

This launch file loads control parameters from file `one_dof_pos_ctl_params.yaml`, it converts the model xacro file into a URDF file and loads it on the parameter server, it spawns the robot model into Gazebo for simulation, and it loads the controller `joint1_position_controller`. The controller parameters must be consistent with the chosen controller type (position controller). Further, the joint name associated with the controller (joint1) must be part of the controller name loaded.

The YAML file with control parameters, `one_dof_pos_ctl_params.yaml`, is listed below:

```
one_DOF_robot:
  # Publish all joint states -----------------------------------
  joint_state_controller:
    type: joint_state_controller/JointStateController
    publish_rate: 50

  # Position Controllers ---------------------------------------
  joint1_position_controller:
    type: effort_controllers/JointPositionController
    joint: joint1
    pid: {p: 400.0, i: 0.0, d: 0.0}
```

The control-parameter file must reference the name of the robot (`one_DOF_robot`), and for each joint (only joint1, in this case), the position-control gains must be listed. In the example, the proportional gain is 400 N/m, and all other gains are suppressed to zero. Damping is provided implicit to the joint definition (which contains the specification: `<dynamics damping="10.0"/>`). In fact, the derivative gain in the default position controller is quite noisy, presumably due to use of backwards-difference velocity estimates. Implicit joint damping in the robot model is better behaved than derivative-position feedback.

Choosing good control gains can be challenging, but ROS offers tools that can help. A detailed tutorial can be found at `http://gazebosim.org/tutorials/?tut=ros_control`. In choosing gains, one must keep in mind that the physics engine runs with a default time step of 1ms, and thus control bandwidths must be designed to accommodate this limitation. For 1kHz update rate, controllers should be limited (roughly) to less than 100Hz bandwidth. In practice, numerical artifacts appear well below this frequency. For the one-DOF example, the mass of link2 was set to 1kg, and the proportional gain was set to 400. The undamped natural frequency of the system is thus $sqrt(K_p/m) = 20$ in rad/sec, or about 3Hz.

The system response to sinusoidal command inputs can be observed using `rqt_plot`, and control gains can be adjusted interactively using `rqt_gui`. Sinusoidal command inputs can also be specified using `rqt_gui`. In the `example_controllers` package, there is an alternative excitation node called `one_dof_sine_commander`, which prompts for frequency and amplitude and publishes sinusoidal commands to the joint1 command topic. The system can be tested by running the following. First, start Gazebo with:

```
roslaunch gazebo_ros empty_world.launch
```

Then spawn the robot and its controller with the associated control gains using:

```
roslaunch example_controllers prismatic_1dof_robot_w_jnt_pos_ctl.launch
```

Start up `rqt_gui` and `rqt_plot`:

```
rqt_plot
rosrun rqt_gui rqt_gui
```

Choose topics to plot in `rqt_plot` to be `one_DOF_robot/joint1_position_controller/command/data`, `one_DOF_robot/joint_states/position[0]`, and `one_DOF_robot/joint_states/effort[0]`.

Using `rqt_gui` is somewhat involved, as there are many options. One can add a Message Publisher and select the topic `one_DOF_robot/joint1_position_controller/command`. Expanding this topic, the sole item is "data". One can edit the "expression" value on this row to define the command to be published. This can be as simple as a constant value, or it can be an expression to be evaluated dynamically (typically, a sinusoidal function).

In a separate panel of `rqt_gui`, one can perform "dynamic reconfigure" to vary control gains interactively. Expanding the joint1 control topic down to the level of "PID" exposes sliders for the control gains. These will be initialized to the values in the control-parameter YAML file. They can be adjusted while the robot is running, so one can immediately observe their effects in `rqt_plot`.

Figure 9.1 shows a snapshot of this process. The Gazebo window shows the robot; the `rqt_gui` window shows the selection of joint-command publisher and PID values, and the `rqt_plot` window shows the transient response of position command, actual position, and control effort due to a step in command from -0.2m to -0.5m.

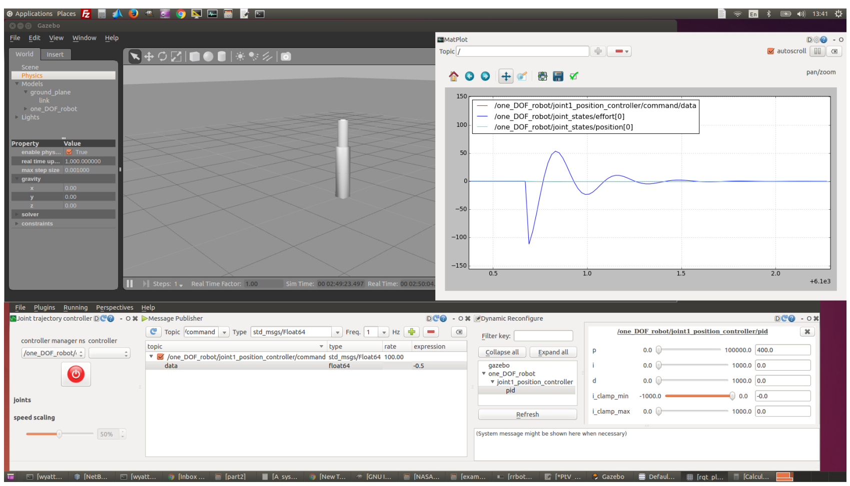


FIGURE 9.1: Servo tuning process for 1-DOF robot

The control effort in Fig 9.1 is large, squeezing the displacement response to a small range. Figure 9.2 shows a zoom of this transient to reveal the step command (in red) and the position response (in cyan). The response shows several overshoots and convergence over roughly half a second. Alternatively, the published stimulus can be a sinusoid, as shown in Fig 9.3. For this case, the published command is a function: `0.1*sin(i/5)-0.5`, which is a sinusoid of amplitude 0.1m, offset by -0.5m (which is the middle of the joint's range of motion). The frequency follows from the publication rate, which is set to 100Hz. The value of "i" increments at this rate, and thus the frequency of this sinusoid is 20 rad/sec. Since our undamped natural frequency is 20 rad/sec, we expect to see poor tracking at this frequency. As expected, the position response lags the command by a phase of 90 deg at this frequency of excitation. The PID sliders can be varied to test alternative gains and their influence on response to step inputs or sinusoids.

A limitation of the position controller is that the derivative gain does not accept velocity feed-forward commands. Consequently, dynamic tracking performance is limited. An alternative is to use a ROS velocity controller instead of a position controller, and to perform position feedback via an external node (a successive loop closure technique).

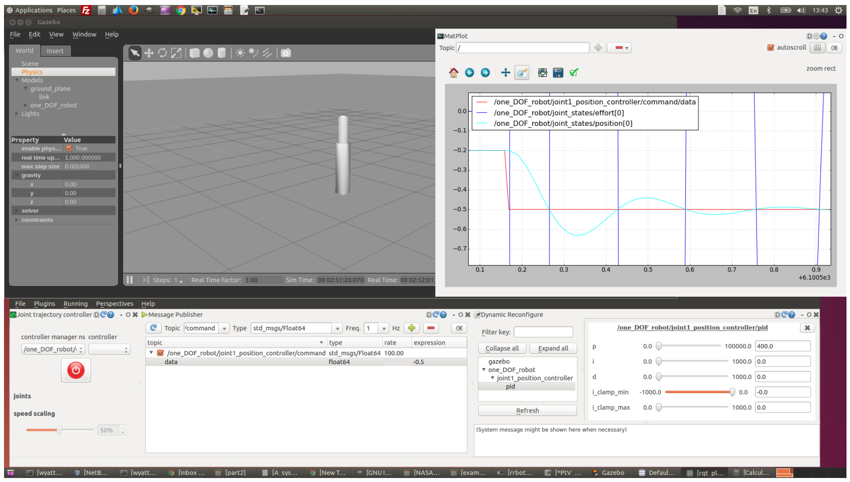


FIGURE 9.2: Servo tuning process for 1-DOF robot: zoom on position response

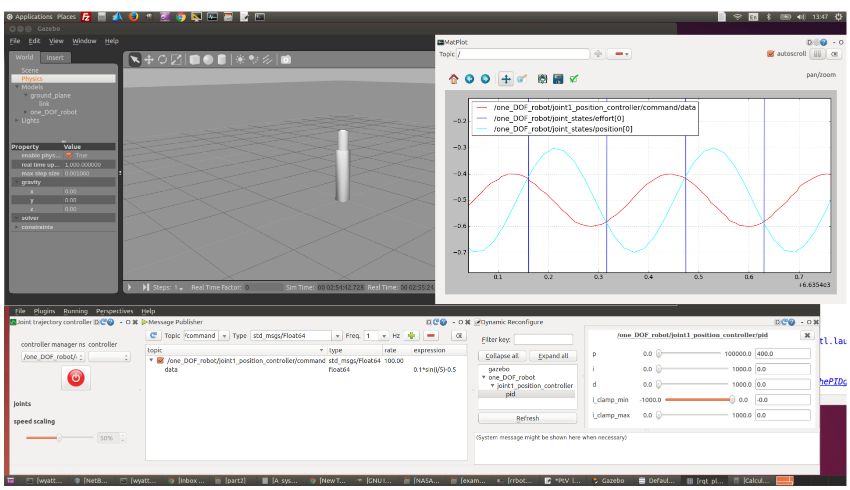


FIGURE 9.3: Servo tuning process for 1-DOF robot, 20 rad/sec sinusoidal command

9.3 AN EXAMPLE VELOCITY CONTROLLER

An alternative launch file that references the same robot model is: `prismatic_1dof_robot_w_jnt_vel_ctl.launch`, which contains:

```
<launch>
  <!-- Load joint controller configurations from YAML file to parameter server -->
  <rosparam file="$(find example_controllers)/control_config/one_dof_vel_ctl_params.yaml"
    command="load"/>

<!-- Convert xacro model file and put on parameter server -->
<param name="robot_description" command="$(find xacro)/xacro.py
   '$(find example_controllers)/prismatic_1dof_robot_description_w_jnt_ctl.xacro'" />

  <!-- Spawn a robot into Gazebo -->
  <node name="spawn_urdf" pkg="gazebo_ros" type="spawn_model"
     args="-param robot_description -urdf -model one_DOF_robot -J joint1 -0.5" />
  <!--start up the controller plug-ins via the controller manager -->
  <node name="controller_spawner" pkg="controller_manager" type="spawner" respawn="false"
    output="screen" ns="/one_DOF_robot" args="joint_state_controller
    joint1_velocity_controller"/>

</launch>
```

As with the position-controlled launch, this launch file loads control parameters from a file (`one_dof_vel_ctl_params.yaml`), it converts the model xacro file into a URDF file and loads it on the parameter server, it spawns the robot model into Gazebo for simulation, and it loads a controller (`joint1_velocity_controller` instead of `joint1_position_controller`). Again, the joint name associated with the controller (joint1) must be part of the controller name loaded (`joint1_velocity_controller`). In this launch file, an additional option is used in spawning the model: `-J joint1 -0.5`. This initializes the robot's joint1 position to the middle of its range of motion.

The YAML file with control parameters, `one_dof_vel_ctl_params.yaml`, is listed below:

```
one_DOF_robot:
  # Publish all joint states -----------------------------------
  joint_state_controller:
    type: joint_state_controller/JointStateController
    publish_rate: 50

  # Velocity Controllers ---------------------------------------
  joint1_velocity_controller:
    type: effort_controllers/JointVelocityController
    joint: joint1
    pid: {p: 200.0, i: 0.0, d: 0.0}
```

This file references the robot name (`one_DOF_robot`) and the joint name(s) (`joint1`). The same process is used to analyze the velocity controller. First starting up Gazebo:

```
roslaunch gazebo_ros empty_world.launch
```

The robot is then spawned using the alternative launch file and gains:

```
roslaunch example_controllers prismatic_1dof_robot_w_jnt_vel_ctl.launch
```

The `rqt_gui` and `rqt_plot` tools are started up:

```
rqt_plot
rosrun rqt_gui rqt_gui
```

Within `rqt_gui`, the `joint1_velocity_controller` is adjusted to change the velocity gain "P", and the commanded and actual joint velocities are plotted, as shown in Fig 9.4.

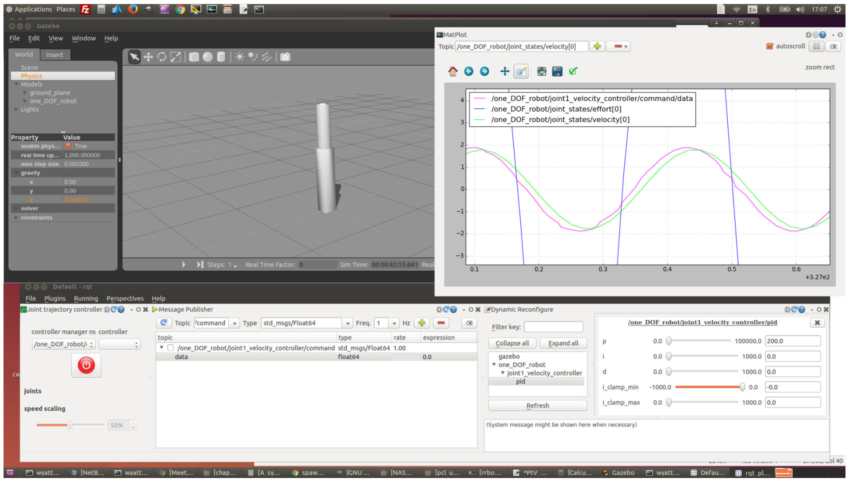


FIGURE 9.4: Velocity controller tuning process for 1-DOF robot

The commanded velocity, in this case, used a separate node:

```
rosrun example_controllers one_dof_sine_commander
```

Responses to the prompts from this node were 0.1m and 3Hz.

Analysis of the velocity controller is somewhat more difficult, since link2 can drift from its center to hit its displacement limits, which introduces distortions when attempting to tune gains. To address this, two adjustments were made. First, gravity was set to 0 in Gazebo, so link2 would not collapse to its lowest level. Second, the robot was started up with joint1 in the the middle of its range of motion. Although it may drift from this average position over time, this allows some time to obtain performance data before interacting with joint limits. An outer loop that considers position can correct for the drift of velocity control. An example is presented next in the context of force feedback.

9.4 AN EXAMPLE FORCE CONTROLLER

A third style of control incorporates consideration of interaction forces to achieve programmable compliance. The 1-DOF prismatic robot model includes the following lines to emulate a force/torque sensor:

```
<!--model a f/t sensor; set up a link and a joint -->
  <link name="ft_sensor_link">
      <collision>
          <origin xyz="0 0 0" rpy="0 0 0"/>
          <geometry>
            <cylinder length="0.005" radius="0.05"/>
```

```
            </geometry>
        </collision>

        <visual>
            <origin xyz="0 0 0" rpy="0 0 0" />
            <geometry>
                <cylinder length="0.005" radius="0.05"/>
            </geometry>
        </visual>

        <inertial>
            <mass value="0.01" />
            <origin xyz="0 0 0" rpy="0 0 0"/>
            <inertia ixx="0.01" ixy="0" ixz="0" iyy="0.01" iyz="0" izz="0.01" />
        </inertial>
    </link>

    <!--ideally, this would be a static joint; until fixed in gazebo, must have dynamic jnt-->
    <joint name="ft_sensor_jnt" type="prismatic">
      <parent link="link2"/>
      <child link="ft_sensor_link"/>
      <origin xyz="0 0 1" rpy="0 0 0"/>
      <axis xyz="0 0 1"/>
      <limit effort="1000.0" lower="0" upper="0.0" velocity="0.01"/>
      <dynamics damping="1.0"/>
    </joint>


<!-- Enable the Joint Feedback -->
<gazebo reference="ft_sensor_jnt">
    <provideFeedback>true</provideFeedback>
</gazebo>
<!-- The ft_sensor plugin -->
<gazebo>
    <plugin name="ft_sensor" filename="libgazebo_ros_ft_sensor.so">
        <updateRate>1000.0</updateRate>
        <topicName>ft_sensor_topic</topicName>
        <jointName>ft_sensor_jnt</jointName>
    </plugin>
</gazebo>
```

An additional link, `ft_sensor_link`, is defined with visual, collision and inertial properties. This link is connected to parent link `link2` via a joint defined as `ft_sensor_jnt`. Ideally, this would be as static joint, but Gazebo (at present) requires that the joint be movable. It is thus defined as a prismatic joint, but with lower and upper joint limits that are equal.

A Gazebo tag brings in the plug-in `libgazebo_ros_ft_sensor.so`, which is associated with the `ft_sensor_jnt`, and which is configured to publish its data to topic `ft_sensor_topic`.

When this model is spawned, the topic `ft_sensor_topic` shows up, and `rostopic info ft_sensor_topic` shows that this topic carries messages of type `geometry_msgs/WrenchStamped`.

If we start up the position-controlled 1-DOF robot:

```
roslaunch example_controllers prismatic_1dof_robot_w_jnt_vel_ctl.launch
```

then command a postion of -0.2 for joint1 (i.e. nearly fully extended), we can observe transients as we drop a weight on the robot, using:

```
roslaunch exmpl_models add_cylinder_weight.launch
```

This introduces a large cylinder model into Gazebo at an initial height of 4m. The cylinder has a mass of 10kg, and it falls towards the ground plane under the influence of gravity (which was reduced to $-5.0m/s^2$ for this example). Figure 9.5 shows the response. Due to the additional inertia supported by the robot, the resonant frequency is reduced and the damping is reduced, resulting in many oscillations before settling down. The force sensor sees a spike of impact when the weight makes contact. As link2 rebounds, the weight is tossed back in the air, and it subsequently makes 3 more impacts (and loses contact 3 more times) before the robot and weight maintain steady contact. From this point on, the robot/payload dynamics is linear, producing a sinusoidal motion due to the robot's position controller acting like a spring.

The actuator effort has the opposite sign as the force-sensor z-value–and the actuator effort magnitude is 5N greater than that of the sensor value. This is because the actuator is supporting the weight of the payload (10kg) plus the weight of link2 (1kg), whereas the force sensor is only influenced by the weight of the payload.

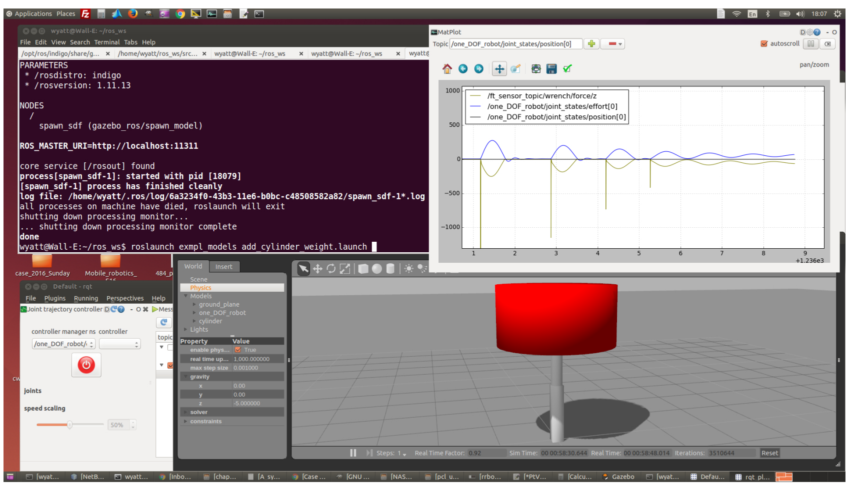


FIGURE 9.5: Force sensor and actuator effort due to dropping 10kg weight on 1-DOF robot

An anomaly of this example is that the joint effort and force sensor equilibrate to the wrong values. In steady state, the actuator effort should balance the weight of link2 plus payload, which should be the weight of the 1kg link2 plus 10kg cylinder. Instead, the system equilibrates to the equivalent of supporting slightly more than 7kg. This may be a bug in the force/torque plug-in, or it may be related to ODE's difficulty handling sustained contact.

The behavior of the position controller responding to the dropped weight is a mass-spring-damper system. However, the response is somewhat unrealistic, since the model has zero Coulomb friction and the transmission is perfectly back-driveable. A similar response can be obtained from more realistic robots by commanding velocities that would correspond to an ideal mass-spring-damper system. An implementation of this is given within the `example_controllers` package as `nac_controller.cpp`. This node subscribes to the joint state and to the force-sensor topic. Within a tight loop it deduces the acceleration that would occur if the system were an ideal mass-spring-damper being acted on by forces exerted at the force-sensor location. The values of the virtual mass, stiffness and damping to be emulated are contained within the variables `M_virt`, `K_virt` and `B_virt`, respectively. The key lines within the main loop of `nac_controller.cpp` are:

```
f_virt = K_virt*(x_attractor-g_link2_pos) + B_virt*(0-g_link2_vel);
f_net =    g_force_z + f_virt;
acc_ideal = f_net/M_virt;
v_ideal+= acc_ideal*dt;
```

In the above, `f_virt` is a virtual force comprised of two terms. The first term is the force exerted by a virtual spring of stiffness `K_virt` stretched between an attractor position, `x_attractor`, and the actual position of link2, `g_link2_pos`, as reported by the joint-state publisher. The second term is due to a virtual damper of damping `B_virt` acting between a reference at zero velocity and link2 moving at velocity `g_link2_vel`, also as reported by the joint-state publisher.

In addition to the virtual forces, there is a physical force exerted at the robot's endpoint, which is sensed by the force sensor. The value of sensed force in the z direction, `g_force_z`, is obtained by subscribing to the force-sensor topic. The sum of the sensed physical force and the virtual forces is the net force, `f_net`. A virtual mass, `M_virt`, acted on by this net force would have an acceleration, `acc_ideal`, of `f_net/M_virt`. Correspondingly, the velocity of this mass, `v_ideal`, is the time-integral of the ideal acceleration. This model-based velocity is then sent as a command to a stiff velocity controller. Preferably, the value of `M_virt` should be close to the actual mass of link2 (thus making this controller a "natural" admittance controller).

To run this controller, again start up Gazebo:

```
roslaunch gazebo_ros empty_world.launch
```

Then spawn the robot using velocity control:

```
roslaunch example_controllers prismatic_1dof_robot_w_jnt_vel_ctl.launch
```

Then start the NAC node:

```
rosrun example_controllers nac_controller
```

The example NAC controller was assigned values of `M_virt` = 1.0 kg, `K_virt` = 1000 N/m and `B_virt` = 50 N/(m/sec). With this controller running, and with gravity set to -9.8 m/s^2, the 10kg weight was dropped on the robot, resulting in the transient shown in Figure 9.6. By tuning the values of `K_virt` and `B_virt`, the transient behavior can be adjusted to a desirable response.

Although the controllers here have been illustrated only in one degree of freedom, the approach generalizes to complex arms. Figure 9.7 shows a 7-DOF arm holding a weight using Natural Admittance Control. Source code for this example is in package `arm7dof_nac_controller`. Conceptually, the approach is the same as the 1-DOF controller. Sensed forces and virtual forces are combined (as vectors rather than scalars), and robot inertia (a matrix, rather than a constant) is considered to compute accelerations (in joint space) expected to result from the effort interactions. The joint accelerations are integrated to compute equivalent joint velocities, and these velocities are commanded to joint velocity servos of the robot.

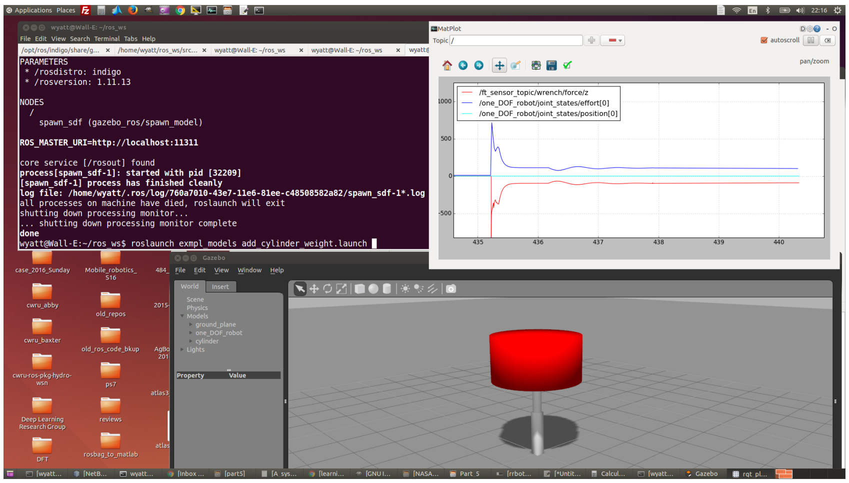


FIGURE 9.6: NAC controller response to dropping 10kg weight on 1-DOF robot

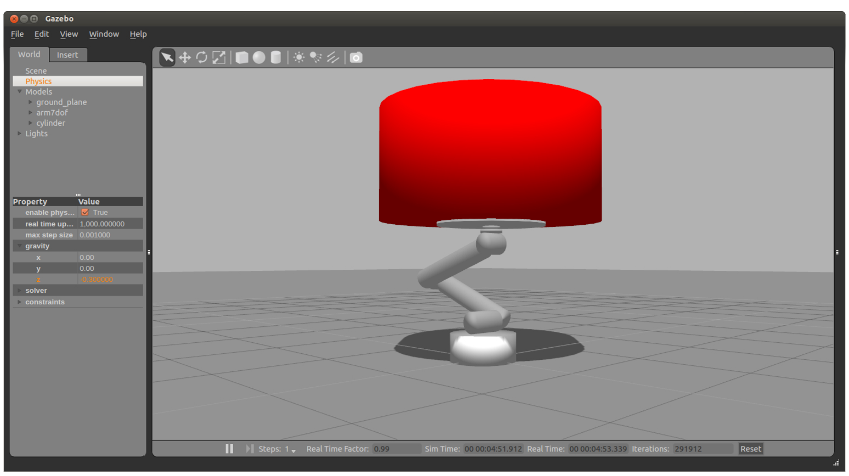


FIGURE 9.7: Seven-DOF arm catching and holding a weight using NAC

CHAPTER 10

Robot Arm Kinematics

CONTENTS

Robot arm kinematics is a starting point for most textbooks in robotics. A fundamental question is, given a set of joint displacements, where is the end effector in space? To be explicit, one must define reference frames. A frame of interest on the robot may be its tool flange, or some point on a tool tip (e.g. welder, laser, glue dispenser, grinder, ...) or a frame defined with respect to a gripper. It is useful to compute the pose of the defined frame of interest with respect to the robot's base frame. In a separate transform, one can express the robot's base frame with respect to a defined world frame. The task of computing the pose of a robot's end effector with respect to its base frame given a set of joint displacements constitutes forward kinematics. For open kinematic chains (i.e., most robot arms), forward kinematics is always simple to compute, yielding a unique, unambiguous pose as a function of joint displacements.

While forward kinematics is relatively simple, the more relevant question is typically the inverse: given a desired pose of the end effector frame, what set of joint angles will achieve this objective? Solving this inverse-kinematics relationship is plagued with problems. There may be no viable solutions (goal is not reachable), there may be a finite number of equally-valid solutions at very different sets of joint angles (typical for 6-DOF robots) , or there may be an infinity of solutions (typical for robots with more than 6 joints). Further, while forward-kinematics solutions can be obtained using a standard procedure, solving for inverse-kinematic solutions may require ad hoc algorithms exploiting creative insights.

The kinematics-and-dynamics package "KDL" (see, e.g. `http://wiki.ros.org/orocos_kinematics_dynamics`) is a popular tool for computing forward kinematics and robot dynamics. One of its capabilities is that it can reference a robot model on the parameter server to obtain the information necessary to compute forward kinematics. Thus, any open-chain robot in ROS already has a corresponding forward-kinematics solver (via KDL). KDL also computes Jacobians (between any two named frames).

Unfortunately, the more useful and more difficult problem of inverse kinematics is not handled well in general. If the robot of interest has specific properties (e.g. 6DOF with a spherical wrist), then inverse-kinematic solutions can be computed analytically, yielding exact expressions for all possible solutions. However, robots with "redundant" kinematics (more than 6DOF) are increasingly popular, due to their enhanced dexterous workspace. Further, many robot designs do not conform to the equivalent of a spherical wrist.

Numerical techniques (gradient search using a kinematic Jacobian) are often employed when analytic inverse-kinematic solutions are not available. As is typical with numerical searches, though, this approach can be slow, can fail to converge, can get stuck in local minima, and (when successful) typically returns a single solution when multiple solutions or an infinity of solutions exist.

When working with a particular robot, if an inverse-kinematics package is available, the available solution may be sufficient. If a suitable package is not already available, it may be necessary to design one's own.

This chapter will introduce examples of forward and inverse kinematics libraries and use these to illustrate common issues in robot kinematics.

10.1 FORWARD KINEMATICS

A simple example to illustrate kinematics is in the package "rrbot." This robot model borrows from the on-line tutorial at `http://gazebosim.org/tutorials?tut=ros_urdf` for the "rrbot" (a 2-DOF robot with two revolute joints). In the subdirectory "model" the file "rrbot.xacro" defines a simple, planar manipulator with three links and two joints. Figure 10.1 shows Gazebo and Rviz views of this robot, which was launched using:

```
roslaunch gazebo_ros empty_world.launch
roslaunch rrbot rrbot.launch
```

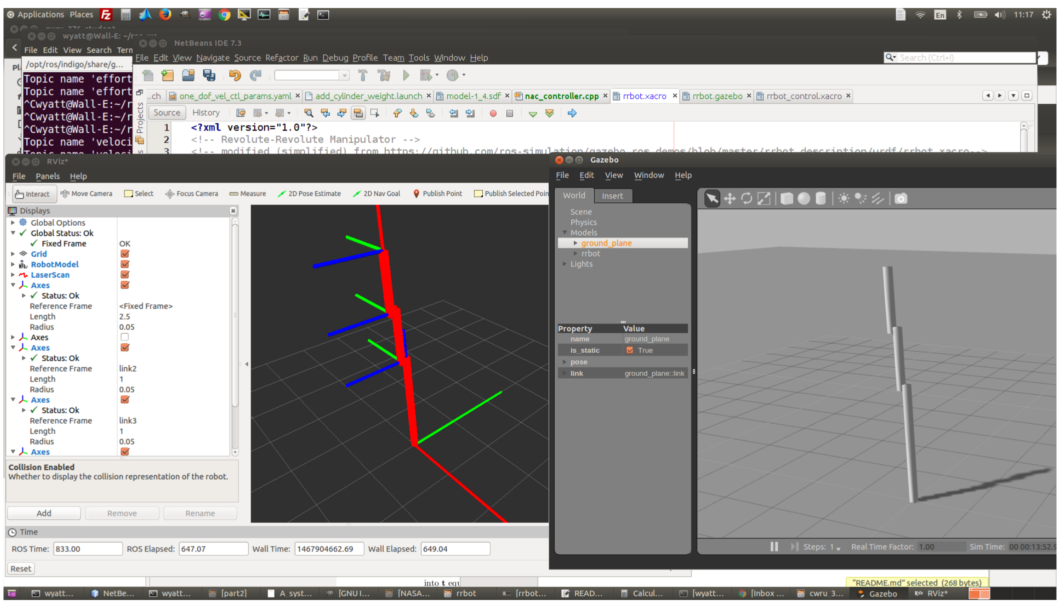


FIGURE 10.1: Gazebo and Rviz views of rrbot

The rrbot has frames defined for each of its links. To be consistent with Denavit-Hartenberg convention, the frames were defined such that the z axes pass through revolute joint axes (for joints 1 and 2), as follows:

```
<joint name="joint1" type="continuous">
  <parent link="link1"/>
  <child link="link2"/>
  <origin xyz="${height1 - axel_offset} 0  ${width} " rpy="0 0 0"/>
  <axis xyz="0 0 1"/>
  <dynamics damping="0.7"/>
</joint>
<joint name="joint2" type="continuous">
  <parent link="link2"/>
```

```
    <child link="link3"/>
    <origin xyz="${height2 - axel_offset*2} 0 ${width} " rpy="0 0 0"/>
    <axis xyz="0 0 1"/>
    <dynamics damping="0.7"/>
  </joint>
```

It is also convenient to define a frame of interest at the tip of the last link. To do so, a fictitious joint (a "static" joint) is defined in rrbot.xacro as follows:

```
  <joint name="flange_jnt" type="fixed">
    <parent link="link3"/>
    <child link="flange"/>
    <origin xyz="${height3 - axel_offset} 0  0" rpy="0 0 0"/>
  </joint>
```

By defining this frame, "tf" will compute transformations out to this frame of interest. The fictitious link "flange" is defined simply as:

```
  <link name="flange"/>
```

Since the link "flange" is connected by a static joint, it is not necessary to define inertial, visual or collision components. The same technique is useful for mounting sensors (rigidly) to robots or defining tool frames on specific end effectors.

With the above joint definitions, the equivalent Denavit-Hartenberg parameters, as defined in the header file `rrbot/include/rrbot/rrbot_kinematics.h`, are:

```
const double DH_a1=0.9; //link length: distance from joint1 to joint2 axes
const double DH_a2=0.95; //link length: distance from joint2 axis to flange-z axis

const double DH_d1 = 0.1;// offset along parent z axis from frame0 to frame1
const double DH_d2 = 0.0; // zero offset along parent z axis from frame1 to flange

const double DH_alpha1 = 0; //joint1 axis is parallel to joint2 axis
const double DH_alpha2 = 0; //joint2 axis is parallel to flange z-axis

//could define robot "home" angles different than DH home pose; reconcile with these ↩
    offsets
const double DH_q_offset1 = 0.0;
const double DH_q_offset2 = 0.0;
```

Converting between a URDF model and a DH model can be confusing. In the present case, having defined the joint axes consistent with DH convention, the z-axes of DH frames 1 and 2 are parallel (as well as the z axis of the flange frame), and consequently the "alpha" angles are zero, and the "a" parameters are the link lengths. Since there is an offset from link1 to link2 along the z direction, there is also a non-zero "d1" parameter.

In the rrbot package, the source file `rrbot_fk_ik.cpp` compiles to a library than computes forward and inverse kinematics for the rrbot. The function `compute_A_of_DH()` returns a 4x4 coordinate-transform matrix, as described in Chapter 4. In general, a coordinate transform between successive DH frames follows from specification of the 4 DH parameters, as shown in `compute_A_of_DH()`.

```
Eigen::Matrix4d compute_A_of_DH(double q, double a, double d, double alpha) {
    Eigen::Matrix4d A;
    Eigen::Matrix3d R;
    Eigen::Vector3d p;

    A = Eigen::Matrix4d::Identity();
    R = Eigen::Matrix3d::Identity();
    //ROS_INFO("compute_A_of_DH: a,d,alpha,q = %f, %f %f %f",a,d,alpha,q);
    double cq = cos(q);
```

```
    double sq = sin(q);
    double sa = sin(alpha);
    double ca = cos(alpha);
    R(0, 0) = cq;
    R(0, 1) = -sq*ca; //% - sin(q(i))*cos(alpha);
    R(0, 2) = sq*sa; //%sin(q(i))*sin(alpha);
    R(1, 0) = sq;
    R(1, 1) = cq*ca; //%cos(q(i))*cos(alpha);
    R(1, 2) = -cq*sa; //%
    //%R(3,1)= 0; %already done by default
    R(2, 1) = sa;
    R(2, 2) = ca;
    p(0) = a * cq;
    p(1) = a * sq;
    p(2) = d;
    A.block<3, 3>(0, 0) = R;
    A.col(3).head(3) = p;
    return A;
}
```

A general procedure for computing forward kinematics then simply involves computing each successive 4x4 transformation matrix, then multiplying these matrices together. This is implemented in the function `fwd_kin_solve_()` within the `rrbot_fk_ik.cpp` library as follows:

```
Eigen::Matrix4d Rrbot_fwd_solver::fwd_kin_solve_(Eigen::VectorXd q_vec) {
    Eigen::Matrix4d A = Eigen::Matrix4d::Identity();
    //%compute A matrix of frame i wrt frame i-1 for each joint:
    Eigen::Matrix4d A_i_iminusi;
    Eigen::Matrix3d R;
    Eigen::Vector3d p;
    //cout << "A_base_link_wrt_world_:" << endl;
    //cout << A_base_link_wrt_world_ << endl;
    for (int i = 0; i < NJNTS; i++) {
        A_i_iminusi = compute_A_of_DH(q_vec[i] + DH_q_offsets[i], DH_a_params[i],
            DH_d_params[i], DH_alpha_params[i]);
        A_mats_[i] = A_i_iminusi;
    }
    //now, multiply these together
    //A_base_link_wrt_world_ * A_frame1_wrt_base_link = A_frame1_wrt_world
    A_mat_products_[0] = A_base_link_wrt_world_ * A_mats_[0];

    for (int i = 1; i < NJNTS; i++) {
        A_mat_products_[i] = A_mat_products_[i - 1] * A_mats_[i];
    }
    //Eigen::Vector4d test_0_vec; //some test code to get the coordinates of the ↩
        flange frame
    //test_0_vec<<0,0,0,1;
    //cout<<"test Amat prod: "<<A_base_link_wrt_world_*A_mats_[0]*A_mats_[1] *↩
        test_0_vec<<endl;
    return A_mat_products_[NJNTS - 1]; //tool flange frame
}
```

This same forward-kinematics code is applied as well to additional example robots in `learning_ros/Part_5`, including a 6DOF ABB robot (model IRB120), the Baxter robot (with 7DOF arms) and a robot model based on the NASA/Goddard sattellite-servicer arm (arm7dof). To customize the forward-kinematics code for a specific robot, it is only necessary to define the number of joints and values for the DH parameters in the respective header file. If using the KDL package, this much is not even necessary, since KDL will parse the robot model to obtain the parameter values. With the code described here, one must manually assure that the values in the URDF file stay synced with the corresponding DH values in the kinematics header file.

Forward and inverse kinematics can be tested using the test function `test_rrbot_fk.cpp`. The rrbot, the test function, and supporting tools are brought up with:

```
roslaunch gazebo_ros empty_world.launch
roslaunch rrbot rrbot.launch
rosrun rrbot test_rrbot_fk
rosrun rqt_gui rqt_gui
rosrun tf tf_echo world flange
```

A screenshot with these nodes running is shown in Figure 10.2. The robot can be moved

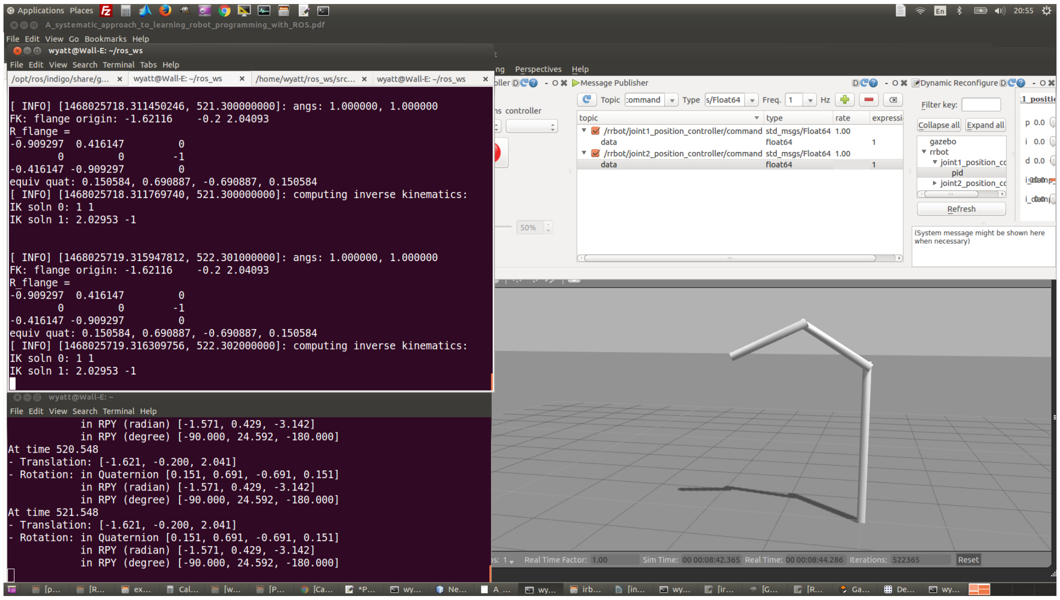


FIGURE 10.2: Gazebo, tf-echo and fk test nodes with rrbot

to arbitrary poses using `rqt_gui`. In the example shown, the joints are both commanded to angles of 1.0 rad. The `test_rrbot_fk` node subscribes to `joint_states` and prints out the published joint values, confirming that the robot is at the commanded joint angles. (In this test, gravity was set to zero, so there is no joint error from gravity droop). Forward kinematics of the flange frame with respect to the base (world) frame is computed and displayed as:

```
FK: flange origin: -1.62116    -0.2 2.04093
```

To test this result, the `tf_echo` node is run to get the flange frame with respect to the world frame. This function displays:

```
- Translation: [-1.621, -0.200, 2.041]
```

To the precision displayed, the results are identical.

A second, more realistic test can be run with:

```
roslaunch irb120_description irb120.launch
rosrun irb120_fk_ik irb120_fk_ik_test_main
rosrun rqt_gui rqt_gui
rosrun tf tf_echo world link7
```

This launches a model of an industrial robot–the ABB IRB120. Using `rqt_gui`, the 6 joints are commanded to the values [1, 1, -1, 2, 1, 1]. The resulting pose is shown in Figure 10.3. The `tf_echo` node prints out the pose of the flange frame (link7) with respect to the world. It displays:

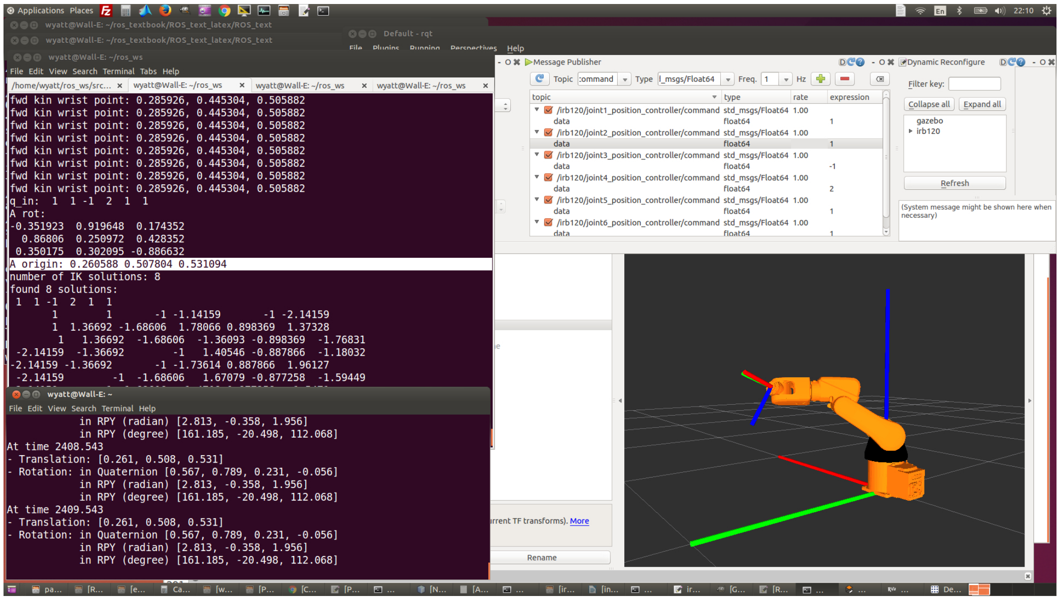


FIGURE 10.3: Rviz, tf-echo and fk test of ABB IRB120

```
- Translation: [0.261, 0.508, 0.531]
```

The test node `irb120_fk_ik_test_main` subscribes to the joint states and uses these values to compute forward kinematics. This node prints out:

```
A origin: 0.260588 0.507804 0.531094
```

Again, to the precision displayed, the results are identical. Additional spot checks yield similar results.

10.2 INVERSE KINEMATICS

The code used to compute forward kinematics for the 2-DOF rrbot and the 6-DOF is virtually identical, except for the number of joints and the specific numerical values of the DH parameters. The algorithmic approach is valid in general for all open kinematic chains. In contrast, inverse kinematics can require robot-specific algorithms and can be problematic with handling multiple solutions, which is exacerbated with kinematically-redundant robots.

Starting with the 2-DOF rrbot, inverse kinematics can be computed using the law of cosines. The program `rrbot_fk_ik.cpp` includes a class for forward kinematics and a class for inverse kinematics. The inverse-kinematics function `ik_solve()` takes an argument for the desired end-effector pose (as an Eigen::Affine3 object), and it populates an std::vector of Eigen::Vector2 objects comprising alternative solutions. There can be 0, 1 or 2 solutions for the rrbot.

With only 2 degrees of freedom, it is (typically) only possible to satisfy two objectives. For example, one cannot find solutions for an arbitrary desired position (x,y,z) of the tool flange. The rrbot can only reach flange coordinates for which y= 0.2. If the requested flange position is (x, 0.2, z), then solutions may be possible.

The function `ik_solve()` first calls `solve_for_elbow_ang()`, which returns 0, 1 or 2 reachable elbow (joint2) angles. These angles follow from the realization that the distance from the shoulder to the flange is a function only of joint2 (not joint1). The value of joint2 angle can be found from the law of cosines: $c^2 = a^2 + b^2 - 2ab\cos(C)$. In this case, a and b are the link lengths of the two movable links and c is the desired distance from the robot's shoulder (joint1) to the tool flange (tip if distal link). These three lenghts form a triangle, and the angle C is the angle opposite the leg with length c. This relationship can be solved for angle C using an inverse cosine. However, there are zero, one or two solutions for C in this equation. If, for example, a flange pose is requested that is beyond the reach of the robot (i.e. $c > a + b$), then there are no solutions for C.

When the requested position is within reach of the robot, there are typically 2 solutions. (A special case occurs when the arm is fully extended, precisely reaching a desired position; in this case there is only one solution). Most commonly, if there are any solutions, then there are two solutions (commonly referred to as "elbow up" and "elbow down").

For each of the elbow solutions, one can find the corresponding shoulder angle (joint1 angle) that achieves the desired flange position. This is computed using the function `solve_for_shoulder_ang()`. This function reduces the shoulder-angle problem to one of solving an equation of the form: $r = A\cos(q) + B\sin(q)$ for the value of q. This equation occurs frequently in inverse-kinematics algorithms. This form of equation is solved within the function `solve_K_eq_Acos_plus_Bsin(K, A, B, q_solns)`. There is some amiguity, though, since 2 solutions are obtained, but only one of these is consistent with a correct robot pose. For a given elbow angle, both candidate shoulder angles are tested using forward kinematics, and the correct shoulder angle is thus identified. Typically, if the desired point is reachable, there will be two valid solutions, (q_{1a}, q_{2a}) and (q_{1b}, q_{2b}).

As an example, the desired flange position of:

```
flange origin: -1.62116    -0.2 2.04093
```

has two solutions, as computed and displayed by `test_rrbot_fk`:

```
IK soln 0: 1 1
IK soln 1: 2.02953 -1
```

These solutions are illustrated in Fig 10.4

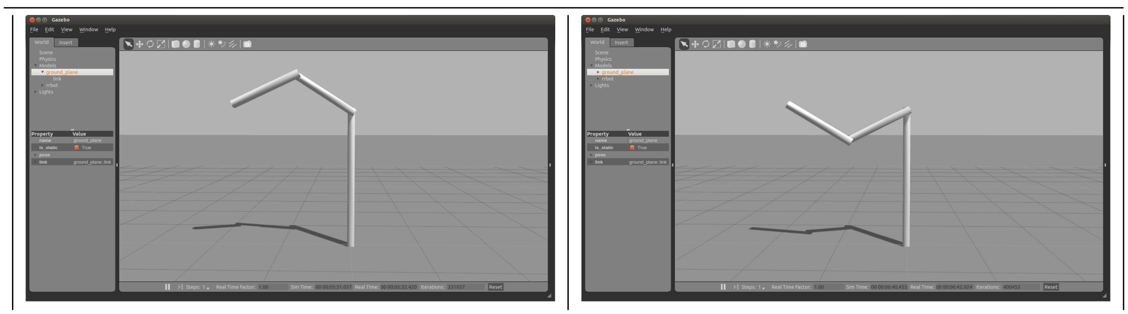


FIGURE 10.4: Two IK solutions for the rrbot: elbow up and elbow down

Inverse kinematics for the ABB IRB120 is more complex. For this robot, however, the design incorporates a "spherical wrist" (for which the last three joint axes intersect in a point). Existence of a spherical wrist simplifies the IK problem, since one can solve for position of the "wrist point" (the point of intersection of the last 3 joint axes) in terms of the proximal joints (joints 1, 2 and 3, in the case of the IRB120). Transformation matrices, law of cosines, and solutions of $r = A\cos(q) + B\sin(q)$ are again exploited to solve for joint-angles q1, q2 and q3. In addition to elbow-up and elbow-down solutions, the robot can also rotate 180-degrees about its turret (base) to reach backwards to achieve two more elbow-up, elbow-down solutions. There thus can be 4 distinct solutions that place the wrist point at its desired position. For each of these solutions, the spherical wrist offers two distinct solutions. All together, these comprise 8 IK solutions. A specific example for the irb120 robot via the `irb120_fk_ik_test_main` node starts with the robot at joint angles (1,1,-1,2,1,1). Forward kinematics was computed, then this flange pose was used to generate inverse-kinematic solutions. For this case, there were 8 solutions, including re-discovery of the original joint angles, (1,1,-1,2,1,1). The eight solutions displayed were:

```
found 8 solutions:
 1          1            -1        2          1          1
 1          1            -1       -1.14159   -1         -2.14159
 1          1.36692      -1.68606  1.78066    0.898369   1.37328
 1          1.36692      -1.68606 -1.36093   -0.898369  -1.76831
 -2.14159  -1.36692      -1        1.40546   -0.887866  -1.18032
 -2.14159  -1.36692      -1       -1.73614    0.887866   1.96127
 -2.14159  -1            -1.68606  1.67079   -0.877258  -1.59449
 -2.14159  -1            -1.68606 -1.4708     0.877258   1.5471
```

These solutions show that the first three joint angles appear twice in separate solutions, which differ only in the last three (wrist) joint angles. All 8 of the solutions for this example are shown in Fig 10.5. The two distinct wrist solution for each combination of (q1,q2,q3) may not be easily discernible, but these solutions are quite distinct in (q4,q5,q6).

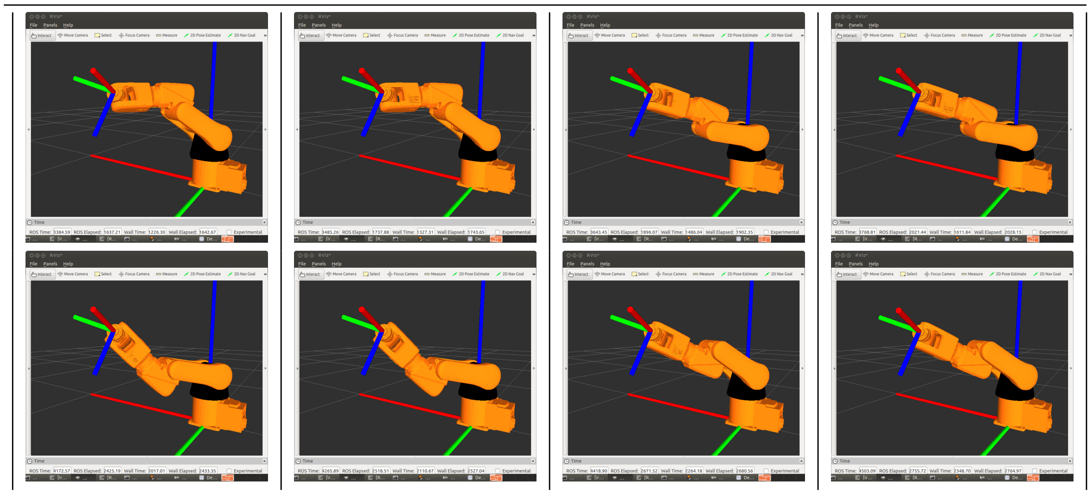


FIGURE 10.5: Example of 8 IK solutions for the ABB IRB120

Although the example shown has 8 IK solutions, the number of solutions can vary. If a requested point is beyond the reach of the robot (or within the body of the robot) there will be no viable solutions. If IK solutions would require moving one or more joints beyond their allowable range of motion, there will be fewer than 8 valid solutions. If the desired pose is reachable at a singular pose (when two or more joints align such that their joint axes are colinear), then there will be an infinity of IK solutions (describable as a function of the sum or difference of the joint angles of the aligned joints).

With an array of IK solutions, one requires an algorithmic means to choose among the solutions to execute one of them. Choosing a “best” solution can be ambiguous, but one expectation is that the robot will not jump suddenly, e.g., between elbow-up and elbow-down solutions during execution of a trajectory.

How to choose among candidate solutions is even more challenging when the robot has more than 6 degrees of freedom (or, more generally, when the robot degrees of freedom is larger than the number of task constraints). In such cases, there is a continuum (or multiple, distinct continua) of solutions) to choose from. An example is in `arm7dof`.

The robot model in `arm7dof_model` is based on the NASA "restore" arm design being considered for servicing satellites in space. A Gazebo view of a proposed NASA design is shown in Fig 10.6. Frame assignments for this 7-DOF arm are shown in Fig **??**.

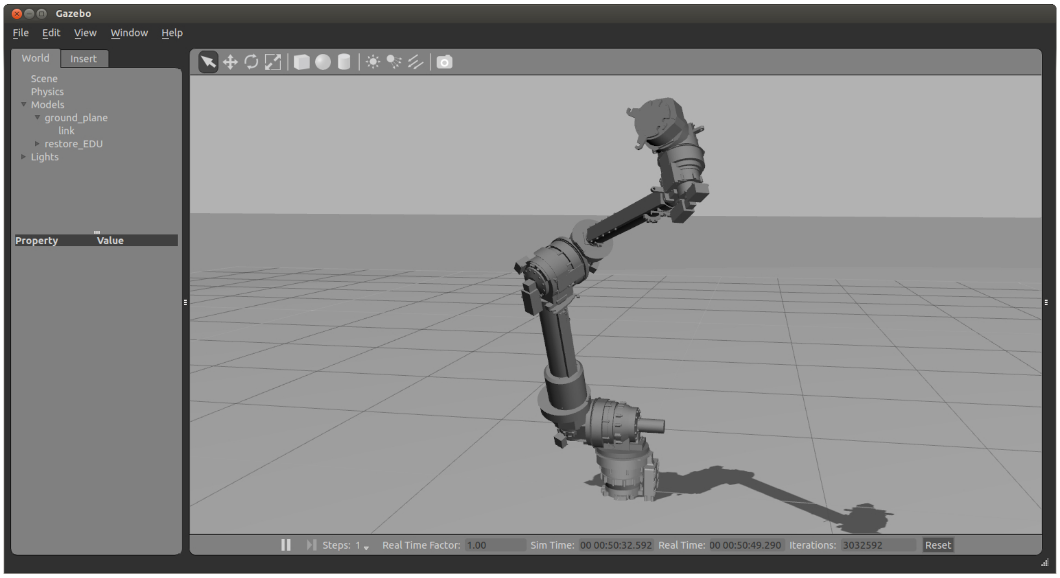


FIGURE 10.6: Proposed NASA satellite-servicer arm

The model in package `arm7dof_model` is a simplified, abstracted version of the sattelite servicer arm. This model (with a large disk appended to the tool flange) appears as in Fig 10.8. This design incorporates a spherical wrist, which simplifies the inverse-kinematics algorithm. For each solution of (q1,q2,q3,q4) that places the wrist point in a desired position, there are two solutions (q5,q6,q7) that achieve the desired flange pose (satisfying both desired position and orientation of the flange). However, if the desired wrist point is reachable, there is typically a continuum (or multiple, disconnected continua) of solutions in the space (q1,q2,q3,q4).[1]

An approach to inverse kinematics for this arm is presented in An2014 [**?**], in which solutions are obtained as a function of a free variable, such as the elbow-orbit angle or the base turret angle. The latter approach is taken here. Within the package `arm7dof_fk_ik`, a forward and inverse kinematics library is designed, which contains a key function `ik_wrist_solns_of_q0()`. This function accepts coordinates of a desired wrist point along with a desired turret angle (q1) and it returns all of the viable solutions for joint angles (q2, q3, q4). As with the ABB 6-DOF robot, once q1 is constrained, only (these) 3 joints imply the wrist-point coordinates, and there can be up to 4 distinct solutions. Also like the ABB 6-DOF robot, each of these wrist-point solutions theoretically has two solutions for (q5,q6,q7) (though these may or may not be reachable within the robot's joint range-

[1] By DH convention, joint-angle numbering starts at 1 at the most proximal joint and increments sequentially to the last joint. However, indexing in C++ starts from 0, and thus joint numbers are sometimes numbered from 0.

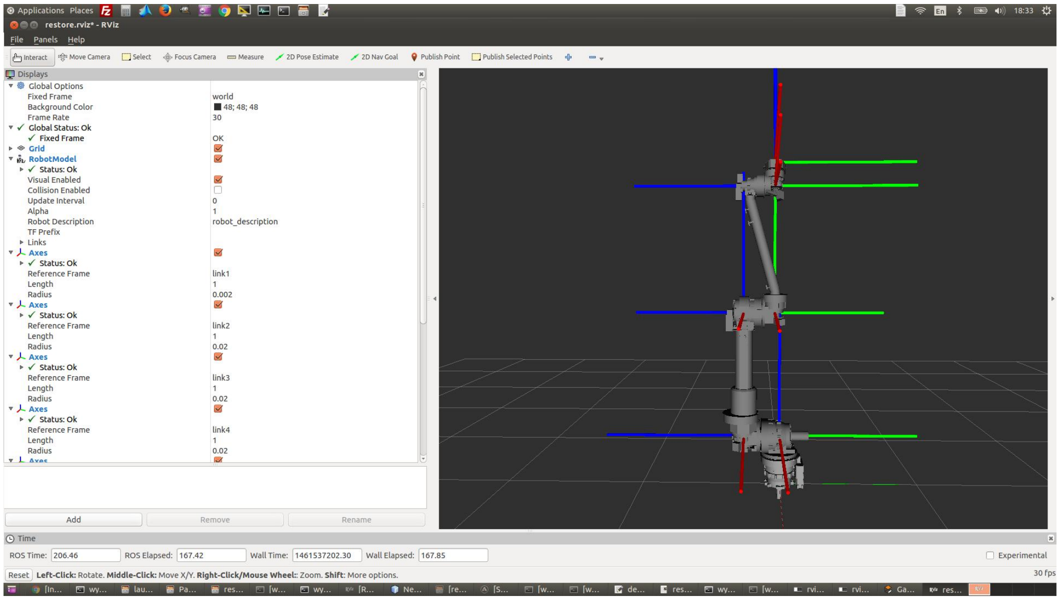


FIGURE 10.7: Proposed NASA satellite-servicer arm

of-motion constraints). In at least some instances, there thus will be 8 distinct, viable IK solutions (at given angle q1). For the pose in Fig 10.8, there are 6 distinct, reachable IK solutions at q1=0.

The choice of base angle, q1, for optimal an optimal IK solution is unclear. In fact, the best choice of q1 may only be knowable in a broader context. To defer choosing a value of q1, one can compute IK solutions for a range of q1 candidates, sampled at some chosen resolution. The function `ik_solns_sampled_qs0()` accepts a specified flange pose and fills in a (reference variable) std::vector of 7-DOF IK solutions. The sampling resolution of q1 is set by the assigned value of `DQ_YAW`, which is set to 0.1 rad in the header file `arm7dof_kinematics.h`. For the pose shown in Fig 10.8, 228 valid IK solutions were computed at a sampling resolution of 0.1rad over q1 from 0 to 2π. This example was computed using the following code in package `arm7dof_fk_ik`:

```
roslaunch gazebo_ros empty_world.launch
roslaunch arm7dof_model arm7dof_w_pos_controller.launch
rosrun rqt_gui rqt_gui (to command joint angles)
rosrun arm7dof_fk_ik arm7dof_fk_ik_test_main2
```

Since all of these solutions are computed analytically, there are no numerical issues, such as failure to converge. Further, the analytic approach yields a large number of options, not simply a single solution (in contrast to a numerical approach). The solution process is also relatively efficient. For the example shown, computing 228 IK solutions required approximately 30msec on a laptop using one core of a 2GHz Intel i7.

The number of solutions expands dramatically in singular poses. Figure **??** shows the 7DOF arm in a singular pose, for which the joint axis of joint 7 is colinear with the joint axis of joint 5. In such poses, given any solution, one can generate a space of solutions by rotating joint7 by $\Delta\theta$ and simultaneously rotating joint5 by $-\Delta\theta$, thus generating a continuum of solutions even at fixed q1.

For some robot arms, an analytic solution may not be known. In such cases, one must

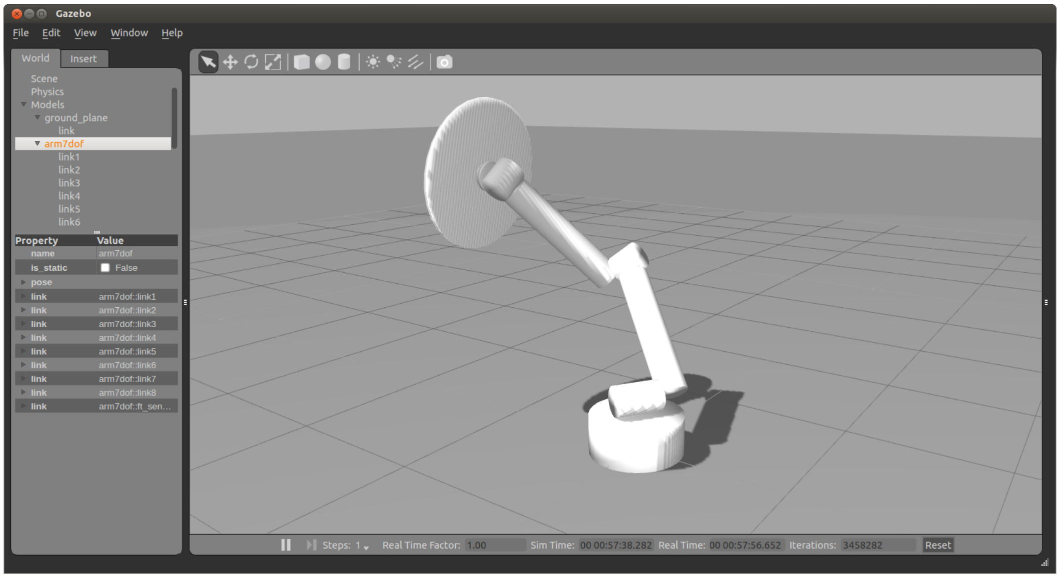


FIGURE 10.8: Approximation of proposed NASA satellite-servicer arm

resort to numerical methods. If an approximate analytic solution is known, it can be useful to solve for approximate solutions and use these as starting points for a numerical search. In the numerical search, the manipulator Jacobian is a key component. Jacobians–like forward kinematics–are easy to compute and are well behaved. The example kinematic libraries provided include Jacobian computations.

CHAPTER 11

Arm Motion Planning

CONTENTS

Robot arm motion planning is a broad field in itself with a variety of sub-problems. For example, one may be concerned with the optimal trajectory a robot can execute to get from some start pose to some final pose in minimum time, subject to constraints on the actuator efforts and joint ranges of motion. Another variant is planning arm motions that optimize the mechanical advantage of available actuators, e.g. as in weight lifting. A common problem is planning how to command a robot's joints to move from an initial pose to a final pose (expressed either in joint space or in task space) while avoiding collisions (with the robot itself or with entities in the environment). The task to be performed may require a specified end-effector trajectory, e.g. as in laser cutting, sealant dispensing, or seam welding. In such cases, the velocity of the endpoint may be constrained by the task (e.g. for optimal material removal rates, torch speeds or dispensing rates). In demanding cases, motion planning must be performed in real time, e.g. to generate motion to catch a ball. At the other extreme, some motion plans for stereotyped behaviors (e.g., folding arms into a compact pose for transportation) may be computed off-line with the results saved for execution as a rote skill. For supervisory control, e.g. remote control with excessive time delay, one may plan an motion and preview the result in simulation before committing the remote system to execute the approved plan.

Arm motion planning packages exist in ROS, and the sophistication and breadth of options may be expected to grow. In the current context, a single example planning approach will be described here to illustrate how arm-motion planning can integrate within a ROS-based hierarchy.

The example planning problem considered here assumes that one has a specified 6-DOF path in Cartesian space that the robot's tool flange must satisfy. For example, a 7DOF arm may be required to move a knife blade along a specified path (along a workpiece) while maintaining a specified orientation of the knife (with respect to the workpiece), equivalently constraining 6DOF poses of the tool flange. It will be assumed that the Cartesian path can be adequately approximated by N_{path} sequential samples along the continuous path. At each sample point along the path, the 7-DOF arm typically would have infinitely many inverse-kinematic options. Pragmatically, this may be approximated by sampled IK solutions, on the order of hundreds of options per Cartesian knot point. Even in this simplified scenario, the planning problem can be intimidating. For example, if there were 100 samples along the Cartesian trajectory, and if 200 IK solutions are computed for each such pose, then there would be 200^{100} joint-space path options, which is overwhelmingly large.

Fortunately, this planning problem can be simplified dramatically using dynamic programming. The approach described here is comprised of two layers: a Cartesian-space planner and an underlying joint-space planner.

11.1 CARTESIAN MOTION PLANNING

The package `cartesian_planner` contains the source file `arm7dof_cartesian_planner.cpp`, which defines a library `CartTrajPlanner`. The role of this library is to sample points along a Cartesian path, compute IK options for each sampled pose along the path, and (with the help of the joint-space planner) find the optimal sequence of joint-space solutions to traverse the desired Cartesian path.

In general, the Cartesian path to be followed is task dependent. The Cartesian planner in `cartesian_planner` is illustrative of how to sample a path and compute an attractive joint-space trajectory, but it is limited in its generality. The Cartesian planner described here has three motion-planning options:

- Specify start and end poses with respect to the arm's base. Only the orientation of the end pose will be considered. Orientation of start pose is ignored. A sequence of Cartesian poses is generated that maintains constant orientation while moving in a straight line from the start position to the end position.
- Specify a start pose, `q_vec`, in joint space. Specify a goal pose in Cartesian space. Sample points (Cartesian poses) along a trajectory will be generated such that orientation of the goal pose will be obtained quickly, then preserved throughout the linear move.
- Specify a start pose, `q_vec`, and a desired delta-p Cartesian motion. Sampled Cartesian poses will be generated starting from the initial pose and translating in a straight line by the specified displacement vector while holding orientation fixed at at the initial orientation.

The node `example_arm7dof_cart_path_planner_main.cpp` shows how to use the Cartesian planner `arm7dof_cartesian_planner.cpp`, which relies on support from the corresponding kinematics library `arm7dof_fk_ik` and from the generic joint-space planner `joint_space_planner`.

Running this node will produce an output file `arm7dof_poses.path` comprised of joint-space poses that produce Cartesian motion with the tool flange orientation constant. In this example, the desired motion specifications are hard-coded in the main program, corresponding to maintaining orientation of the tool flange pointing up while translating at y-desired, z-desired from x-start to x-end. Samples along this path are computed every 5cm. These values can be edited to test alternative motions. With a roscore running, the planner example can be run with:

```
rosrun  cartesian_planner example_arm7dof_cart_path_planner_main
```

For this example, there are 60 samples computed along the Cartesian path. At each sample point, inverse-kinematic options are computed at increments of 0.1 radians of the base joint. This results in roughly 200 IK solutions per Cartesian sample point (ranging from 130 to 285 IK solutions at each Cartesian pose, for this example).

Key lines of the code `example_arm7dof_cart_path_planner_main.cpp` are:

```
    CartTrajPlanner cartTrajPlanner; //instantiate a cartesian planner object
```

```
R_gripper_up = cartTrajPlanner.get_R_gripper_up();

//specify start and end poses:
a_tool_start.linear() = R_gripper_up;
a_tool_start.translation() = flange_origin_start;
a_tool_end.linear() = R_gripper_up;
a_tool_end.translation() = flange_origin_end;

//do a Cartesian plan:
found_path = cartTrajPlanner.cartesian_path_planner(a_tool_start, a_tool_end, ↩
    optimal_path);
```

The planner option invoked in this example is specification of both start and end Cartesian poses. After calling `cartesian_path_planner()`, a joint-space plan will be returned in `optimal_path`. This plan is written to an output file called `arm7dof_poses.path`, which consists of 61 lines (one for each Cartesian sample point), the first few lines of which are:

```
2.9, -0.798862, 0.227156, -1.35163, -0.191441, 2.12916, -3.16218
2.9, -0.752214, 0.210785, -1.42965, -0.173242, 2.1637, -3.15245
2.9, -0.707348, 0.196287, -1.50431, -0.156966, 2.19621, -3.1424
```

Each line specifies 7 joint angles corresponding to a joint-space pose that is an IK solution of the corresponding desired Cartesian-space pose. Selection of optimal joint-space poses from among the candidate IK solutions is performed by a supporting library, the joint-space planner.

11.2 DYNAMIC PROGRAMMING FOR JOINT-SPACE PLANNING

Finding optimal joint-space solutions from among IK options can be challenging. For example, the motion plan for a 7DOF arm illustrated by `example_arm7dof_cart_path_planner_main.cpp` results in 61 Cartesian samples, each of which has between 130 and 285 joint-space IK options. For each Cartesian sample point, a single IK solution must be chosen. However, these cannot be selected independently within each Cartesian sample point. Transitions between successive sample points must result smooth motion–i.e., no large jumps in joint angles. Avoiding large jumps is not as simple as looking one step ahead, as in a "greedy" algorithm, since this can lead to trajectories that result in poor options (large jumps) downstream. Instead, the entire path must be considered in context.

An approach to finding the optimal sequence of joint-space solutions from among a large number of options is dynamic programming. Conceptually, consider the feed-forward graph in Fig 11.1. The example network shown consists of 6 "layers" (columns), each comprised of some number of "nodes" (circles within each column). The network is "feedforward" in the following sense. The "m'th" node in layer "l", denoted as node $n_{l,m}$, may have a "link" from to the n'th node in layer "l+1" (node $n_{(l+1),n}$). However, node $n_{l,m}$ has no link to any node in any layer other than layer l+1. (e.g. links are never directed backwards, nor do they skip over subsequent layers).

A path within this network starts at some node in layer L1 and concludes at some node in layer L6, visiting a total of 6 nodes (including start and finish) in the path. Any such path has an associated cost "C" that is the sum of state costs and transition costs. A state cost is a cost associated with passing through a given node ($c_{l,m}$ for node m in layer l). A transition cost is the cost associated with following a link from node $n_{l,m}$ to node $n_{(l+1),n}$, which may be referred to as transition cost $c_{l,m,n}$ (e.g. as labelled in Fig 11.1). The path cost "C" is a scalar consisting of the sum of all state and transition costs (6 state costs and 5 transition costs, in this example). Thus a candidate path can be scored with a penalty function of path cost. The optimization objective of the proposed problem is to find the path from L1 to L6 that has the minimum path cost.

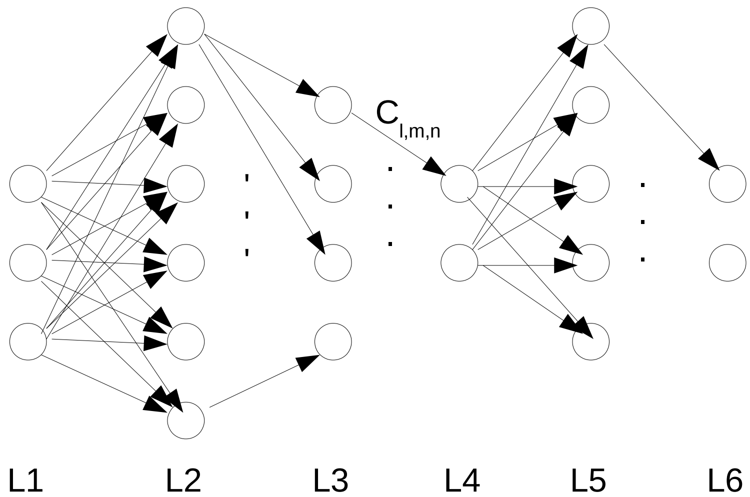


FIGURE 11.1: Conceptualization of a feedforward-network dynamic-programming problem

With respect to our joint-space planning problem, we can interpret each layer as a 6DOF task-space pose sampled along a specified Cartesian path. The objective is to move the tool flange forward along this path, e.g. performing a cutting operation, and we can label (e.g.) 6 points along this path sequentially. It would not make sense to skip over any of these points nor to ever go backwards.

We require that our end-effector visit each of the Cartesian poses. However, computing inverse kinematics, we find that there are multiple joint-space solutions that achieve pose "l." We may refer to the m'th joint-space solution at the l'th Cartesian pose to be node $n_{l,m}$. (Depending on the chosen sampling resolution, we may consider hundreds of IK alternatives at each task-space sample or "layer."). Each node within our network describes a unique joint-space pose of the robot. A sequence of nodes, chosen 1 from each layer consecutively, describes a (discretized) joint-space trajectory that is guaranteed to pass through the sequential Cartesian-space samples along the task-space trajectory.

The state cost for a given node in this context may be scored by multiple criteria. For example, a robot pose that results in a self collision or a collision with the environment may be assigned a cost of infinity (or, more simply, removed from the network altogether). A node (pose) that is a near collision may have a high state cost. Nodes that are near singularities might also be penalized. Other pose-dependent criteria might also be considered. Transition costs should penalize large moves. For example, a path that included moving the base joint by π moving from layer l to layer l+1 would result in high accelerations of high-inertia joints, which would be slow, disruptive, and presumably would result in wild tool-flange gyrations while moving only a short distance along the Cartesian path. Similarly, a sudden wrist flip from layer l to layer l+1 would be unacceptable, although the IK solutions may be correct at both of these layers. Thus, large changes in joint angles from layer to layer should be penalized.

Different optimal path solutions will result from different strategies of assigning state and transition penalties. The designer must choose how to assign costs to joint-space poses, how to assign transition costs, the number and locations of the Cartesian-space sample points (layers), and how many IK solutions (nodes) to consider at each sample point. Having cast the planning problem as a feedforward graph, we can apply graph-solving techniques to our abstracted problem, yielding a sequence of joint-space poses that advance the robot through the chosen Cartesian-space poses.

For this type of graph, dynamic programming can be applied effectively. The process is as follows. Working backwards from the final layer, each node will be assigned a "cost to go". For the final layer, the cost to go is simply the state cost associated with each final-layer node.

Backing up to layer L-1 (layer L5, in our example), each node in this layer is assigned a cost-to-go. For our example, for node 1 in layer L5, $n_{5,1}$, there are only two options: to transition to node $n_{6,1}$ at a cost of $c_{5,1} + c_{5,1,1} + c_{6,1}$, or transition to node $n_{6,2}$ at a cost of $c_{5,2} + c_{5,1,2} + c_{6,2}$. The cost-to-go for node $n_{5,1}$ will be assigned the minimum of these two options, which we may label as $C_{5,1}$. The cost-to-go for node $n_{5,2}$ is computed similarly, resulting in an assigned value $C_{5,2}$.

Continuing to work backwards through the network, when computing cost-to-go for nodes in layer l, we may assume that the cost-to-go has already been computed for all nodes in layer l+1. To compute the cost-to-go for node m in layer l, $C_{l,m}$, consider each node n in layer l+1 and compute the corresponding cost $C_{l,m,n} = c_{l,m} + C_{(l+1),n}$. Find the minimum cost (over all options $n_{(l+1),n}$ in layer l+1) among these options and assign it to $C_{l,m}$. In this process, a cost-to-go is computed for every node in the network, and computing these costs is comparably efficient at every layer.

Given cost-to-go assignments for every node, the optimal path through the network can be found by following the steepest gradient of costs $C_{l,m}$ through the network.

A library that performs this optimal-path computation is `joint_space_planner`. This planner assumes all state costs are merely zero (with the presumption that dangerous poses have already been removed from the network). Transition costs are assigned to be a weighted sum of squares of delta joint angles corresponding to a transition. One must provide an entire network and a vector of weights (associated with joints) to the planner constructor, and the planner will return the optimal path through the network. The network passed to the solver is represented as:

```
vector<vector<Eigen::VectorXd> > &path_options
```

The `path_options` argument is a (std) vector of (std) vectors of (Eigen) vectors. The outermost vector contains L elements corresponding to L layers in the network. Each layer within this vector contains a vector of nodes (IK options at this point along the trajectory). The number of nodes within a layer may vary from layer to layer. Each node within a layer is an N-jointed IK solution, expressed as an Eigen-type vector. The network solver is written to accommodate arbitrary dimensions in terms of number of layers, number of nodes in any layer, and number of dimensions of each node. However, care should be taken to avoid excessive numbers of layers and numbers of nodes per layer, as the planning time can become slow.

In the example `example_arm7dof_cart_path_planner_main.cpp`, a 7DOF robot is considered moving along a Cartesian path sampled every 5cm, resulting in 61 layers (including start and end poses). Each layer has between 130 and 285 IK joint-space options. On one core of a 2GHz i7 Intel processor, the planning process requires about 8 seconds, which includes approximately 3 seconds to compute approximately 13,000 IK solutions and about 5 seconds to find the min-cost path through the resulting network.

The result of the example plan is the file `arm7dof_poses.path`, which lists the 7-DOF joint angles recommended for each of the 61 steps along the Cartesian path. The next step would be to execute this plan on the robot.

11.3 TRAJECTORY MESSAGES FOR ROBOT ARMS

In ROS, the conventional means to execute a trajectory plan is to populate a "trajectory" message and send this message to a trajectory-execution action server.

As described within a URDF file, a robot model consists of a collection (a tree) of robot links, connected pair-wise via joints. Each joint allows a single degree of freedom of motion– either a rotation or a translation. Either type of motion may be referred to generically as a "displacement." A low-level servo controller, as discussed in Chapter 9, exerts a torque or force (generically referred to as an "effort") to attempt to achieve a desired "state." The desired state is comprised of a desired position, possibly augmented by desired velocity, possibly further augmented by desired acceleration. The controller compares the desired state to the actual (measured) state to derive a control effort, which is to be imposed by the respective actuator.

If the desired state is too far from the actual state, the controller will try to exert unreasonable efforts, resulting in effort saturation and leading to poor following of or convergence to desired states. The resulting motion can be unpredictable and dangerous.

In order for the robot to achieve a desirable motion, the motion commands should be evaluated in advance to make sure they are achievable within the constraints of:

- The robot should not hit objects in its environment
- The path in joint space should conform to the constraints of min and max range of motion of the joints
- The joint velocities should remain within the velocity limits of the respective actuators
- The required joint efforts should remain within the effort limits of the actuators

The motion planner described in Section 11.1 partially addresses these concerns (assuming IK solutions resulting in collisions are removed from the set of pose options). The result of the motion planner, however, is a "path", not a "trajectory." The path defines a sequence of poses to be realized. To convert a path to a trajectory, one must augment each pose with an arrival time (time from start of motion). Arrival times can be computed such that velocity and effort constraints are satisfied.

One condition to avoid is a "step" command to any joint. If a joint is commanded to move instantaneously from A to B, the motion will be physically impossible to achieve, and the resulting behavior will likely be undesirable.

A necessary (but not sufficient) requirement of a safe robot command is that the commands should be updated frequently (e.g. 100Hz or faster is typical), and the commands should form an approximation of a smooth, continuous stream for all joints.

Since this is a common requirement, ROS includes a message type for this style of command: `trajectory_msgs/JointTrajectory`.

Invoking:

```
rosmsg show trajectory_msgs/JointTrajectory
```

shows that this message is comprised of the following fields:

```
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
```

In the header, the `frame_id` is not meaningful, since the commands are in joint space (desired state of each joint).

The vector of strings `joint_names` should be populated with text names assigned to the joints (consistent with naming in the URDF file). For a serial-chain robot, joints are conventionally known by integers, starting from joint 1, the joint closest to ground (most "proximal" joint), and progressing sequentially out to the most distal joint. However, robots with multiple arms and/or legs are not so easily described, and thus names are introduced.

In specifying a vector of desired joint displacements, one must associate the joint commands with the corresponding joint names. Generically, there is no requirement to specify the joint commands in any specific order. Further, there is generally no requirement that one must specify all joint commands on every iteration. For example, one might command a neck rotation only in one instance, then follow that by a separate command to a subset of joints of the right arm, etc. However, some packages require that all joint states be specified in every command (whether or not it is desired to move all joints). Other packages receiving trajectory messages might implicitly depend on specifying joint commands in a specific, fixed order, ignoring the `joint_names` field (although this is not preferred).

The bulk of the trajectory message is a vector of type `trajectory_msgs/JointTrajectoryPoint`. This type contains 4 variable-length vectors and a "duration." A trajectory command can use as few as one of these fields and as many as all four. One common minimal usage is to specify only the joint displacements in the "positions" vector. This can be adequate, particularly for low-speed motions controlled by joint position feedback. Alternatively, the trajectory command might communicate with a velocity controller, e.g. for speed control of wheels. A more sophisticated motion plan communicating with a more sophisticated joint controller would include multiple fields (specifying both positions and velocities is common).

To command coordinated motion of all 7 joints of a 7-DOF arm, for example, one would populate (at least) the "positions" vector for each of N "points" to visit, starting from the current pose and ending at some desired pose. Preferably, these points would be relatively close together in space (i.e. with relatively small changes in any one joint displacement command between sequential "point" specifications). Alternatively, coarser trajectories may be communicated to a trajectory interpolator, which breaks up motion between coarse subgoals into streams of fine motion commands.

It is desirable that each point include specification of joint velocities that are consistent with the specified displacements and arrival times (although it is legal to specify a trajectory without specifying the joint velocities). Alternatively, generation of consistent velocity commands might be left up to a lower-level trajectory interpolator node.

Each joint-space point to be visited must specify a `time_from_start`. These time specifications must be monotonically increasing for sequential points. Further, these time specifications should be consistent with the velocity specifications. It is the user's responsibility to evaluate that the specified joint displacements, joint velocities and point arrival times are all self consistent and achievable within the robot's limitations.

The `time_from_start` value, specified for each joint-space point to be visited, distinguishes a "path" from a "trajectory." If one were to specify only the sequence of poses to be realized, this would comprise a path description. By augmenting space (path) information with time, the result is a trajectory (in joint space).

While a trajectory message may be fine-grained and lengthy, it is more common (and more practical) to send messages that consist of a sequence of subgoals (with adequate resolution). Fine-grained commands can be generated by interpolation via an action server. An illustrative example is provided in the package `example_trajectory`.

In this package, an action message is defined: `TrajAction.action`. The "goal" field of this action messages contains a `trajectory_msgs::JointTrajectory` trajectory. This action message is used by a trajectory client to send goals to a trajectory action server.

Two illustrative nodes are the `example_trajectory_action_client` and `example_trajectory_action_server`. The client computes a desired trajectory–in this case, consisting of samples of a sinusoidal motion of "joint1." (This could easily be extended to N joints, but this is sufficient for testing on the `minimal_robot`.) The samples are deliberately taken at irregular and fairly coarse time intervals. An example output is shown in Fig 11.2. The values of position command

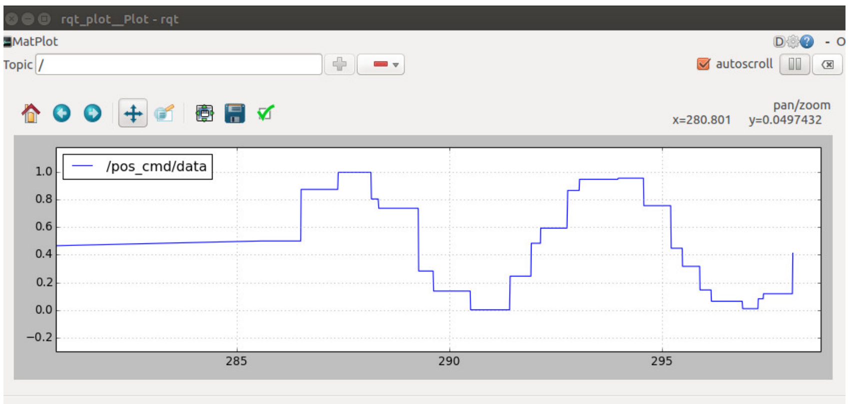


FIGURE 11.2: Coarse trajectory published at irregularly sampled points of a sine wave

(radians) from the above figure Fig 11.2 show that the originating sinusoidal function is sampled coarsely and irregularly. This was done to illustrate the generality of the trajectory message, and that it need not be fine grained.

The joint-command samples are packaged into a trajectory message along with associated arrival times, and this message is transmitted to the trajectory action server within the action goal.

The `example_trajectory_action_server` receives the goal message and interpolates linearly between specified points, resulting in the profile shown in Fig 11.3. The resulting profile is piecewise linear, but sufficiently smooth; the `minimal_robot` can follow this command reasonably well, yielding a smooth motion.

The example is run with:

```
roslaunch minimal_robot_description minimal_robot.launch
```

which brings up the one-DOF robot and its controller. The trajectory interpolation action server is started with:

```
rosrun example_trajectory example_trajectory_action_server
```

and the corresponding trajectory client is started with:

```
rosrun example_trajectory example_trajectory_action_client
```

This client generates a coarse trajectory and sends it to the action server within a goal

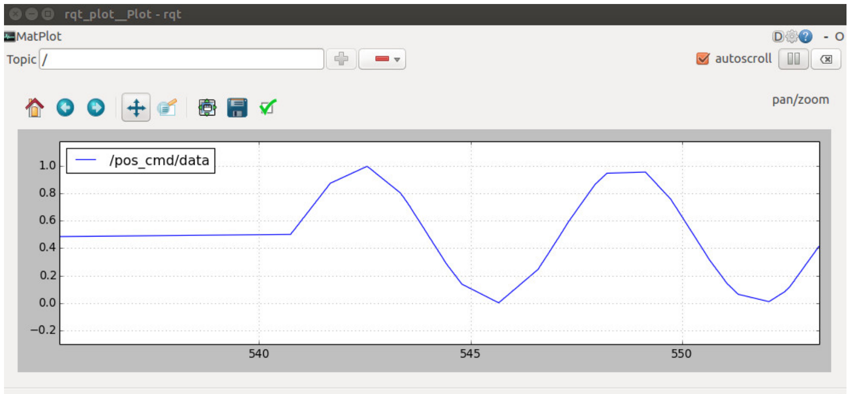


FIGURE 11.3: Action-server linear interpolation of a coarse trajectory

message. In turn, the action server interpolates this trajectory and sends a fast, smooth stream of commands to the robot.

For the example trajectory action server, the velocities specified by the client in the goal message are ignored. In the one-DOF robot case, the minimal joint controller does not accept velocity feedforward commands. More generally, one could do better with a servo controller that accepts both position and velocity-feedforward commands. In this case, including consistent velocity commands sent to the servo controller would improve tracking performance. These velocity commands could be generated on the fly by the trajectory interpolator, or could be included in a longer, fine-grained trajectory message.

The example trajectory interpolator is piecewise linear. It would also be desirable to have a smoother interpolator, e.g. cubic splines. (See e.g. `http://sdk.rethinkrobotics.com/wiki/Joint_Trajectory_Action_Server` for a joint trajectory action server for the Baxter robot, written in Python, which uses cubic spline interpolation).

It is a design decision as to how intelligent the trajectory action server should be. Optimizing a trajectory with considerations of speed, precision and collision avoidance is a computationally difficult problem. Further, the optimization criteria could change from one instance to another. The trajectory specification–even if it is coarse– must still take into consideration issues of collisions and speed and effort constraints. Rather than perform this optimization twice (for construction of a viable trajectory, then further smoothing and optimization of that trajectory by the action server), one could perform trajectory optimization as part of the planning process that generates a trajectory specification. This could yield a dense sampling of points in the trajectory specification, along with compatible velocities, accelerations and gravity-load compensation. If this approach is taken, then linear interpolation of the dense trajectory commands should be adequate in general–although the trajectory action server should also pass along the corresponding velocity and effort values contained within these otpimized trajectories.

Trying to make the trajectory action server more intelligent than mere linear interpolation is thus not warranted–and may, in fact, interfere with optimization of pre-computed trajectories. Thus, this simple example may be considered adequate in general.

This discussion of trajectory messages applies directly to ROS-Industrial (see `http://rosindustrial.org`). To bridge ROS to existing industrial robots, one can write the equivalent of the `example_trajectory_action_server` in the native language of the target robot (and run it on the native robot controller). This (non-ROS) program also requires custom communications to receive packets containing the equivalent of trajectory messages.

A complementary ROS node would receive trajectory messages via a ROS topic or ROS goal message, translate them into the format expected by the robot's communication program, and transmit the corresponding packets to the robot controller for interpolation and execution.

Correspondingly, the robot controller would also run a program that samples and transmit the robot's joint state (at least the joint angles). A ROS node would receive these values in some custom format, translate them into ROS `sensor_msgs::JointState` messages and publish these on the topic `joint_states` for use by other ROS nodes (including Rviz). Using this approach, the ROS-I consortium has retrofit ROS interfaces to a growing number of industrial robots.

11.4 TRAJECTORY INTERPOLATION ACTION SERVERS FOR A 7-DOF ARM

The package `arm7dof_traj_as` contains a helpful library, `arm7dof_trajectory_streamer`, and an action server, `arm7dof_action_server`. The action server in this package extends the minimal-robot example to 7DOF. This action server responds to goal messages that contain joint-space trajectories. The incoming trajectories may be coarse, since they are interpolated at 50Hz (a parameter defined in the header file `arm7dof_trajectory_streamer.h`).

An example client program, `arm7dof_traj_action_client_prompter`, illustrates use of the trajectory action server. This example client first sends the robot to a hard-coded pose, then it prompts the user to enter joint numbers and joint values, which it then commands to the robot via the trajectory action server. To run this example, first start up Gazebo:

```
roslaunch gazebo_ros empty_world.launch
```

Next, bring up the 7-DOF robot arm with its position controllers, using:

```
roslaunch arm7dof_model arm7dof_w_pos_controller.launch
```

Note that these two steps are only required for robot simulation. When interacting with a real robot with a ROS interface, the actual robot dynamics takes the place of Gazebo's physics engine, and the real controllers take the place of the ROS controllers. Typically, the physical robot would host the roscore. The robot would expose its topics for publishing robot state and receiving joint commands.

Next:

```
rosrun arm7dof_traj_as arm7dof_traj_as
```

brings up the trajectory action server, which provides a trajectory interface between low-level joint-angle commands and higher-level trajectory plans. This node should be run whether in simulation or physical operation.

The example interactive trajectory client can be started with:

```
rosrun arm7dof_traj_as arm7dof_traj_action_client_prompter
```

This will pre-position the robot, then accept commands from the user (one joint at a time). More generally, a trajectory client would be instantiated within a higher-level application that performs trajectory planning, presumably based on sensory information.

see:

```
make plans with: rosrun cartesian_planner example_arm7dof_cart_path_planner_main2
rosrun cartesian_planner example_arm7dof_cart_path_planner_main3
(watch where these results are saved; run playfile from same dir)

roslaunch gazebo_ros empty_world.launch
rosrun arm7dof_traj_as arm7dof_traj_as (publishes to both jnt pos ctl and my ↩
    successive-loop ctlr
roslaunch arm7dof_model arm7dof_w_vel_controller.launch
rosrun nested_loop_control inner_vel_loop

move the robot back and forth with:
rosrun arm7dof_traj_as  arm7dof_playfile_path  arm7dof_fwd.path
rosrun arm7dof_traj_as  arm7dof_playfile_path  arm7dof_rvrs.path
(don't know why core dump)

can run: rosrun tf tf_echo world link8 to confirm motion of end effector
```

```
(placeholder--from package README)
# baxter_trajectory_streamer
This package contains a library of useful utilities, baxter_trajectory_streamer.cpp, for controlling Baxter a
Also, two action servers: left_arm_as and rt_arm_as, which accept trajectory goal messages, using the action
message defined in this package.  The interpolators accept trajectories and interpolate linearly between succ
joint-space points.

An example action client, pre_pose (from traj_action_client_pre_pose.cpp), uses the baxter_trajectory_streame
and sends trajectory goals to the left and right arm action servers.  The example trajectories start from the
current arm poses and go to hard-coded goal poses.

## Example usage
Start the robot or the simulator, e.g.:
`roslaunch baxter_gazebo baxter_world.launch`
Enable the robot:
`rosrun baxter_tools enable_robot.py -e`
Start the trajectory-interpolation action servers:
`rosrun baxter_trajectory_streamer rt_arm_as`
`rosrun baxter_trajectory_streamer left_arm_as`
These action servers should be run for all Baxter code examples provided here.  For an example, run the pre-p
which commands both arms to a hard-coded initial pose (mirrored left and right arms):
`rosrun baxter_trajectory_streamer pre_pose`
```

CHAPTER 12

The Baxter Simulator

CONTENTS

In the context of learning control of robot arms, the “Baxter” robot simulator for Gazebo is highly useful. This simulator is a useful model of the “Baxter” robot by ReThink Robotics, Inc. This simulator, and the physical Baxter robot, offer a ROS interface. Programs written for the simulator can be ported quickly to the corresponding physical robot.

The Baxter simulator can be found at: `http://sdk.rethinkrobotics.com/wiki/Baxter_Simulator`. It can be downloaded and installed using the example installation scripts supporting the associated repository of example code (see `https://github.com/cwru-robotics/cwru_scripts/tree/master/mobile_robotics`).

To start the simulator, environment variables are first set via a script, “baxter.sh”, that resides in the ROS workspace (defined as `ros_ws` in the present context). To set these variables, move to the ROS workspace (roscd) and run:

```
./baxter.sh sim
```

The “sim” argument tells this script that the simulator will be used instead of a real robot.

Next, start up the simulator (from this same terminal) with:

```
roslaunch baxter_gazebo baxter_world.launch
```

This process launches Gazebo with a model of the Baxter robot. The start-up process can be slow (expect about 45 sec). The simulator is ready once the launch window displays “Gravity compensation was turned off.” The Gazebo display will appear as in Fig 12.1.

With the simulator (or physical robot) running, example programs can be run, to visualize how Baxter can move. Before commanding motion, however, the joint actuators must be “enabled,” which is done by issuing:

```
rosrun baxter_tools enable_robot.py -e
```

which will result in the response “Robot Enabled.”

Some example motions can be invoked with provided demos, including:

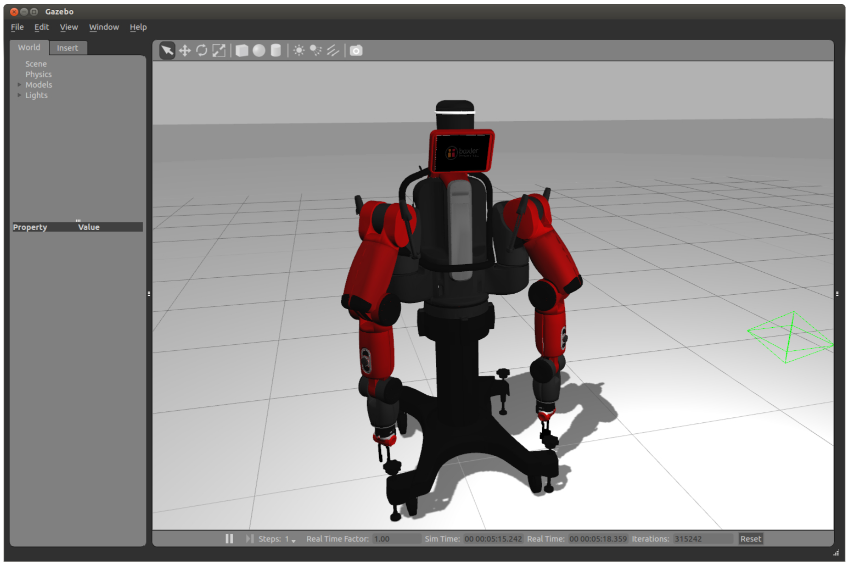


FIGURE 12.1: Gazebo view of the Baxter simulator in an empty world

```
rosrun baxter_tools tuck_arms.py -t
```

Which moves the robot's arms into a pose convenient for shipping, as shown in Fig 12.2. The complementary command:

```
rosrun baxter_tools tuck_arms.py -u
```

will untuck the arms. Another interesting example is:

```
rosrun baxter_examples joint_velocity_wobbler.py
```

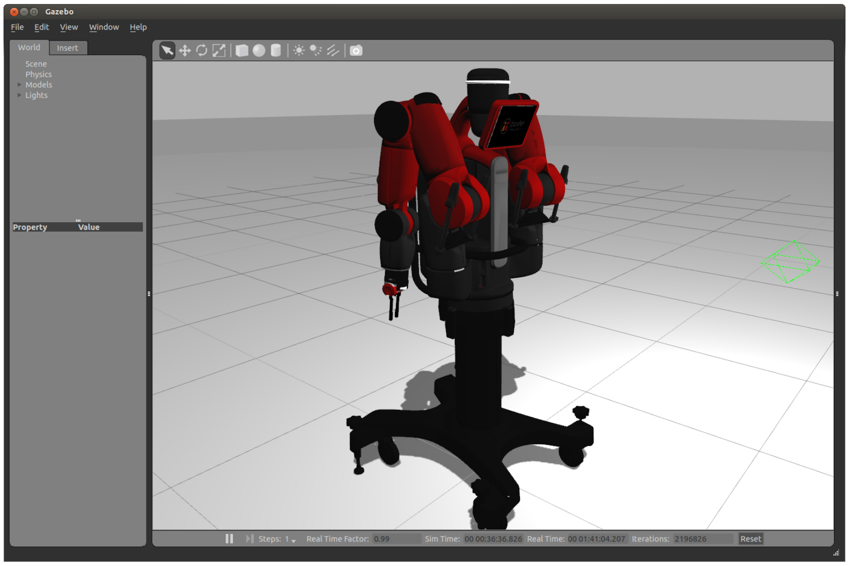


FIGURE 12.2: Gazebo view of the Baxter in tuck pose

12.1 BAXTER JOINTS AND TOPICS

The Baxter robot has 15 servoed degrees of freedom, including 7 joints of the right arm, 7 joints of the left arm and a pan (left-right swivel) motion of the neck. The head (display) can also nod, although this is only a binary command (tilt of the neck is not servo controlled). The Baxter robot and simulator include three cameras: one on the display (head) and one each on the wrists. Other sensors include: sonar sensors around the crown, an short-range infra-red distance sensor in each tool flange, joint-angle sensors on each servoable joint, and joint torque sensors.

Launching rviz, one can visualize more information about the robot. Figure 12.4 shows the robot model with some frames displayed. The torso frame is a useful reference frame. Its z axis points up, x is forward, and y is to the robot's left.

Frames corresponding to arm joints are also displayed for the robot's right arm. For each of these 7 frames, the blue axis corresponds to a joint rotation axis. (For forearm rotation, the blue axis is not visible, as it is buried within the forearm shell in this view). It can be seen that the 7 arm joints are organized has having a "twist" of 90 deg for each sequential pair of joints (roll-pitch-roll-pitch-roll-pitch-roll).

Mathematically, to control six degrees of freedom of a desired gripper pose, one typically needs at least 6 independent joint degrees of freedom. However, once joint limits are imposed, it is typically quite difficult to satisfy all 6 constraints of a desired pose with 6 controllable joints. Having a seventh joint (analogous to a human arm) dramatically improves manipulability. At the same time, this presents additional challenges for inverse kinematics, given that the arm is "kinematically redundant." Increasingly, industrial robots are offering 7 controllable joints, and thus the issue of addressing kinematic redundancy is of increasing importance.

An additional challenge for Baxter kinematics is that this robot does not have "spherical" wrists. The term "spherical wrist" applies to robots for which the final 3 joint axes all intersect at a point. This kinematic property simplifies the mathematics of solving inverse kinematics. For the Baxter robot, the last two axes (blue axes of the last two frames displayed) do intersect–and we may refer to this intersection point as the "wrist point." However, the forearm rotation axis (not shown) does not intersect this point; it misses intersecting the wrist point by a small offset. Since the offset is small, it can be a useful approximation to treat the wrist as spherical, which helps in the computation of fast, approximate inverse-kinematic solutions.

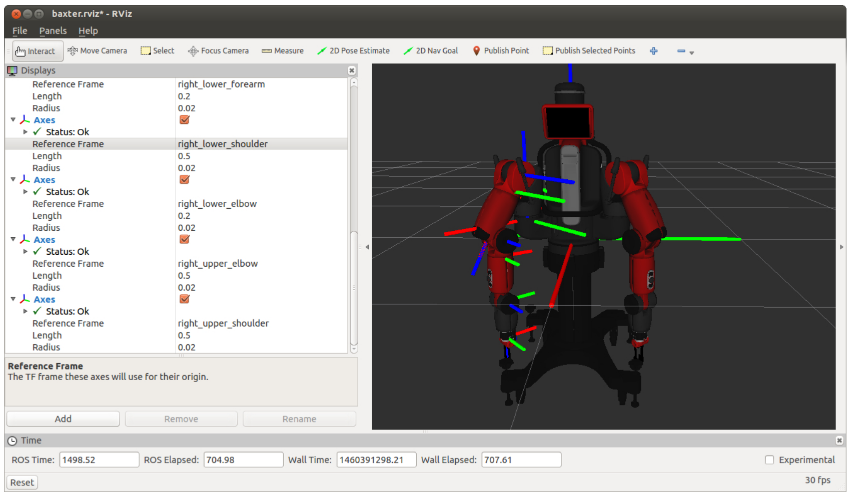


FIGURE 12.3: Rviz view of the Baxter simulator model illustrating right-arm frames

With the Baxter simulator (or physical Baxter) running, executing `rostopic list` reveals that approximately 300 topics are being used. There are nearly 40 topics under "cameras" (Baxter has 3 built-in cameras), several "gazebo" topics, nearly 250 "robot" topics, `tf`, `tf_static` and two "laserscan" sensor topics. One of the most useful topics is the `/robot/joint_states` topic, which is described further below. Running: `rosnode list` shows that a `robot_state_publisher` is running, which is necessary for Rviz to display the robot model with correctly-positioned joints.

Invoking `rosparam list`, it can be seen that a `robot_description` has been loaded onto the parameter server (performed as part of the baxter launch process).

The topic `/robot/joint_states` can be examined by running:

```
rostopic info /robot/joint_states
```

which shows that this topic carries messages of type `sensor_msgs/JointState`, which is a standard ROS means of publishing robot joint states. The JointState message type can be examined by running:

```
rosmsg show sensor_msgs/JointState
```

which shows that this message is comprised of the following fields:

```
std_msgs/Header header
uint32 seq
time stamp
string frame_id
string[] name
float64[] position
```

```
float64[] velocity
float64[] effort
```

Running:

```
rostopic hz /robot/joint_states
```

shows that messages on this topic are being updated at 50Hz. To get a glimpse of the data being published on this topic, run:

```
rostopic echo /robot/joint_states
```

A screenfull of this output is shown in Fig **??**. This output shows position, velocity and

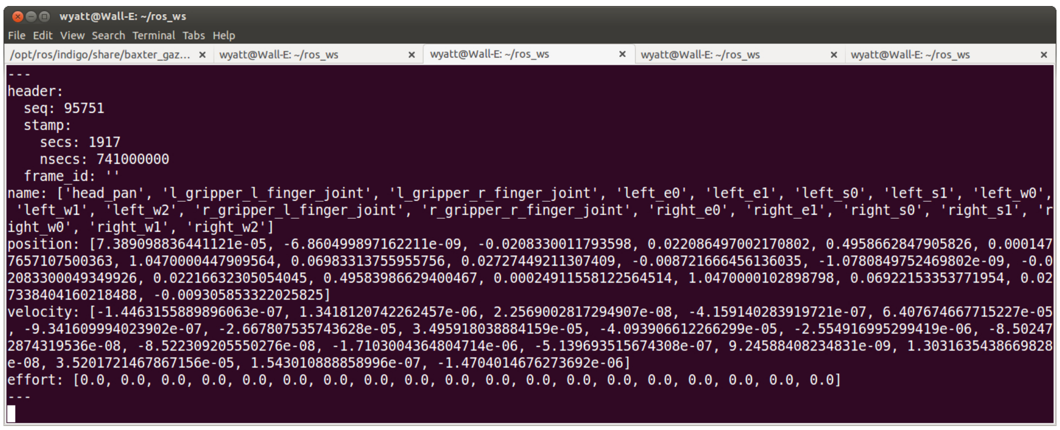


FIGURE 12.4: rostopic echo of `robot/joint_states` for Baxter

"effort" for each of 19 named joints. These joints include the 14 arm joints, the head "pan" joint and 4 gripper-finger joints. In the simulation, all of the reported efforts are zero. However, for the physical robot, effort values for the arm joints are measured and published.

The joint states are reported in the same order as this listing of joint names. The order used by Baxter is somewhat odd; conventionally, joints are reported in order of the chain they describe, from torso to gripper. ROS, however, is generally indifferent to the order of joints, either for reporting states or accepting commands; the order used in the "name" vector is the order ROS will use, and this may change from iteration to iteration.

To clarify Baxter's joint ordering and naming, the joints defined sequentially from torso to wrist for right arm are:

```
right_s0, right_s1, right_e0, right_e1, right_w0, right_w1, right_w2
```

For the left arm, the naming sequence is the same, but substitute "left" for "right." All of position[], velocity[] and effort[] are reported in the order listed by order of names (as seen from the rostopic echo).

The topic `/tf` publishes a large number of link-to-link transforms. All link-to-link relationships described by a connecting joint are updated rapidly. These can be viewed by running `rostopic echo tf` (although this means of viewing is inconvenient, since the display scrolls too quickly). To view a specific relationship, it is more convenient to use `tf_echo`. For example, to display the pose of the right hand (actually, the tool flange frame of the right arm) with respect to the robot's torso frame, run:

```
rosrun tf tf_echo torso right_hand
```

The position and orientation of the `right_hand` frame with respect to the `torso` frame will then be printed out, updated once per second.

The camera topics and laserscan topics are useful for perceptual processing. If more sensors are added to Baxter, the sensor values should be published on topics as well to be made available for interpretation.

12.2 COMMANDING BAXTER JOINTS

The Baxter simulator (and actual Baxter robot) subscribes to the topics `/robot/limb/right/joint_command` and `/robot/limb/left/joint_command` to accept joint position commands for the right and left arms, respectively. These topics carry messages of type `baxter_core_msgs/JointCommand`. We can examine this message type with:

```
rosmsg show baxter_core_msgs/JointCommand
```

```
int32 POSITION_MODE=1
int32 VELOCITY_MODE=2
int32 TORQUE_MODE=3
int32 RAW_POSITION_MODE=4
int32 mode
float64[] command
string[] names
```

The code snippet below illustrates how to command joint angles to joints of Baxter's right arm in position-control mode:

```
//Here is an instantiation of the proper message type:
baxter_core_msgs::JointCommand right_cmd;
//Assuming we have desired joint angles in the array vector qvec[],
//we can fill the command message with:
right_cmd.mode = 1; // position-command mode

//Define a right-arm publisher as:
joint_cmd_pub_right =
nh.advertise<baxter_core_msgs::JointCommand>("/robot/limb/right/joint_command", 1);

// Define the ordering of joints to be commanded as follows:
// define the joint angles 0-6 to be right arm, from shoulder to wrist;
// we only need to do this part once, and we can subsequently re-use this message,
// changing only the position-command data
right_cmd.names.push_back("right_s0");
right_cmd.names.push_back("right_s1");
right_cmd.names.push_back("right_e0");
right_cmd.names.push_back("right_e1");
right_cmd.names.push_back("right_w0");
right_cmd.names.push_back("right_w1");
right_cmd.names.push_back("right_w2");
//Note: in the above, we have created room for 7 joint names. The push_back() command
// should not be repeated, else the list of names will grow with every iteration.
// The joint-command object can be retained, and this ordering of joint names can be
// persistent, so this step can be treated as an initialization. Within a control loop
// only the desired joint values would need to be changed within this command message.

//Assuming a pose of interest is in qvec[], we can specify the right-arm joint-angle
//commands (in radians) with:
for (int i = 0; i < 7; i++) {
right_cmd.command[i] = qvec[i];
}
//and publish this command as:
joint_cmd_pub_right.publish(right_cmd);
```

As we saw with the minimal robot, it is useful to have an action server that can interpolate joint commands to smoothly execute trajectories–as described in a trajectory message. ROS defines a message type for this purpose, `trajectory_msgs/JointTrajectory`.

12.3 RECORD AND PLAYBACK NODES

- get and save jntvals
- baxter record trajectory
- baxter playfile jointspace

```
(placeholder--from package README)
# baxter_playfile_nodes
Handy functions to record and playback Baxter joint states.

`rosrun baxter_playfile_nodes get_and_save_jntvals`
will sample the current joint angles of left and right arms.  Each time the user
enters "1", the current joint angles will be appended to the files "baxter_r_arm_angs.txt"
and "baxter_l_arm_angs.txt".

The node "baxter_recorder" extends this to recording and saving joint trajectories (with
the actual timing of manual arm motions).  This node
samples the right-arm and left-arm joint angles,
and saves them to disk in the files "baxter_r_arm_traj.jsp" (joint-space playfile)
and "baxter_l_arm_traj.jsp". (control-C when done recording).

Trajectories recorded in this fashion can be played back with the playfile reader,
"baxter_playback."

## Example usage
With Baxter (sim or real) running, cd to a trajectory appropriate to save recordings.
Be careful to rename "baxter_r_arm_traj.jsp" and "baxter_l_arm_traj.jsp" to avoid overwriting previous record
When ready to record, start:
`rosrun baxter_playfile_nodes baxter_recorder`
enter "1" at the program prompt, then move the arms in the desired trajectory (path and speed).
When done with recording, ctl-C.  The results will be in "merry_r_arm_traj.jsp" and "merry_l_arm_traj.jsp".

To play back joint-space trajectory files, start up the robot and enable it.  Start up the trajectory
interpolation action servers:
Start the trajectory-interpolation action servers:
`rosrun baxter_trajectory_streamer rt_arm_as`
`rosrun baxter_trajectory_streamer left_arm_as`

In another terminal, cd to the directory containing the desired trajectory filename(s).
Run the playback node, with command-line arguments for the right-arm trajectory and left-arm trajectory.
(If only 1 filename is provided, it will be interpreted and executed as the right-arm file). E.g.:
`rosrun baxter_playfile_nodes
`rosrun baxter_playfile_nodes baxter_playback baxter_r_arm_traj.jsp`
(to move just the right arm).  Or,
`rosrun baxter_playfile_nodes baxter_playback baxter_r_arm_traj.jsp baxter_l_arm_traj.jsp`
to move both arms.
```

12.4 BAXTER KINEMATICS

- tool transform
- forward kinematics
- Jacobian
- inverse kinematics

See package **baxter_fk_ik**. Makes library (of the same name). See, e.g., function

```
int Baxter_IK_solver::ik_solve_approx_wrt_torso(Eigen::Affine3d const& ↩
    desired_flange_pose,std::vector<Vectorq7x1> &q_solns)
```

This function takes a desired hand pose, expressed as an Affine with respect to the torso frame, and returns a vector of sampled arm poses (in joint space) that (approximately) satisfy the desired hand pose. (More precise solutions can be refined from such seed solutions, if desired).

12.5 JOINT-SPACE PLANNER

This package creates a library that performs dynamic programming on a feed-forward graph to minimize a path cost from one or more start options (alternative IK solutions) to one or more goal options (alternative IK solutions). Intermediate "layers" are alternative IK solutions at points sampled along a Cartesian path.

12.6 CARTESIAN MOVES

To refer to yale-gripper hand frame, run:

```
roslaunch baxter_launch_files yale_gripper_xform.launch
```

Can then view frame in rviz and print out data with:

```
rosrun tf tf_echo torso yale_gripper_frame
```

```
bool CartTrajPlanner::cartesian_path_planner(Vectorq7x1 q_start,Eigen::Affine3d ↩
    a_tool_end, std::vector<Eigen::VectorXd> &optimal_path)
```

This function takes a desired hand pose (in Cartesian space) and a starting arm pose (in joint space) and fills in