Con iguring the Baxter shell

The Baxter SDK requires the `baxter.sh` script ile to establish the connections between the Baxter robot and the workstation computer. This connection will depend on how your network is set up. Further details on a network connection to a real Baxter will be discussed in the *Con iguring a real Baxter setup* section.

Baxter Simulator additionally uses the `baxter.sh` file to establish a simulation mode where the ROS environment variables are set up to identify the host workstation computer. The `baxter.sh` script file contains a *special hook* for Baxter Simulator.

1. First, the file must be copied to the `baxter_ws` directory and the file permissions changed to grant execution privileges to all users:
   ```
   $ cp ~/baxter_ws/src/baxter/baxter.sh ~/baxter_ws
   $ chmod +x baxter.sh
   ```
2. Next, open the `baxter.sh` script in your favorite editor and find the `your_ip` parameter (around line 26). Change the `your_ip` value to the IP address of your workstation computer, e.g.:
   ```
   your_ip="192.168.XXX.XXX"
   ```
   If the IP address of your computer is unknown, connect to the network used for Baxter, then use the `ifconfig` command:
   ```
   $ ifconfig
   ```
   The screen results will contain the `inet_addr` field for the IP address of the workstation computer.
3. Alternatively, if you wish to use the hostname, comment out the line for `your_ip` and uncomment the line for `your_hostname`. To use the real hostname of your workstation computer, use the following command:
   ```
   $ hostname
   ```
   Then, add this to the `your_hostname` parameter.
   These parameters will assign either the `ROS_IP` or `ROS_HOSTNAME` environment variable. If both are present, the `ROS_HOSTNAME` variable takes precedence.
4. Also, near line 30, change `ros_version` to `noetic`. Then, save and close the `baxter.sh` script.

Installing MoveIt!

MoveIt! is an important ROS tool for path planning and can be used with Baxter Simulator or a real Baxter. The installation of MoveIt! is described here, while the operation of MoveIt! is detailed later in the chapter, in the *Introducing MoveIt!* section. Instructions for the installation can also be found on the Rethink website at `http://sdk.rethinkrobotics.com/wiki/MoveIt_Tutorial`. *These instructions are for ROS Indigo, but they have been updated for ROS Noetic here.*

The MoveIt! software should be loaded into the source (`src`) directory of the catkin workspace `baxter_ws` created earlier in the chapter in the section *Loading the Baxter software*. The commands are as follows:

```
$ cd ~/baxter_ws/src
$ git clone https://github.com/ros-planning/moveit_robots.git
$ sudo apt-get update
$ sudo apt-get install ros-noetic-moveit
```

Then, new additions to the workspace are incorporated with the `catkin_make` command:

```
$ cd ~/baxter_ws
$ catkin_make
```

To verify that all the Baxter SDK, simulator, and MoveIt packages were downloaded and installed, type the following command:

```
$ ls ~/baxter_ws/src
```

The output should be as follows:

```
baxter          baxter_examples     baxter_simulator    CMakeLists.txt
baxter_common   baxter_interface    baxter_tools        MoveIt!_robots
```

Launching Baxter Simulator in Gazebo

Before launching Baxter Simulator in Gazebo, it is important to check the ROS environment variables. To start up Baxter Simulator, use the following commands to get to your Baxter catkin workspace and run your `baxter.sh` script with the `sim` parameter:

```
$ cd ~/baxter_ws
$ ./baxter.sh sim
```

The command prompt should return with the following tag appended to the beginning of the prompt:

```
[baxter - http://localhost:11311]
```

You are now talking to the simulated Baxter! At this point, check your ROS environment with the following command:

```
$ env | grep ROS
```

Within the output screen text, look for the following result:

```
ROS_MASTER_URI=http://localhost:11311
ROS_IP= <your workstation's IP address>
```

or

```
ROS_HOSTNAME=<your workstation's hostname>
```

The `ROS_HOSTNAME` field need not be present.

If the `ROS_IP` or `ROS_HOSTNAME` environment variables does not match the IP address of your workstation (use `ifconfig` to check), type `exit` to stop communication with the simulated Baxter. Then, edit the `baxter.sh` script to change the `your_ip` variable (near line 26) to the current IP address of your workstation. To continue, repeat the preceding steps for a final check.

If there are issues with Baxter's hardware, software, or network, refer to the general Baxter troubleshooting website at `http://sdk.rethinkrobotics.com/wiki/Troubleshooting`.

The `baxter.sh` script should run without errors and the ROS environment variables should be correct. The next section covers running Baxter Simulator for the first time.

Bringing Baxter Simulator to life

To start Baxter Simulator, go to the `baxter_ws` workspace and run the Baxter shell script with the sim parameter specified:

```
$ cd ~/baxter_ws
$ ./baxter.sh sim
```

Next, call the `roslaunch` command to start the simulation with controllers:

```
$ roslaunch baxter_gazebo baxter_world.launch
```

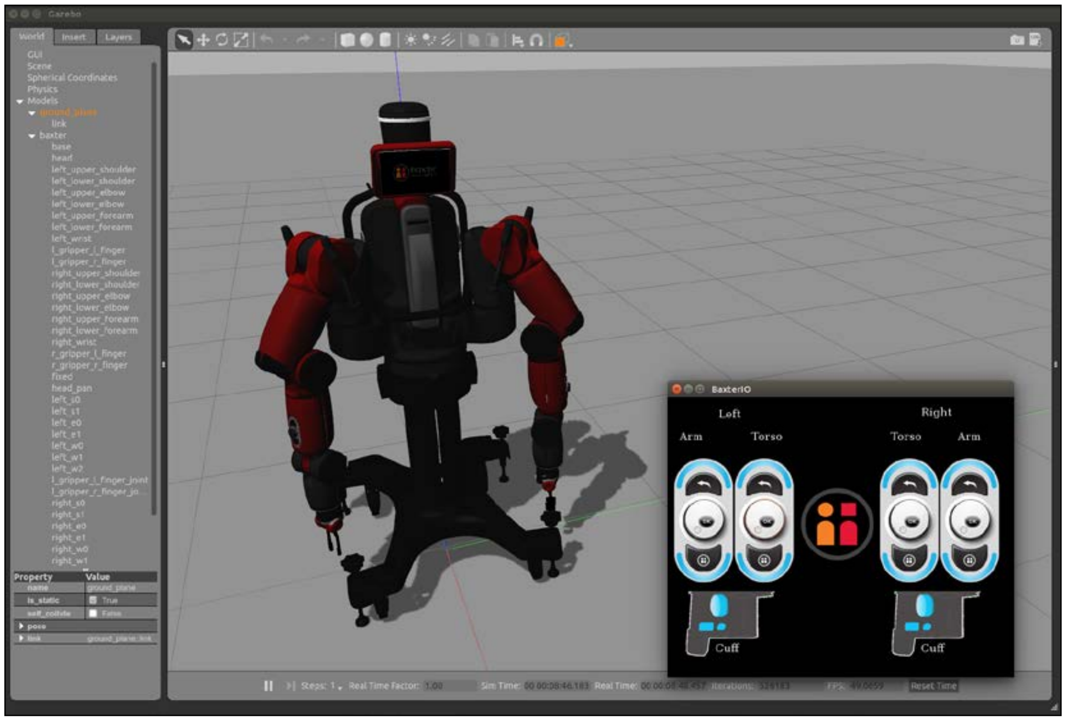
The following lines are some of the results you will see on the screen while Baxter Simulator starts:

```
NODES
  /
    base_to_world (tf2_ros/static_transform_publisher)
    baxter_emulator (baxter_sim_hardware/baxter_emulator)
    baxter_sim_io (baxter_sim_io/baxter_sim_io)
    baxter_sim_kinematics_left (baxter_sim_kinematics/kinematics)
    baxter_sim_kinematics_right (baxter_sim_kinematics/kinematics)
    gazebo (gazebo_ros/gzserver)
    gazebo_gui (gazebo_ros/gzclient)
    robot_state_publisher (robot_state_publisher/robot_state_publisher)
    urdf_spawner (gazebo_ros/spawn_model)
  /robot/
    controller_spawner (controller_manager/controller_manager)
    controller_spawner_stopped (controller_manager/controller_manager)
    left_gripper_controller_spawner_stopped (controller_manager/
controller_manager)
    right_gripper_controller_spawner_stopped (controller_manager/
controller_manager)
```

As the process is completing, look for these lines:

```
[ INFO] [1502315064.794924787, 0.718000000]: Simulator is loaded and
started successfully
[ INFO] [1502315064.905968083, 0.830000000]: Robot is disabled
[ INFO] [1502315064.906014361, 0.830000000]: Gravity compensation was
turned off
```

The following screenshot should appear with Baxter in a disabled state:



Baxter's initial state in Gazebo

If Gazebo and Baxter Simulator fail to appear or there are red error messages in your terminal window, refer to the Gazebo Troubleshooting page provided by Rethink Robotics at `http://sdk.rethinkrobotics.com/wiki/Gazebo_Troubleshooting`.

To remove all Gazebo processes after shutdown, use the command:

```
$ killall gzserver
```

For an introduction to using Gazebo, refer to the *Gazebo* section in *Chapter 2*, *Creating Your First Two-Wheeled ROS Robot (in Simulation)*. In that section, the various Gazebo display panels, menus, and toolbars are explained. Gazebo uses similar cursor/mouse controls to those of rviz, and these mouse/cursor actions are described in the *Introducing rviz*: *Mouse control* section of *Chapter 2*, *Creating Your First Two-Wheeled ROS Robot (in Simulation)*.

In the previous screenshot, the **World** panel on the left shows the **Models** element open to reveal the two models in the environment: `ground_plane` and `baxter`. Under the `baxter` model, all of Baxter's links are listed and you are welcome to select the links to explore the details about each one. The screenshot also shows the smaller display window that contains Baxter's IO. Baxter's four navigators, located one on each side of the rear torso (near the shoulders) and one on each arm, are also shown. The oval-shaped navigators have three push buttons, one of which is a scroll wheel. Baxter's cuff buttons are also shown in this window. There are two buttons and one touch sensor on each cuff.

The terminal window in which the `roslaunch` command was performed will be unable to run additional commands, so a second terminal window should be opened. In this window, go to the `baxter_ws` workspace and run the `baxter.sh` script with the `sim` parameter:

```
$ cd ~/baxter_ws
$ ./baxter.sh sim
```

For each additional terminal window opened, go to the `baxter_ws` workspace and run the `baxter.sh` script with the `sim` parameter.

Baxter (in simulation) is initially in a disabled state. To confirm this, use the `enable_robot` script from the `baxter_tools` package using the following command:

```
$ rosrun baxter_tools enable_robot.py -s
```

The screen should display the following output:

```
ready: False
enabled: False
stopped: False
error: False
estop_button: 0
estop_source: 0
```

To enable Baxter, use the same `enable_robot` script with the `-e` option:

```
$ rosrun baxter_tools enable_robot.py -e
```

The output is similar to the following:

```
[INFO] [1501189929.999603, 141.690000]: Robot Enabled
```

Confirm Baxter is enabled using the following command:

```
$ rosrun baxter_tools enable_robot.py -s
```

The output should be as follows:

```
ready: False
enabled: True
stopped: False
error: False
estop_button: 0
estop_source: 0
```

Always enable Baxter Simulator before attempting to control any of the motors.

At this point, a **cheat sheet** for use with Baxter Simulator is provided for you to use with the example programs that follow. The commands for launching, enabling, and untucking are provided here for your reference:

Baxter Simulator cheat sheet

To launch Baxter Simulator in Gazebo, use the following commands:

```
$ cd ~/baxter_ws
$ ./baxter.sh sim
$ roslaunch baxter_gazebo baxter_world.launch
```

For subsequent terminal windows, use the following commands:

```
$ cd ~/baxter_ws
$ ./baxter.sh sim
```

To enable the robot, use the following command:

```
$ rosrun baxter_tools enable_robot.py -e
```

To enable and set the arms in a known position, use the following command:

```
$ rosrun baxter_tools tuck_arms.py -u
```

With Baxter enabled, the next section describes some of Baxter's example scripts using the head display screen.

Warm-up exercises

Rethink Robotics has provided a collection of example scripts to demonstrate Baxter's interfaces and features. These example programs are contained in the `baxter_examples` package and work primarily with a real Baxter and the SDK. A portion of these example programs also work with Baxter Simulator.

The `baxter_examples` are Python programs that access Baxter's hardware and functionality through the `baxter_interface` package. The `baxter_examples` programs are written to demonstrate how to use Baxter interfaces. The `baxter_interface` package is a repository of Python APIs to use for interacting with the Baxter Research Robot. The repository contains a set of classes that are ROS wrappers to communicate with and control Baxter's hardware and functionality. These Python classes are built on top of the ROS API layer.

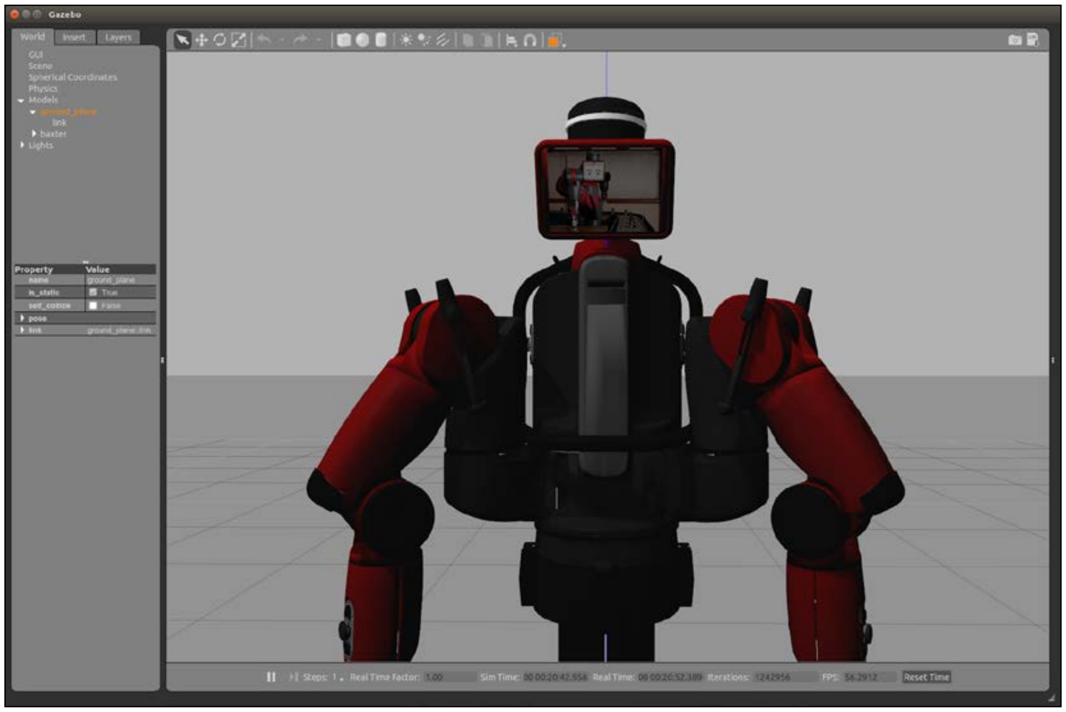
This section and the following sections present SDK example programs that can be used with Baxter Simulator. To find additional information on the SDK example programs implemented in Baxter Simulator, visit the following website:

`http://sdk.rethinkrobotics.com/wiki/API_Reference#tab=Simulator_API`

The first example program will display an image on Baxter's (simulated) head display screen using the following command:

```
$ rosrun baxter_examples xdisplay_image.py --file=`rospack find baxter_
examples`/share/images/baxterworking.png
```

Your screen should look similar to the following screenshot:



Baxter after xdisplay_image.py

The `xdisplay_image.py` program locates the `baxterworking.png` image in the specified location under the `baxter_examples` package. This image data is published as a `sensor_msgs/Image` ROS message. The display image must be a `.png` or `.jpg` file with a display resolution of 1024 x 600 pixels or smaller. Smaller images will appear in the top-left corner of Baxter's display screen.

A second `baxter_examples` program will cause Baxter Simulator to nod Baxter's head up and down, then turn from side to side:

```
$ rosrun baxter_examples head_wobbler.py
```

The simulated Baxter should randomly wobble its head until *Ctrl* + *C* is pressed. The movement demonstrates both the head pan motion (side to side) and head nod motion (up and down) interfaces. This program shows the use of the `baxter_interface Head` class (`head.py`). The `command_nod` function is called first to trigger an up-down motion of the head. It is not possible to command a specific angle for the nod motion. The pan motion is achieved with several calls to the `set_pan` function, with random angles provided as the parameter.

Another `baxter_examples` program also moves Baxter's head through a set of head positions and velocities. The Head Action Client Example demonstrates the use of the Head Action Server. This example is similar to the head wobble just performed, but provides a good example of an action server and client interaction. If you wish to try the Head Action Client Example, access the instructions and explanations at `http://sdk.rethinkrobotics.com/wiki/Head_Action_Client_Example`.

The next section will demonstrate some example programs for Baxter's arms.

Flexing Baxter's arms

The focus of the following sections will be on Baxter's arms. The section on *Bringing Baxter Simulator to life* should be completed before starting these sections. Baxter Simulator should be launched in Gazebo and the robot should be enabled with its arms untucked.

The following example programs use the `baxter_interface Limb` class (`limb.py`) to create instances for each arm. The `joint_names` function is used to return an array of all the joints in the limb.

Commands for the joint control modes are via ROS messages within the `baxter_core_msgs` package. To move the arm, a `JointCommand` message must be published on the `robot/limb/<left/right>/joint_command` topic. Within the `JointCommand` message, a mode field indicates the control mode to the Joint Controller Boards as `POSITION_MODE`, `VELOCITY_MODE`, `TORQUE_MODE`, or `RAW_POSITON_MODE`.

In the following sections, various methods of controlling Baxter's arm movements will be demonstrated. After several example arm programs have been presented, a Python script to command Baxter's arms to move to a home position will be shown.

Untucking Baxter's arms

Before Baxter's arms can be commanded, Baxter must be enabled. This can be accomplished using the `tuck_arms.py` program provided by Rethink using the `untuck` option. During untuck movements, Baxter's collision avoidance is disabled. Collision avoidance for Baxter Simulator is modeled as part of the URDF. Each of Baxter's links is tagged with a collision block that is slightly larger than the visual element. For further details on the URDF, collision blocks, and the visual element, refer to *Chapter 2, Creating Your First Two-Wheeled ROS Robot (in Simulation)*. Typically, when the position of the arms places the collision blocks into contact with each other, the collision model detects the contact and ends the movement to avoid the parts colliding.

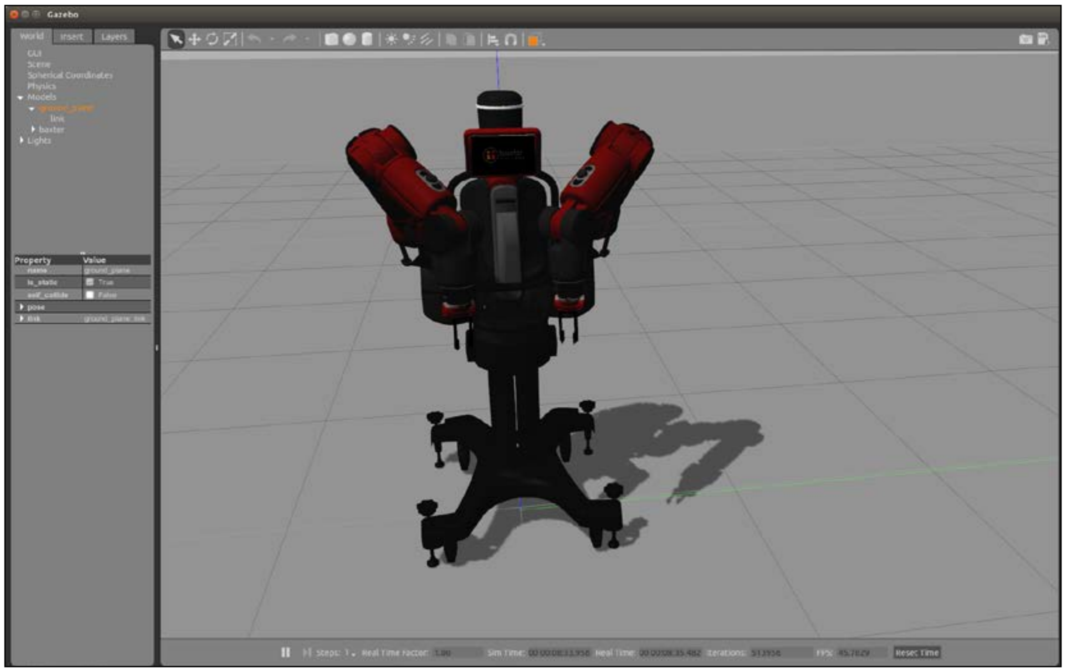
To command Baxter into the untuck position, use the following command:

```
$ rosrun baxter_tools tuck_arms.py -u
```

The output should be similar to the following:

```
[INFO] [1501190044.262606, 0.000000]: Untucking arms
[INFO] [1501190044.375889, 255.938000]: Moving head to neutral position
[INFO] [1501190044.376109, 255.938000]: Untucking: Arms already Untucked;
Moving to neutral position.
[INFO] [1501190045.673587, 257.234000]: Finished tuck
```

The following screenshot shows the simulated Baxter in the untucked position:



Baxter untucked

To explore Baxter's untuck operation further, refer to the Rethink wiki *Tuck Arms Tool* information at `http://sdk.rethinkrobotics.com/wiki/Tuck_Arms_Tool`.

Wobbling arms

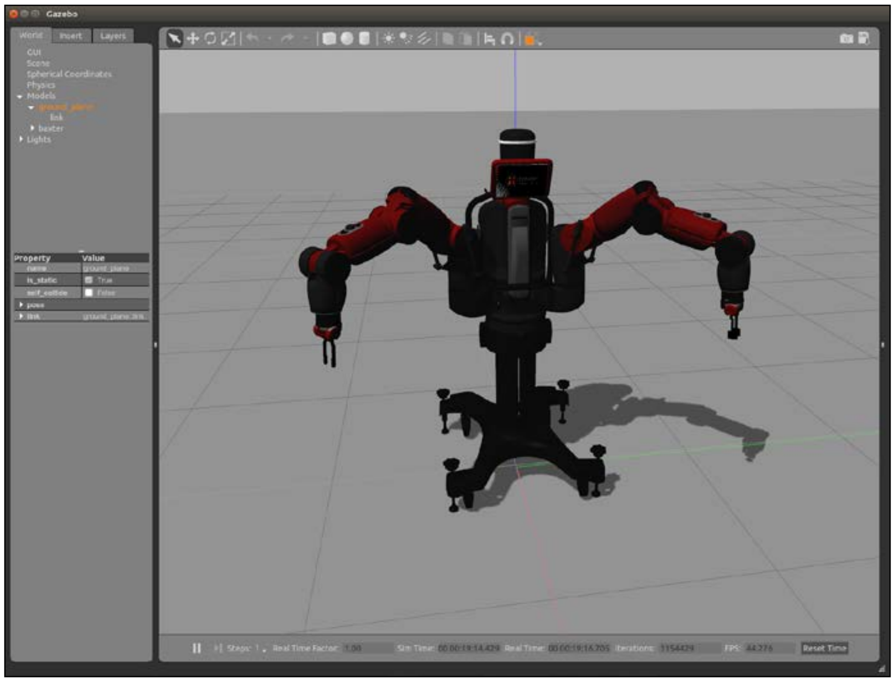
The next example program provides a demonstration of controlling Baxter's arms using joint velocity control. The joint control modes for Baxter's arms were described in the *Baxter's arms* section. In simulation, the joint velocity wobble can be observed by typing the following command:

```
$ rosrun baxter_examples joint_velocity_wobbler.py
```

The output should be as follows:

```
Initializing node...
Getting robot state...
Enabling robot...
[INFO] [1501190177.147845, 388.564000]: Robot Enabled
Moving to neutral pose...
Wobbling. Press Ctrl-C to stop...
```

The program will begin by moving Baxter's arms to a preset neutral starting position. Next, random velocity commands are sent to each arm to create a sinusoidal motion across both limbs. The following screenshot shows Baxter's neutral starting position:



Baxter's neutral position

To explore Baxter's arms' `joint_velocity_wobbler` operation in more detail, refer to the Rethink wiki *Wobbler Example* information at `http://sdk.rethinkrobotics.com/wiki/Wobbler_Example`.

Controlling arms and grippers with a keyboard

Baxter's arms can also be controlled with keyboard keystrokes. Keystrokes are used to control the positions of the joints, with each keyboard key mapped to either increase or decrease the angle of one of Baxter's 14 arm joints. Keys on the right side of the keyboard are mapped to Baxter's left arm and keys on the left side of the keyboard are mapped to Baxter's right arm.

This example demonstrates another of Baxter's arm control modes: joint position control.

To start the keyboard joint position control example, use the following command:

```
$ rosrun baxter_examples joint_position_keyboard.py
```

This should be the output on the screen:

```
Initializing node...
Getting robot state...
Enabling robot...
[INFO] [1501190427.217690, 638.355000]: Robot Enabled
Controlling joints. Press ? for help, Esc to quit.
key bindings:
  Esc: Quit
  ?: Help
```

`/: left: gripper calibrate`	`b: right: gripper calibrate`
`,: left: gripper close`	`c: right: gripper close`
`m: left: gripper open`	`x: right: gripper open`
`y: left_e0 decrease`	`q: right_e0 decrease`
`o: left_e0 increase`	`r: right_e0 increase`
`u: left_e1 decrease`	`w: right_e1 decrease`
`i: left_e1 increase`	`e: right_e1 increase`
`6: left_s0 decrease`	`1: right_s0 decrease`
`9: left_s0 increase`	`4: right_s0 increase`
`7: left_s1 decrease`	`2: right_s1 decrease`
`8: left_s1 increase`	`3: right_s1 increase`
`h: left_w0 decrease`	`a: right_w0 decrease`
`l: left_w0 increase`	`f: right_w0 increase`
`j: left_w1 decrease`	`s: right_w1 decrease`
`k: left_w1 increase`	`d: right_w1 increase`
`n: left_w2 decrease`	`z: right_w2 decrease`
`.: left_w2 increase`	`v: right_w2 increase`

The output has been modified to aid ease of use.

Controlling arms and grippers with a joystick

This example program uses a joystick to control Baxter's arms. The `joint_position_joystick` program uses the ROS drivers from the `joy` package to interface with a generic Linux joystick. Joysticks with a USB interface are supported by the `joy` package. The `joy` package creates a `joy_node` to generate a `sensor_msgs/Joy` message containing the various button-push and joystick-move events.

The first step is to check for the joystick driver package `joy` using the following command:

```
$ rospack find joy
```

If the ROS package is on the computer, the screen should display this:

```
/opt/ros/noetic/share/joy
```

If it is not, then an error message is displayed:

```
[rospack] Error: stack/package joy not found
```

If the `joy` package is not present, install it with the following command:

```
$ sudo apt-get install ros-noetic-joystick-drivers
```

For a PS3 joystick controller, you will need the `ps3joy` package. Instructions can be found at `http://wiki.ros.org/ps3joy/Tutorials/PairingJoystickAndBluetoothDongle`.

Next, type the command to start the `joint_position_joystick` program using one of the joystick types (`xbox`, `logitech`, or `ps3`):

```
$ roslaunch baxter_examples joint_position_joystick.launch
joystick:=<joystick_type>
```

We used the Xbox controller joystick in our example; the output is as follows:

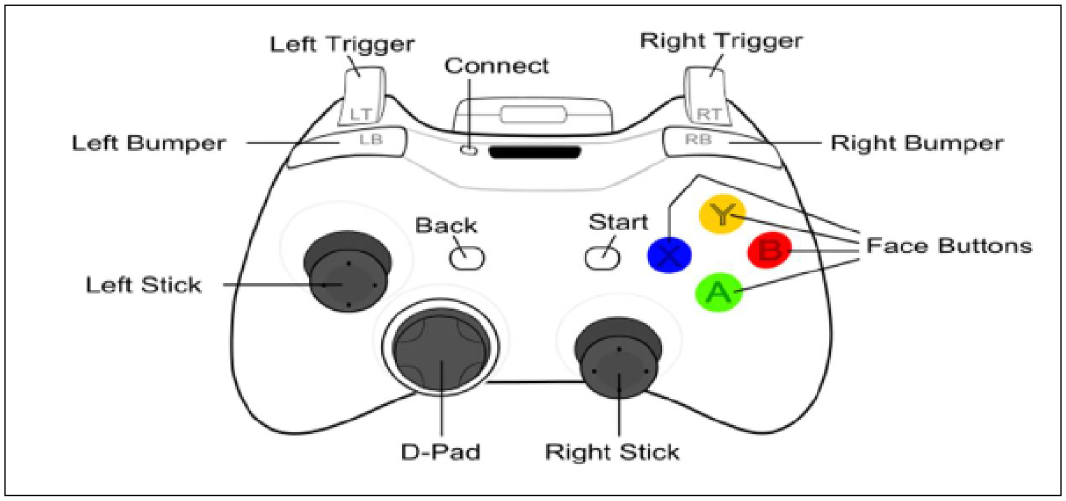
```
...
NODES
  /
    joy_node (joy/joy_node)
    rsdk_joint_position_joystick (baxter_examples/joint_position_
joystick.py)
...
[INFO] [1501196267.914400, 251.752000]: Robot Enabled
Press Ctrl-C to quit.
rightTrigger: left gripper close
rightTrigger: left gripper open
leftTrigger: right gripper close
leftTrigger: right gripper open
leftStickHorz: right inc right_s0
leftStickHorz: right dec right_s0
rightStickHorz: left inc left_s0
rightStickHorz: left dec left_s0
leftStickVert: right inc right_s1
leftStickVert: right dec right_s1
rightStickVert: left inc left_s1
rightStickVert: left dec left_s1
rightBumper: left: cycle joint
```

```
leftBumper: right: cycle joint
btnRight: left calibrate
btnLeft: right calibrate
function1: help
function2: help
Press Ctrl-C to stop.
```

The preceding output shows the Xbox joystick buttons and knobs to move Baxter's joints. The joystick controls two joints at a time on each of Baxter's two arms using the **Left Stick** and the **Right Stick** (see the following diagram). The up-down (vertical) control of the stick controls increasing and decreasing one of the joint angles. The side-to-side (horizontal) control increases and decreases another joint angle. The **Left Bumper** and **Right Bumper** cycle the joystick control through all of Baxter's arm joints in the order: S0-S1-E0-E1-W0-W1-W2. For example, initially, the Left Stick control will be in command of the (right arm) S0 joint using horizontal direction, and the S1 joint using vertical direction. When the Left Bumper is pressed, the **Left Stick** horizontal control will command the S1 joint, and the vertical control will command the E0 joint. Cycling the joints continues in a continuous loop, where the S0 joint will be selected after the W2 joint.

[The joystick mapping of *joystick left* = *robot right* allows the operator ease of use while the operator is positioned facing Baxter.]

The following image and table describe the mapping of the Xbox joystick controls:



Xbox joystick controls

Buttons	**Action for RIGHT Arm**	**Buttons**	**Action for LEFT Arm**
Back	Help	*Ctrl* + *C* or *Ctrl* + *Z*	Quit
Left Button (*X*)	gripper calibrate	Right Button (*B*)	gripper calibrate
Top Button (*Y*)	none	Bottom Button (*A*)	none
Left Trigger [PRESS]	gripper close	Right Trigger[PRESS]	gripper close
Left Trigger [RELEASE]	gripper open	Right Trigger[RELEASE]	gripper open
Left Bumper	cycle joints	Right Bumper	cycle joints
Stick Axes	**Action**		
Left Stick Horizontal	right: increase/decrease <current joint 1> (S0)		
Left Stick Vertical	right: increase/decrease <current joint 2> (S1)		
Right Stick Horizontal	left: increase/decrease <current joint 1> (S0)		
Right Stick Vertical	left: increase/decrease <current joint 2> (S1)		

Controlling arms with a Python script

In this section, we will create a simple Python script to command Baxter's arms into a specific pose. The following script commands Baxter's arms to a home position, similar to the untuck position. Comments have been placed throughout the code to provide information on the process. Further explanation of this Python code operation is given following the script:

```
#!/usr/bin/env python

"""
Script to return Baxter's arms to a "home" position
"""

# rospy - ROS Python API
import rospy

# baxter_interface - Baxter Python API
import baxter_interface

# initialize our ROS node, registering it with the Master
```

```
rospy.init_node('Home_Arms')

# create instances of baxter_interface's Limb class
limb_right = baxter_interface.Limb('right')
limb_left = baxter_interface.Limb('left')

# store the home position of the arms
home_right = {'right_s0': 0.08, 'right_s1': -1.00, 'right_w0': -0.67,
'right_w1': 1.03, 'right_w2': 0.50, 'right_e0': 1.18, 'right_e1':
1.94}
home_left = {'left_s0': -0.08, 'left_s1': -1.00, 'left_w0': 0.67,
'left_w1': 1.03, 'left_w2': -0.50, 'left_e0': -1.18, 'left_e1': 1.94}

# move both arms to home position
limb_right.move_to_joint_positions(home_right)
limb_left.move_to_joint_positions(home_left)

quit()
```

This code can be placed in a file named `home_arms.py`. Then, it can be made executable using the Ubuntu `chmod + x` command. Execute this Python script with this terminal command:

```
$ python home_arms.py
```

In this script, the `rospy` ROS-Python interface package is used to create ROS components from Python code. The `rospy` client API provides software routines for initializing the ROS node, `Home_Arms`, to invoke the process. The `baxter_interface` package provides the API for interacting with Baxter. In the script, we instantiate instances of the `Limb` class for both the right and left arms. A Python dictionary is used to assign joint angles to specific joints for both the right and left arms. These joint angle dictionaries are passed to the `move_to_joint_positions` method to command the respective arm to the provided position. The `move_to_joint_positions` method is also a part of the `baxter_interface` package.

Recording and replaying arm movements

Another capability provided by the `baxter_examples` programs is the ability to record and play back arm positions. A recorder program captures the time and joint positions in an external file. The `armRoutine` filename is used in the following command lines, but you may substitute your own filename instead. After the command for the recorder program is executed, the operator should move Baxter's arms manually or using the keyboard, joystick, ROS commands, or a script. When the operator wishes to end the recording, *Ctrl* + *C* or *Ctrl* + *Z* must be pressed to stop the recording. The playback program can be executed with the external file passed as a parameter. The playback program will run through the arm positions in the file once and then exit. The following instructions show the commands and the order of operation:

```
$ rosrun baxter_examples joint_recorder.py -f armRoutine
```

The output should be as follows:

```
Initializing node...
Getting robot state...
Enabling robot...
[INFO] [1501198989.301174, 2970.058000]: Robot Enabled
Recording. Press Ctrl-C to stop.
```

At this time, you should use your hands or the joystick, keyboard, Python script, and/or commands to move Baxter's arms. Press *Ctrl* + *C* when you are finished moving Baxter's arms. Next, execute the following command to play the file back:

```
$ rosrun baxter_examples joint_position_file_playback.py -f armRoutine
```

The output on the screen should be similar to the following:

```
Initializing node...
Getting robot state...
Enabling robot...
[INFO] [1501199319.366765, 3299.749000]: Robot Enabled
Playing back: armRoutine
Moving to start position...
 Record 10462 of 10462, loop 1 of 1
Exiting example...
```

If the file `armRoutine` is brought up in an editor, you should see that it contains data similar to the following:

```
time,left_s0,left_s1,left_e0,left_e1,left_w0,left_w1,left_w2,left_
gripper,right_s0,right_s1,right_e0,right_e1,right_w0,right_w1,right_
w2,right_gripper

0.221000,-0.0799704928309,-1.0000362209,-0.745950772448,-
0.0499208630966,-1.6948350728,1.03001017593,-0.500000660376,0.0,-
1.04466483018,-0.129655442605,1.5342459042,1.94952695585,-
0.909650985497,1.03000093981,0.825985250377,0.0

...
```

As shown, the first line contains the labels for the data on each of the subsequent rows. As the first label indicates, the first field contains the timestamp. The subsequent fields hold the joint positions for each of the left and right arm joints and grippers.

Baxter's arms and forward kinematics

Considering Baxter's arms up to the wrist cuff, each arm has seven values that define the rotation angle of each joint. Since the link lengths and joint angles are known, it is possible to calculate the position and orientation of the gripper attached to the wrist. This approach to calculating the **pose** of the gripper, given the configuration of the arm is called **forward kinematic analysis**.

Fortunately, ROS has programs that allow the calculation and publishing of the joint angles, given a particular position and orientation of the gripper. The particular topic for Baxter is `/robot/joint_states`.

Joints and joint state publisher

Baxter has seven joints in each of its two arms and two more joints in its head. The `/robot/joint_states` topic publishes the current joint states of the head pan (side-to-side) joint and the 14 arm joints. These joint states show position, velocity, and effort values for each of these joints. Joint position values are in radians, velocity values are in radians per second, and torque values are in Newton meters. The robot state publisher internally has a kinematic model of the robot. So, given the joint positions of the robot, the robot state publisher can compute and broadcast the 3D pose of each link in the robot.

For the examples in this section, it is assumed that Baxter Simulator is running, `baxter_world` is launched from `baxter_gazebo`, and the simulated robot is enabled:

```
$ cd baxter_ws
$ ./baxter.sh sim
$ roslaunch baxter_gazebo baxter_world.launch
```

In a second terminal, type the following commands:

```
$ cd baxter_ws
$ ./baxter.sh sim
$ rosrun baxter_tools enable_robot.py -e
```

Baxter's arms will be placed in the *home* position using the Python script presented previously via the following command:

```
$ python home_arms.py
```

The joint states will be displayed with the screen output edited to show the arm positions as angles of rotation in radians. To view one output of the joint states, type this:

```
$ rostopic echo /robot/joint_states -n1
```

Here is our output on the screen:

```
header:
  seq: 42448
  stamp:
    secs: 850
    nsecs: 553000000
  frame_id: ''

name: ['head_pan', 'l_gripper_l_finger_joint', 'l_gripper_r_finger_
joint', 'left_e0', 'left_e1', 'left_s0', 'left_s1', 'left_w0', 'left_
w1', 'left_w2', 'r_gripper_l_finger_joint', 'r_gripper_r_finger_joint',
'right_e0', 'right_e1', 'right_s0', 'right_s1', 'right_w0', 'right_w1',
'right_w2']

position: [9.642012118504795e-06 (Head),
Left: -9.409649977892339e-08, -0.02083311343765363, -1.171334885477055,
1.9312641121225074, -0.07941855421008803, -0.9965989736590268,
0.6650922280384437, 1.0314330310192892, -0.49634000104265397,


Right: 0.020833000098799456, 2.9266103072174966e-10, 1.1714460516466971,
1.9313701087550257, 0.07941788278369621, -0.9966421178258322,
-0.6651529936706897, 1.0314155121179436, 0.49638770883940264]


velocity: [8.463358573117045e-09, 2.2845555643152853e-05,
2.766005018018799e-05, 6.96516608889685e-08, -1.4347584964474649e-07,
5.379243329637427e-08, -3.07783563763457e-08, -5.9625446169838476e-06,
-2.765075210928186e-06, 4.37915209815064e-06, -1.9330586583769175e-08,
-3.396963606705046e-08, -4.1024914575147146e-07, -6.470964538079114e-07,
1.2464164369422782e-07, -3.489373517131325e-08, 1.3838850846575283e-06,
1.1659521943505596e-06, -3.293066091641411e-06]


effort: [0.0, 0.0, 0.0, -0.12553439407980704, -0.16093410986695034,
1.538268268319598e-06, -0.1584186302672208, 0.0026223415490989055,
-0.007023475006633362, -0.0002595722218323715, 0.0, 0.0,
0.12551329635801522, -0.16096013901023554, -1.4389475655463002e-05,
-0.1583874287014453, -0.0026439994199378702, -0.0070054474078151685,
0.00024931690616014635]
```

Compare the radian values from `home_arms.py` and the result of `rostopic echo` of joint states, but watch the order of listing of the joints:

```
    # store the home position of the arms
    home_right = {'right_s0': 0.08, 'right_s1': -1.00, 'right_w0': -0.67,
    'right_w1': 1.03, 'right_w2': 0.50, 'right_e0': 1.18, 'right_e1':
    1.94}
    home_left = {'left_s0': -0.08, 'left_s1': -1.00, 'left_w0': 0.67,
    'left_w1': 1.03, 'left_w2': -0.50, 'left_e0': -1.18, 'left_e1': 1.94}
```

The velocity and effort (torque) terms are essentially zero, since Baxter's arms are not moving. Rounding off the arm joint position values to two places shows that the angular positions of the arm joints are equivalent to the values in the Python script.

We find the type of messages for joint states from `sensor_msgs` using this command:

```
$ rostopic type /robot/joint_states
```

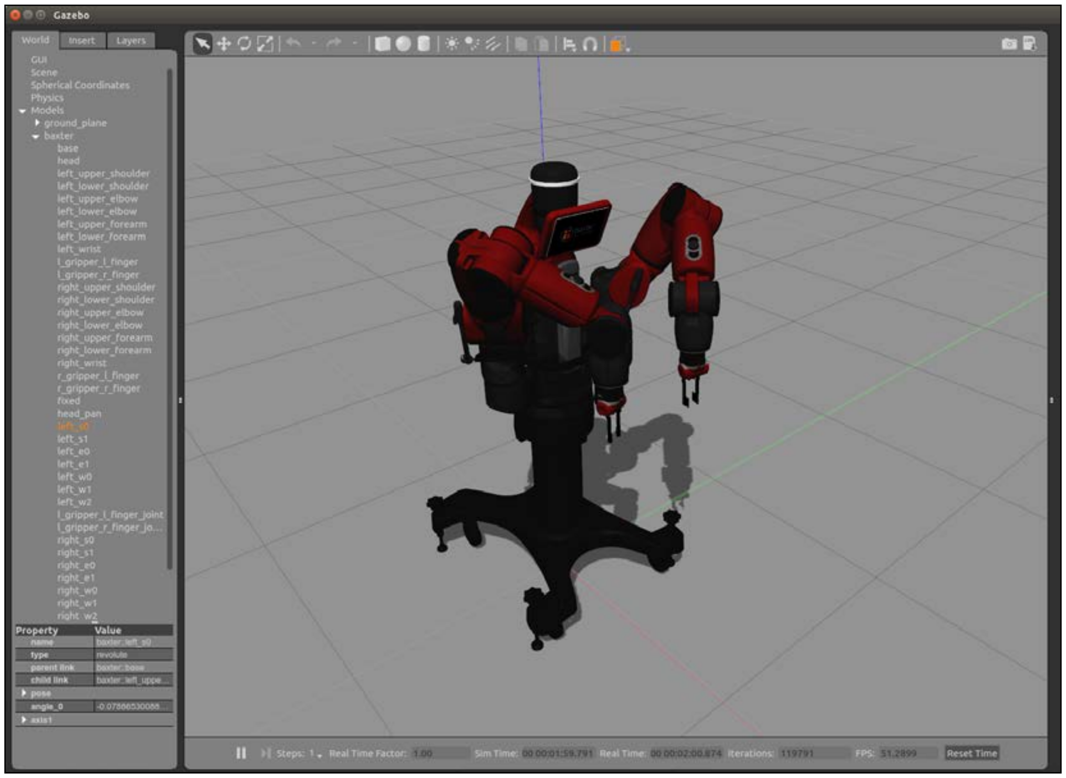
The output is as follows:

```
sensor_msgs/JointState
```

To show the `home_arms` pose for Baxter in Gazebo, follow these steps:

1. Go to **World** | **Models**, click on **baxter**, and then select **left_s0**.
2. Pull the **Property** window into view by clicking and dragging the three small ticks above this panel.

   The figure should look like this:



Property	Value
name	baxter::left_s0
type	revolute
parent link	baxter::base
child link	baxter::left_uppe...
pose	
angle_0	-0.07886530088...
axis1	

Baxter home position

3. Choose **pose** and look at the value of `angle_0`: `-0.07886530088`. Rounded off, this is `left_s0`: -0.08 selected in `home_arms.py`. You can view other information by selecting another joint or link of Baxter from the **World** panel.

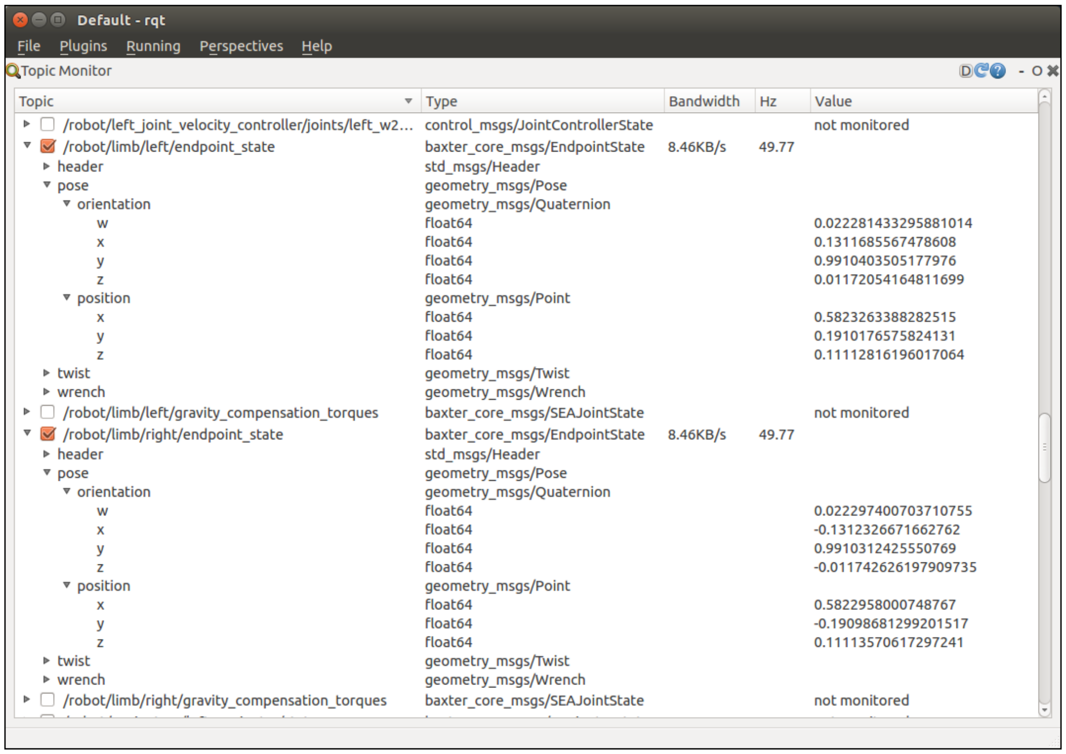
Another command shows the position and orientation of the end of the left arm:

```
$ rostopic echo /robot/limb/left/endpoint_state/pose -n1
```

The output should be similar to the following:

```
---
header:
  seq: 62403
  stamp:
    secs: 1249
    nsecs: 653000000
  frame_id: ''
pose:
  position:
    x: 0.582326339279
    y: 0.191017651504
    z: 0.111128161508
  orientation:
    x: 0.131168552915
    y: 0.991040351028
    z: 0.0117205349262
    w: 0.0222814367168
```

Yet another way to see the values is to start rqt and select **Topics** as **Plugins** and **Topic Monitor**. Select the `/robot/limb/left/endpoint_state` and `/robot/limb/right/endpoint_state` topics. The result is shown in the following screenshot:



Topic	Type	Bandwidth	Hz	Value
/robot/left_joint_velocity_controller/joints/left_w2...	control_msgs/JointControllerState			not monitored
/robot/limb/left/endpoint_state	baxter_core_msgs/EndpointState	8.46KB/s	49.77	
header	std_msgs/Header			
pose	geometry_msgs/Pose			
orientation	geometry_msgs/Quaternion			
w	float64			0.022281433295881014
x	float64			0.1311685567478608
y	float64			0.9910403505177976
z	float64			0.01172054164811699
position	geometry_msgs/Point			
x	float64			0.5823263388282515
y	float64			0.1910176575824131
z	float64			0.11112816196017064
twist	geometry_msgs/Twist			
wrench	geometry_msgs/Wrench			
/robot/limb/left/gravity_compensation_torques	baxter_core_msgs/SEAJointState			not monitored
/robot/limb/right/endpoint_state	baxter_core_msgs/EndpointState	8.46KB/s	49.77	
header	std_msgs/Header			
pose	geometry_msgs/Pose			
orientation	geometry_msgs/Quaternion			
w	float64			0.022297400703710755
x	float64			-0.1312326671662762
y	float64			0.9910312425550769
z	float64			-0.011742626197909735
position	geometry_msgs/Point			
x	float64			0.5822958000748767
y	float64			-0.19098681299201517
z	float64			0.11113570617297241
twist	geometry_msgs/Twist			
wrench	geometry_msgs/Wrench			
/robot/limb/right/gravity_compensation_torques	baxter_core_msgs/SEAJointState			not monitored

Topic Monitor in rqt for endpoint states

The left arm's endpoint *x*, *y*, and *z* position agrees with the output from the `rostopic echo` command for the left `endpoint_state` topic. The right arm endpoint has the same *x* and *z* positions, but a negative value for *y*. This indicates that it is to the right of Baxter's vertical centerline.

Understanding tf

Tf is a transform system used to keep track of the relation between different coordinate frames in ROS. The relationship between the coordinate frames is maintained in a tree structure that can be viewed. In Baxter's example, the robot has many coordinate frames that can be referenced to Baxter's base frame.

Tutorials about tf are given on the ROS wiki at `http://wiki.ros.org/tf/Tutorials`.

To demonstrate the use of tf, the following Baxter examples will be provided:

- Show tf in rviz after Baxter's arms are moved to a position in which the angles of the joints are zero
- Show various coordinate frames for Baxter's elements, such as cameras or grippers

A program to move Baxter's arms to a zero angle position

With Baxter Simulator running (Gazebo), executing the Python script `arms_to_zero_angles.py` will move Baxter's arms to a position in which all the joint angles are zero.

The following code simply sets the joint angles to zero:

```
#!/usr/bin/env python   # arms_to_zero_angles.py
#
"""
Script to return Baxter's arms to a " zero" position
"""

# rospy - ROS Python API
import rospy

# baxter_interface - Baxter Python API
import baxter_interface

# initialize our ROS node, registering it with the Master
rospy.init_node('Zero_Arms')

# create instances of baxter_interface's Limb class
limb_right = baxter_interface.Limb('right')
limb_left = baxter_interface.Limb('left')

# store the zero position of the arms
zero_zero_right = {'right_s0': 0.0, 'right_s1': 0.00, 'right_w0':
0.00, 'right_w1': 0.00, 'right_w2': 0.00, 'right_e0': 0.00, 'right_
e1': 0.00}
zero_zero_left = {'left_s0': 0.0, 'left_s1': 0.00, 'left_w0': 0.00,
'left_w1': 0.00, 'left_w2': 0.00, 'left_e0': 0.00, 'left_e1': 0.00}

# move both arms to zero position
```

```
    limb_right.move_to_joint_positions(zero_zero_right)
    limb_left.move_to_joint_positions(zero_zero_left)

    quit()
```

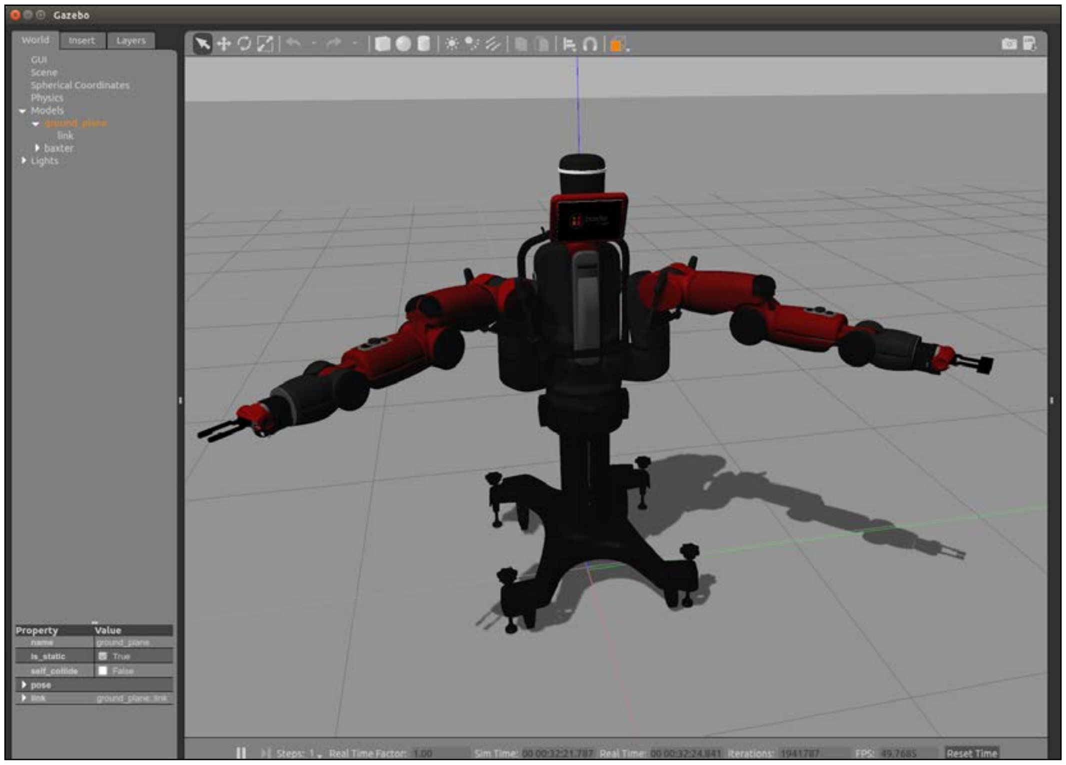
Make the Python script executable:

```
$ chmod +x arms_to_zero_angles.py
```

Then, run the script:

```
$ python arms_to_zero_angles.py
```

The position of the arms can be visualized in Gazebo and the values for position, velocity, and effort can be displayed. In the following Gazebo window, Baxter has arms outstretched at an angle from its torso:



Property	Value
name	ground_plane
is_static	True
self_collide	False
pose	
link	ground_plane::link

Steps: 1 Real Time Factor: 1.00 Sim Time: 00 00:32:21.787 Real Time: 00 00:32:24.841 Iterations: 1941787 FPS: 49.7685

Baxter's joints at zero degrees

The results of the joint states showing only the name and position are as follows:

```
---
header:
  seq: 120710
  stamp:
    secs: 2415
    nsecs: 793000000
  frame_id: ''

name: ['head_pan', 'l_gripper_l_finger_joint', 'l_gripper_r_finger_
joint', 'left_e0', 'left_e1', 'left_s0', 'left_s1', 'left_w0', 'left_
w1', 'left_w2', 'r_gripper_l_finger_joint', 'r_gripper_r_finger_joint',
'right_e0', 'right_e1', 'right_s0', 'right_s1', 'right_w0', 'right_w1',
'right_w2']

position: [2.1480795875383762e-05,
0.02083300010459807, 7.094235804419552e-09,
 -0.0052020885142498585, 0.008648349108503872, -0.0003526224733487737,
-0.004363080957646481, 0.0029469234535000055, 0.004783709772852696,
-0.0022098995590349446,
-4.685055459408831e-10, -0.02083300002921974, 0.005137618938708677,
0.008541712202397633, 0.0003482148331919177, -0.004308001456631239,
-0.0029103069740452625, 0.004726431947482013, 0.002182588672263286]
```

Within numerical error tolerance, the values are zero for the arm joint angles.

Commanding the joint angles directly

You can send joint angles directly to Baxter using the `JointCommand` message from the `baxter_core_messages` package.

The `JointCommand` message is defined as follows:

```
int32 mode
float64[] command
string[] names

int32 POSITION_MODE=1
int32 VELOCITY_MODE=2
int32 TORQUE_MODE=3
int32 RAW_POSITION_MODE=4
```

The message defines the control mode, the command as an angle for the joints, and the names of the joints being controlled. The details of this are discussed on the following website:

`http://sdk.rethinkrobotics.com/wiki/Arm_Control_Modes`

As an example, move Baxter's arms into an arbitrary pose and then, to set the angles of four of Baxter's joints to zero using position control, type this command:

```
$ rostopic pub /robot/limb/left/joint_command baxter_core_msgs/
JointCommand
"{mode: 1, command: [0.0, 0.0, 0.0, 0.0], names: ['left_w1', 'left_e1',
'left_s0', 'left_s1']}" -r 10
```

rviz tf frames

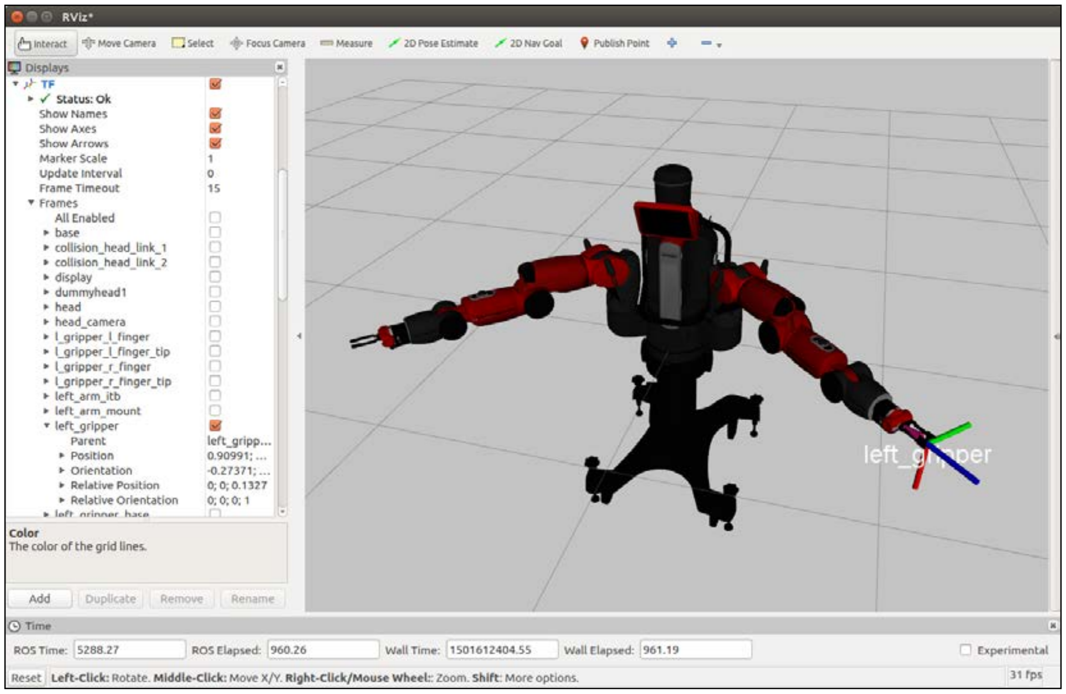
With Gazebo running and Baxter's arms in a zero-angle pose, start rviz:

```
$ rosrun rviz rviz
```

Now select the parameters for rviz:

1. Select the field next to **Fixed Frame** (under **Global Options**) and select `base`.
2. Select the **Add** button under the **Displays** panel, add **Robot Model**, and you will see Baxter appear.
3. Select the **Add** button under the **Displays** panel, add **TF**, and see all the frames that are too complicated to use.
4. Arrange the windows to see the left panel and the figure. Close the **Views** window on the right panel.
    - Expand **TF** in the **Displays** panel by clicking on the triangle symbol on the left
    - Under **TF**, expand **Frames** by clicking on the left triangle
    - Uncheck the checkbox next to **All Enabled**
5. Now, check `left_gripper` to display the axes.

The rviz display looks similar to the following screenshot:



```
TF                              ✓
  Status: Ok
  Show Names                    ✓
  Show Axes                     ✓
  Show Arrows                   ✓
  Marker Scale                  1
  Update Interval               0
  Frame Timeout                 15
  Frames
    All Enabled                 ☐
    base                        ☐
    collision_head_link_1       ☐
    collision_head_link_2       ☐
    display                     ☐
    dummyhead1                  ☐
    head                        ☐
    head_camera                 ☐
    l_gripper_l_finger          ☐
    l_gripper_l_finger_tip      ☐
    l_gripper_r_finger          ☐
    l_gripper_r_finger_tip      ☐
    left_arm_itb                ☐
    left_arm_mount              ☐
    left_gripper                ✓
      Parent                    left_gripp...
      Position                  0.90991; ...
      Orientation               -0.27371; ...
      Relative Position         0; 0; 0.1327
      Relative Orientation      0; 0; 0; 1
    left_gripper_base           ☐

Color
The color of the grid lines.

ROS Time: 5288.27   ROS Elapsed: 960.26   Wall Time: 1501612404.55   Wall Elapsed: 961.19   ☐ Experimental
Left-Click: Rotate. Middle-Click: Move X/Y. Right-Click/Mouse Wheel:: Zoom. Shift: More options.   31 fps
```

tf transform base and left gripper

You will see the left gripper axes in color on your screen: *x* is down (red), *y* is to the right (green), and *z* is forward (blue) in the preceding screenshot.

Now you can choose various elements of Baxter to see the tf coordinate axes.

Viewing a tf tree of robot elements

The `view_frames` program can generate a PDF file with a graphical representation of the complete tf tree. To try the program, Baxter Simulator or the real Baxter should be communicating with the terminal window. To run `view_frames`, use the following command:

```
$ rosrun tf view_frames
```

In the current working folder, you should now have a file called `frames.pdf`. Open the file with the following command:

```
$ evince frames.pdf
```

More information about the tf frames can be found at `http://wiki.ros.org/tf/Tutorials/Introduction%20to%20tf`.

Introducing MoveIt!

One of the challenging aspects of robotics is defining a path for the motion of a robot's arms to grasp an object, especially when obstacles may obstruct the most obvious path of motion. Fortunately, a ROS package called MoveIt! allows us to plan and execute a complicated trajectory.

A video created by Rethink Robotics shows how to use MoveIt! to plan the motion of Baxter's arms and then have MoveIt! actually cause a real or simulated Baxter to execute that motion. To see the video, go to: `https://www.youtube.com/watch?feature=player_detailpage&v=1Zdkwym42P4`.

A tutorial is available on the Rethink wiki site at `http://sdk.rethinkrobotics.com/wiki/MoveIt_Tutorial`.

First, start the Baxter simulator in Gazebo:

```
$ cd baxter_ws
$ ./baxter.sh sim
$ roslaunch baxter_gazebo baxter_world.launch
```

In a second terminal window, untuck Baxter's arms and start the Python script that starts `joint_trajectory_action_server`:

```
$ cd baxter_ws
$ ./baxter.sh sim
$ rosrun baxter_tools tuck_arms.py -u
$ rosrun baxter_interface joint_trajectory_action_server.py
```

The output on the screen should be as follows:

```
Initializing node...
Initializing joint trajectory action server...
Running. Ctrl-c to quit
```

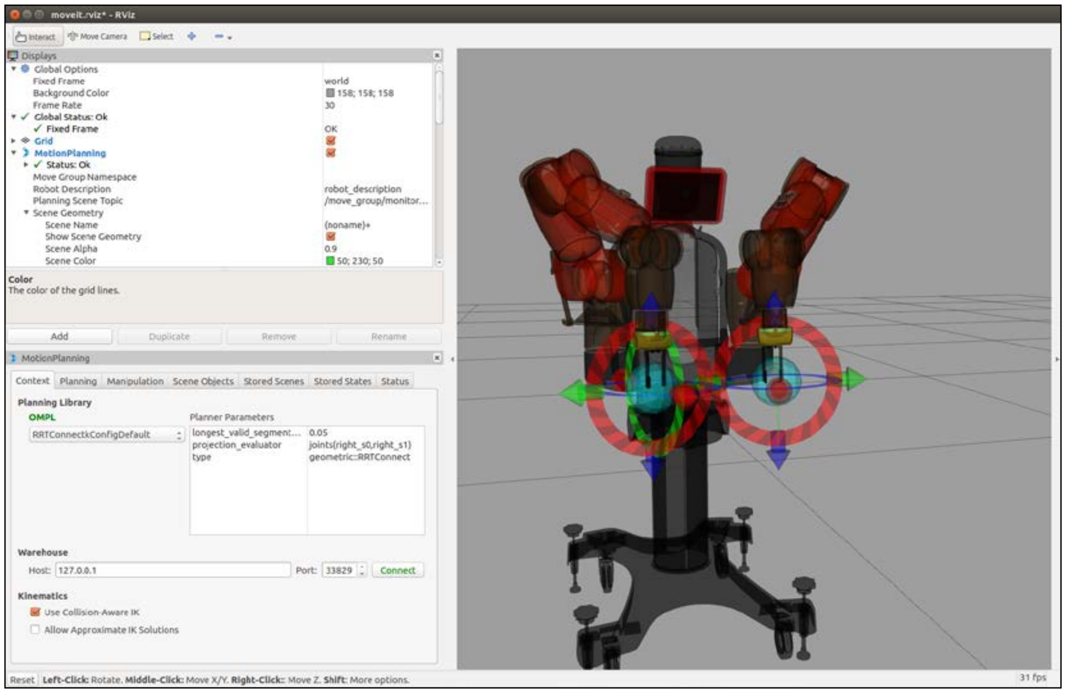
In a third terminal, start MoveIt! and wait for the response:

```
$ cd baxter_ws
$ ./baxter.sh sim
$ roslaunch baxter_moveit_config baxter_grippers.launch
```

Look for the output:

```
. . .
You can start planning now!
```

Looking at the Gazebo window and the MoveIt! window, you'll see that Baxter looks the same in terms of the positions of its arms:



```
Displays
▾ Global Options
    Fixed Frame                 world
    Background Color            ▨ 158; 158; 158
    Frame Rate                  30
▾ ✓ Global Status: Ok
    ✓ Fixed Frame               OK
▸ Grid                          ✓
▾ MotionPlanning                ✓
  ▸ ✓ Status: Ok
    Move Group Namespace
    Robot Description           robot_description
    Planning Scene Topic        /move_group/monitor...
  ▾ Scene Geometry
      Scene Name                (noname)+
      Show Scene Geometry       ✓
      Scene Alpha               0.9
      Scene Color               🟩 50; 230; 50

Color
The color of the grid lines.

MotionPlanning
Planning Library
  OMPL
  RRTConnectkConfigDefault
Planner Parameters
  longest_valid_segment...   0.05
  projection_evaluator       joints(right_s0,right_s1)
  type                       geometric::RRTConnect
Warehouse
  Host: 127.0.0.1     Port: 33829
Kinematics
  [✓] Use Collision-Aware IK
  [ ] Allow Approximate IK Solutions

Left-Click: Rotate. Middle-Click: Move X/Y. Right-Click:: Move Z. Shift: More options.
```

MoveIt! startup

In the screenshot, which is the rviz screen, the **Displays** and **Motion Planning** windows are shown on the left with the **Context** tab information showing. On the right, you can see the simulated Baxter in the starting position of MoveIt! with its arms untucked.

You can select any one of the **Displays** categories and modify the parameters. For example, the screenshot shows Baxter with a lightened **Background Color** chosen under **Global Options**.

Under the **MotionPlanning** panel, the **Context**/**Planning**/**Manipulation**/**Scene Objects**/**Stored Scenes**/**Stored States**/**Status** tabs are defined in the following table:

Tab	**Uses**
Context	Select the planning library and planner parameters; set collision awareness for IK solver
Planning	Set the start state, the goal state, and plan and execute moves of Baxter's arms
Manipulation	Object detection and manipulation
Scene Objects	Import or export scenes such as pillars or tabletops from a disk file or database, manipulate objects, and Publish Scene
Stored Scenes	Stored scenes on a database
Stored States	Store and load robot states
Status	Status

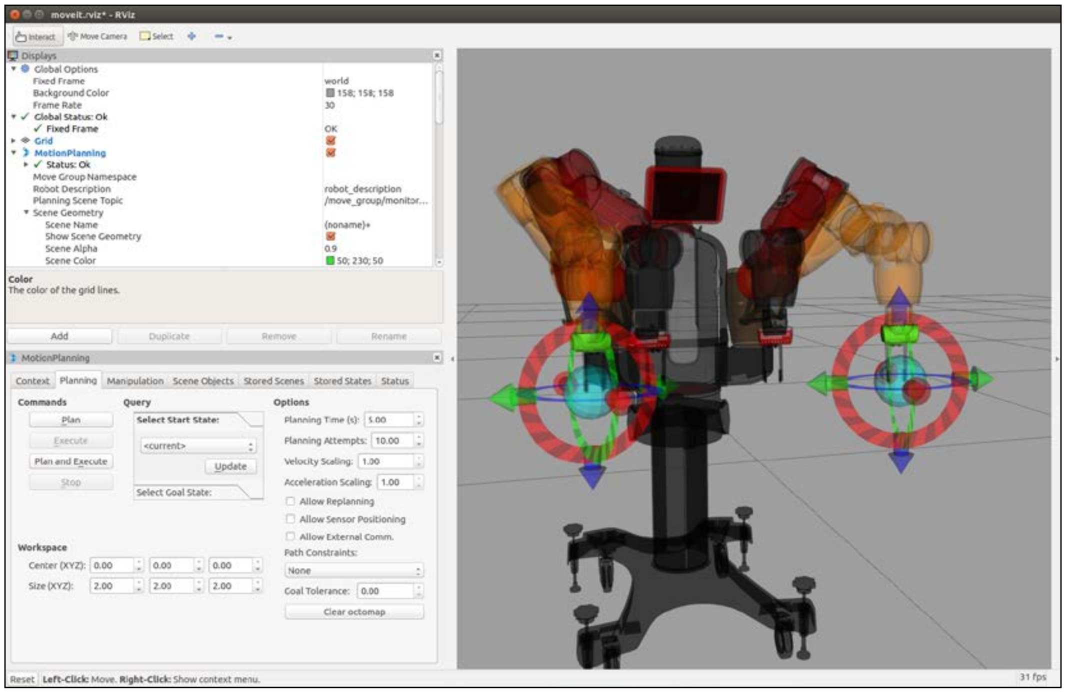
Planning a move of Baxter's arms with MoveIt!

Click on the **Planning** tab under **MotionPlanning**. On the **Planning** panel, look for the **Query** field and the **Select Start State** heading. Click on **Select Start State** to reveal a menu box and set it to `<current>`. Then, click on the **Update** button.

Now to move Baxter's arms, we will do the following:

1. Use arrows and rings to move Baxter's simulated arms to the desired positions. The desired goal positions should appear in orange in the simulation window.
2. Under the **Commands** area, choose the **Plan** button to see the trajectory of Baxter's arms in MoveIt!.
3. Choose the **Execute** button to see Baxter's arms move to the goal positions.

4. You should see red arms move from the start state to the final (goal) states:



```
Global Options
  Fixed Frame                 world
  Background Color            158; 158; 158
  Frame Rate                  30
Global Status: Ok
  Fixed Frame                 OK
Grid                          ✓
MotionPlanning                ✓
  Status: Ok
  Move Group Namespace
  Robot Description           robot_description
  Planning Scene Topic        /move_group/monitor...
  Scene Geometry
    Scene Name                (noname)+
    Show Scene Geometry       ✓
    Scene Alpha               0.9
    Scene Color               50; 230; 50
```

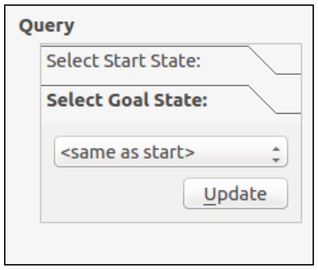
Color
The color of the grid lines.

```
Select Start State:  <current>
Select Goal State:

Workspace
  Center (XYZ):  0.00  0.00  0.00
  Size (XYZ):    2.00  2.00  2.00

Options
  Planning Time (s):      5.00
  Planning Attempts:      10.00
  Velocity Scaling:       1.00
  Acceleration Scaling:   1.00
  ☐ Allow Replanning
  ☐ Allow Sensor Positioning
  ☐ Allow External Comm.
  Path Constraints:       None
  Goal Tolerance:         0.00
```

Left-Click: Move. **Right-Click:** Show context menu.

Baxter's arms and goal state for its arms

Next, have the arms move back to the original start positions to perform another move. To do this, under the **Query** field, click on **Select Goal State** to reveal the menu box and set it to `<same as start>`. Then, click on the **Update** button:



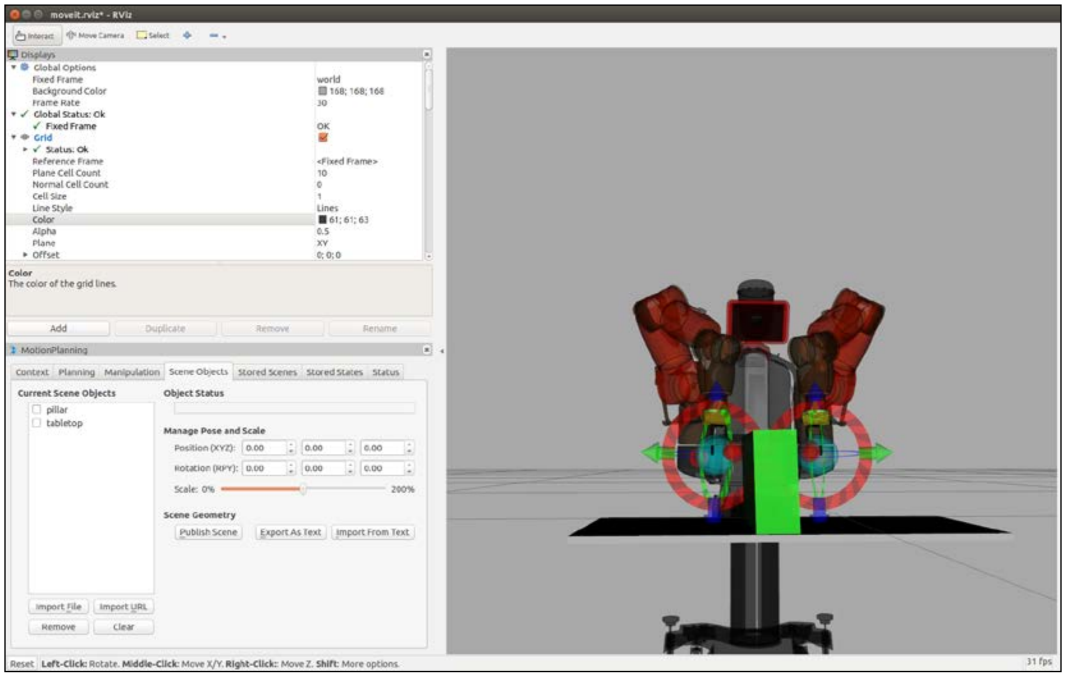
Query field to return arms to start positions

Adding objects to a scene

Select the **Scene Objects** tab from the **MotionPlanning** frame. If you have scenes in your computer's directories, you can import them using the **Scene Geometry** field by selecting **Import From Text**. Alternatively, use the following scene, which we created in the `PillarTable.scene` file. Make sure that this file is saved as a text file.

```
(noname)+
* pillar
1
box
0.308 0.13056 0.6528
0.7 -0.01 0.03
0.0108439 0.706876 0.0103685 0.707178
0 0 0 0
* tabletop
1
box
0.7 1.3 0.02
0.7 0.04 -0.13
0 0 0 1
0.705882 0.705882 0.705882 1
.
```

The first line of dimensions under `box` in the file represents the height, width, and length of the pillar in meters. The next line defines the position from Baxter's origin. The third line is the pose of the pillar as a quaternion. The following screenshot shows the results of importing the scene elements:



Global Options	
Fixed Frame	world
Background Color	168; 168; 168
Frame Rate	30
✓ Global Status: Ok	
✓ Fixed Frame	OK
Grid	✓
✓ Status: Ok	
Reference Frame	<Fixed Frame>
Plane Cell Count	10
Normal Cell Count	0
Cell Size	1
Line Style	Lines
Color	61; 61; 63
Alpha	0.5
Plane	XY
Offset	0; 0; 0

Color
The color of the grid lines.

Current Scene Objects
- ☐ pillar
- ☐ tabletop

Object Status

Manage Pose and Scale

Position (XYZ): 0.00 0.00 0.00
Rotation (RPY): 0.00 0.00 0.00
Scale: 0% 200%

Scene Geometry

Left-Click: Rotate. **Middle-Click:** Move X/Y. **Right-Click:**: Move Z. **Shift:** More options.

Baxter with tabletop and pillar

To manipulate the objects by moving or rotating them, select the object name (not the checkbox), and the arrows and rings should appear. Change their position with the green and blue arrows and rotate them with the ring. Moving the **Scale** slider will change the size of the object. Move the mouse to rotate the object, and roll the mouse wheel, if you have one, to zoom the object's size. You can save the scene (**Export As Text**) after you finish manipulating it.

Next, on the **Scene Objects** panel, click on the **Publish Scene** button under the **Scene Geometry** field. *This step is important and tells MoveIt! to plan around obstacles in the environment!*

Position of objects

When **pillar** is selected under **Current Scene Objects** (select the word, not the checkbox), values for its **Position (XYZ)** and **Rotation (RPY)** appear under **Manage Pose and Scale**. The position of *x*, *y*, and *z* of the centroid of the pillar is shown with respect to Baxter's origin. Baxter's *x* axis extends outward toward the viewer. The positive *y* axis is to the right in the view and the *z* axis runs upward. Note that the roll, pitch, and yaw are about the *x*, *y*, and *z* axes, respectively.

Planning a move to avoid obstacles with MoveIt!

In the following example, the left arm is going to move to the other side of the obstacle. MoveIt! will plan the trajectory so that Baxter's arm will not hit the pillar.

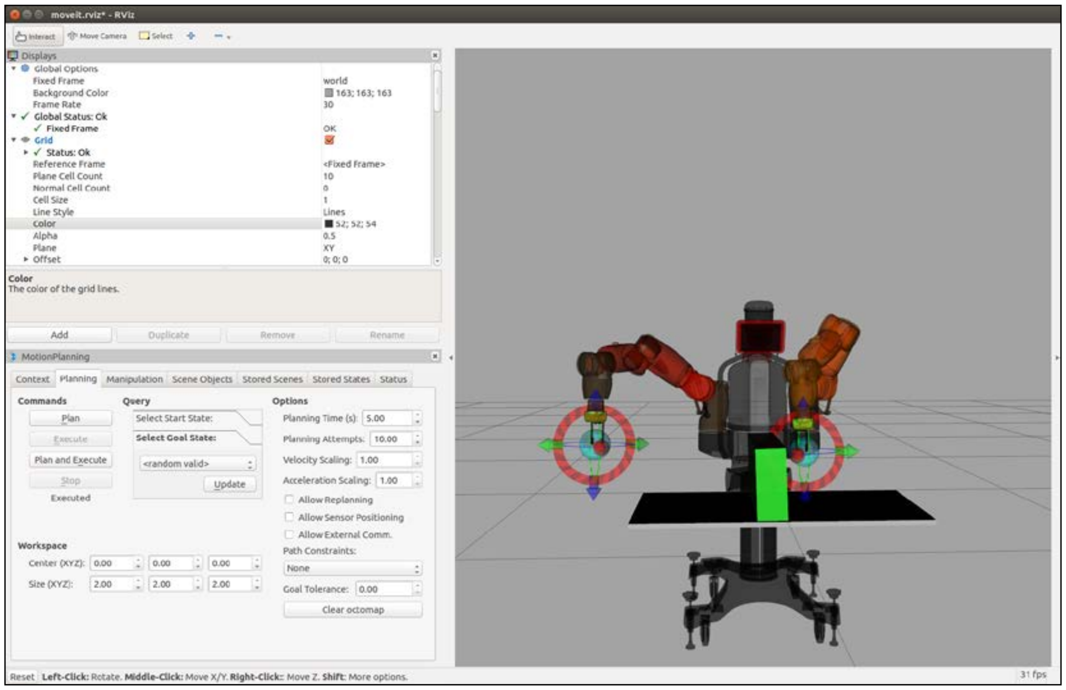
Return to the **Planning** tab on the **MotionPlanning** panel. First, move Baxter's right arm away from the pillar. **Plan** and then **Execute** moving the right arm.

We can now drag our interactive markers for Baxter's left arm to move the goal state to a location on the opposite side of the pillar. Each time you click on the **Plan** button, a different arm trajectory path is shown on the virtual Baxter. Each path avoids collision with the pillar.

Caution!

We move Baxter's other arm (the right arm, in this case) out of the way to avoid any possible collisions. This is necessary if the MoveIt! trajectories are used on the real Baxter. Move the right arm with the markers and choose **Plan** and **Execute**.

The following screenshot shows Baxter prior to moving the left arm around the pillar. Notice that the right arm is moved out of the way:



Global Options	
Fixed Frame	world
Background Color	163; 163; 163
Frame Rate	30
✓ Global Status: Ok	
✓ Fixed Frame	OK
Grid	✓
✓ Status: Ok	
Reference Frame	<Fixed Frame>
Plane Cell Count	10
Normal Cell Count	0
Cell Size	1
Line Style	Lines
Color	■ 52; 52; 54
Alpha	0.5
Plane	XY
Offset	0; 0; 0

Color
The color of the grid lines.

Select Start State:
Select Goal State: <random valid>

Planning Time (s): 5.00
Planning Attempts: 10.00
Velocity Scaling: 1.00
Acceleration Scaling: 1.00
☐ Allow Replanning
☐ Allow Sensor Positioning
☐ Allow External Comm.
Path Constraints: None
Goal Tolerance: 0.00

Center (XYZ): 0.00 0.00 0.00
Size (XYZ): 2.00 2.00 2.00

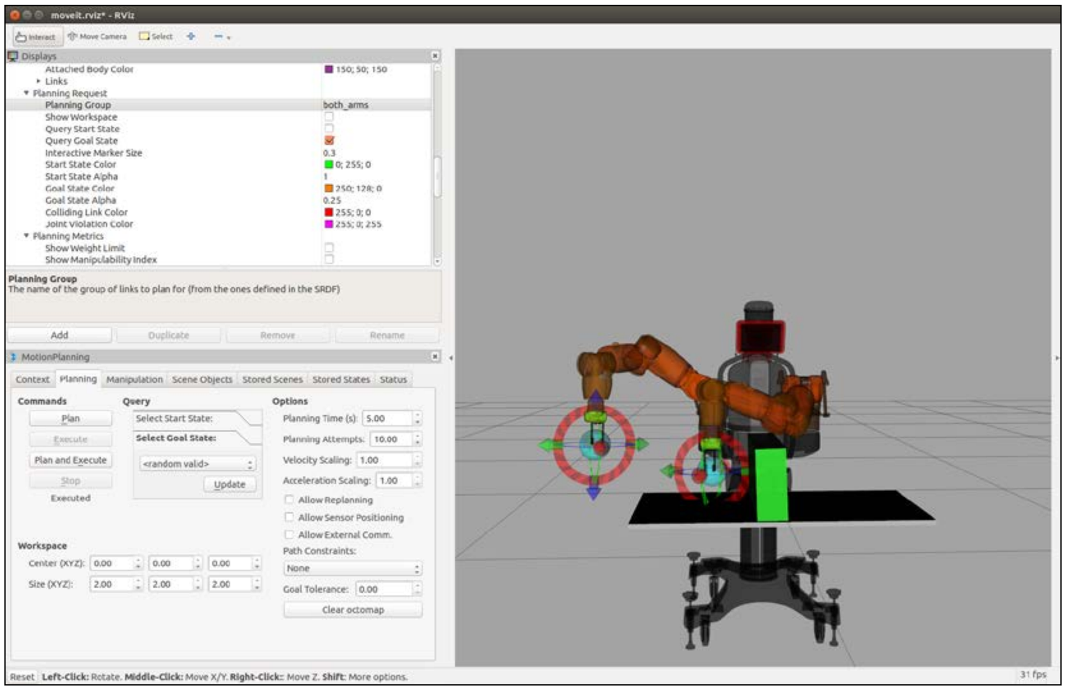
Reset **Left-Click:** Rotate. **Middle-Click:** Move X/Y. **Right-Click:**: Move Z. **Shift**: More options.

Baxter's right arm moved aside

Caution!

When using MoveIt! with the real Baxter, the arms sometimes move into odd positions. If this happens, move them apart and restart MoveIt!.

Now, click on **Execute** to see Baxter's arm avoid the obstacle and move to the goal position on the other side of the pillar. The following screenshot shows that Baxter's left arm has moved around the pillar to the other side of it:



Attached Body Color	150; 50; 150
▸ Links	
▾ Planning Request	
Planning Group	both_arms
Show Workspace	☐
Query Start State	☐
Query Goal State	☑
Interactive Marker Size	0.3
Start State Color	0; 255; 0
Start State Alpha	1
Goal State Color	250; 128; 0
Goal State Alpha	0.25
Colliding Link Color	255; 0; 0
Joint Violation Color	255; 0; 255
▾ Planning Metrics	
Show Weight Limit	☐
Show Manipulability Index	☐

Planning Group
The name of the group of links to plan for (from the ones defined in the SRDF)

Commands
Executed

Query
Select Start State:
Select Goal State: <random valid>

Options
Planning Time (s): 5.00
Planning Attempts: 10.00
Velocity Scaling: 1.00
Acceleration Scaling: 1.00
☐ Allow Replanning
☐ Allow Sensor Positioning
☐ Allow External Comm.
Path Constraints: None
Goal Tolerance: 0.00

Workspace
Center (XYZ): 0.00 0.00 0.00
Size (XYZ): 2.00 2.00 2.00

Reset **Left-Click:** Rotate. **Middle-Click:** Move X/Y. **Right-Click:** Move Z. **Shift:** More options.

Baxter's simulated arm moved to the other side of the obstacle

Configuring a real Baxter setup

In the *Installing Baxter SDK software* section, we loaded the workstation computer SDK software used to control Baxter. This control can be either through the `baxter_example` programs described previously, through Python scripts, or using the command line. To control the real Baxter, the `baxter.sh` script is used to set up the environmental variables for configuring the network for Baxter. Ensure that you have completed the *Configuring Baxter shell* section before continuing.

The Baxter Research Robot can be configured to communicate with a development workstation computer over various network configurations. An Ethernet network or a wireless network can be established between Baxter and the workstation computer for bi-directional communication. For full descriptions of the various network configurations for Baxter and the workstation, refer to the Rethink wiki at `http://sdk.rethinkrobotics.com/wiki/Networking`.

To communicate with Baxter, the `baxter.sh` script must be edited (again) to modify the `baxter_hostname` variable. The default for `baxter_hostname` is the robot's serial number, located on the back of the robot, next to the power button. An alternative method is to assign a new robot hostname using the **Field Service Menu** (**FSM**), accessed by plugging a USB keyboard into the back of Baxter. Refer to the Rethink wiki for more details (`http://sdk.rethinkrobotics.com/wiki/Field_Service_Menu_(FSM)`).

Using Baxter's serial number or assigned hostname within the `baxter.sh` script, find and edit the following line:

```
baxter_hostname="baxter_hostname.local"
```

Next, verify that either the `your_ip` variable *or* the `your_hostname` variable is set to specify the IP address or hostname of your workstation computer. The IP address should be the one assigned for the network connection. (When using the `$ ifconfig` command, this would be the IP address associated with the `inet addr` field.)
The hostname must be resolvable by Baxter to identify the workstation computer hostname.

If there is any doubt, use the `ping <IP address>` or `ping <hostname>` command to verify that the network communication is working.

Find and modify one of the following variables inside the `baxter.sh` script:

```
your_ip="192.168.XXX.XXX"
your_hostname="my_computer.local"
```

Do not use both of these variables; otherwise `your_hostname` will take precedence. The unused variable should be commented out with a `#` symbol. Save your changes to `baxter.sh`.

To verify the ROS environment setup for Baxter, it is wise to run the `baxter.sh` script and verify the ROS variables. To do so, use the following commands:

```
$ cd ~/baxter_ws
$ ./baxter.sh
$ env | grep ROS
```

Check whether these important fields are set with the correct information:

- `ROS_MASTER_URI`: This should now contain Baxter's hostname
- `ROS_IP`: This should contain the workstation computer's IP address

  Or:
- `ROS_HOSTNAME`: If not using the IP address, this field should contain the workstation computer's hostname

Again, a cheat sheet for use with the real Baxter is provided for you to use with the example programs that follow. The commands for communicating, enabling, and untucking are provided here for your reference:

Real Baxter cheat sheet

To communicate with the real Baxter, use the following commands:

```
$ cd ~/baxter_ws
$ ./baxter.sh
```

For subsequent terminal windows, use the following commands:

```
$ cd ~/baxter_ws
$ ./baxter.sh
```

Be sure that Baxter is enabled and untucked for the examples using the real Baxter:

```
$ rosrun baxter_tools enable_robot.py -e
$ rosrun baxter_tools tuck_arms.py -u
```

If there are issues with Baxter's hardware, software, or network, refer to the general Baxter troubleshooting website at `http://sdk.rethinkrobotics.com/wiki/Troubleshooting`.

If there are problems with the workstation computer setup, refer to the Rethink wiki site at `http://sdk.rethinkrobotics.com/wiki/Workstation_Setup`.

Controlling a real Baxter

The `baxter_examples` programs described in the subsections within the *Launching Baxter Simulator in Gazebo* section also work on a real Baxter robot. Some additional arm control programs that work on a real Baxter but not on Baxter Simulator are described in the following sections.

Commanding joint position waypoints

This program is another example of joint position control for Baxter's arms. Baxter's arm is moved using the Zero-G mode to freely configure the arm's joints to the desired position. When the desired position is attained, the corresponding navigator button on the arm is pressed to record the waypoint position. This `baxter_examples` program is executed with the following command, specifying either right or left for the arm that is to be moved:

```
$ rosrun baxter_examples joint_position_waypoints.py -l <right or left>
```

The output should be as follows:

```
...
Press Navigator 'OK/Wheel' button to record a new joint position
waypoint.
Press Navigator 'Rethink' button when finished recording waypoints to
begin playback.
...
```

On the navigator, the center button (scroll wheel) is the control used to record all seven joint angles of the specified arm's current position. Waypoints can be recorded repeatedly until the lower button (the button with the Rethink icon) is pressed. This Rethink button activates playback mode, when the arm will begin going back to the waypoint positions in the order that they were recorded. This playback mode will continue to loop through the waypoints until the *Ctrl* + *C* or *Ctrl* + *Z* key combination is pressed. Parameters for speed and accuracy can be passed with the `joint_position_waypoints.py` command. Refer to Rethink's wiki site at `http://sdk.rethinkrobotics.com/wiki/Joint_Position_Waypoints_Example`.

Commanding joint torque springs

This `baxter_examples` program provides an example of Baxter's joint torque control. This program moves the arms into a neutral position, then applies joint torques at 1000 Hz to create an illusion of virtual springs. The program calculates and applies linear torques to any offset from the arm's starting position. When the arm is moved, these joint torques will return the arm to the starting position. Depending on the stiffness and damping applied to the joints, oscillation of the joints will occur.

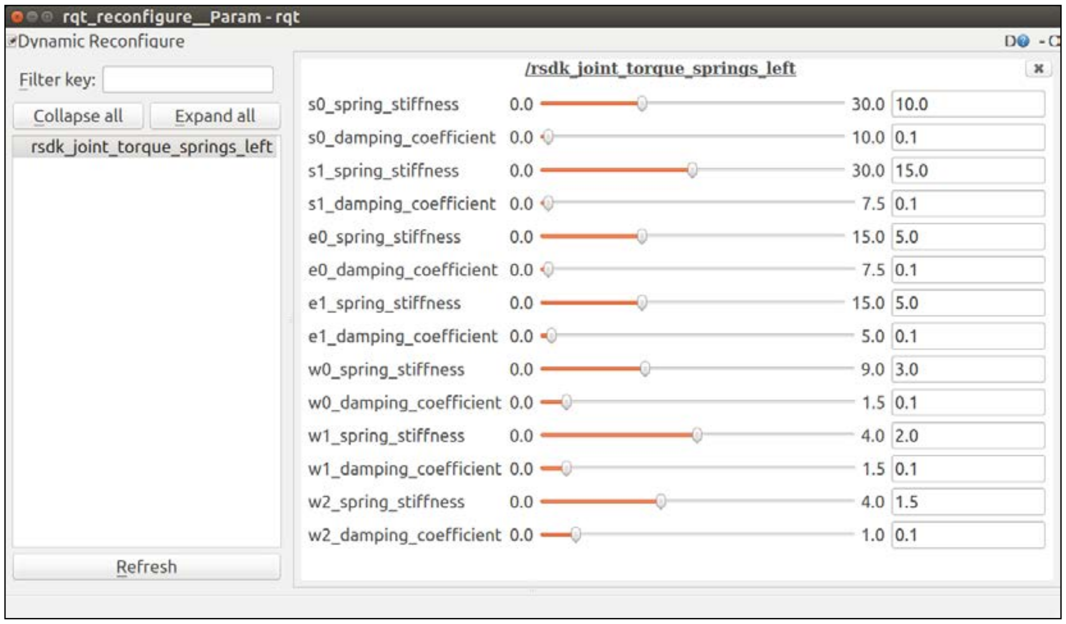
This joint torque springs program is executed with the following command, specifying right or left for the arm that is to be manipulated:

```
$ rosrun baxter_examples joint_torque_springs.py -l <right or left>
```

The joint torques are configurable using the rqt reconfigure tool. To adjust the torque settings, type the following command in a new terminal:

```
$ rosrun rqt_reconfigure rqt_reconfigure
```

The following screenshot shows the `rqt_reconfigure` screen for `joint_torque_springs.py` for the left arm:


rqt_reconfigure__Param - rqt
Dynamic Reconfigure
Filter key:
Collapse all
Expand all
rsdk_joint_torque_springs_left
Refresh
/rsdk_joint_torque_springs_left

s0_spring_stiffness	0.0	30.0	10.0
s0_damping_coefficient	0.0	10.0	0.1
s1_spring_stiffness	0.0	30.0	15.0
s1_damping_coefficient	0.0	7.5	0.1
e0_spring_stiffness	0.0	15.0	5.0
e0_damping_coefficient	0.0	7.5	0.1
e1_spring_stiffness	0.0	15.0	5.0
e1_damping_coefficient	0.0	5.0	0.1
w0_spring_stiffness	0.0	9.0	3.0
w0_damping_coefficient	0.0	1.5	0.1
w1_spring_stiffness	0.0	4.0	2.0
w1_damping_coefficient	0.0	1.5	0.1
w2_spring_stiffness	0.0	4.0	1.5
w2_damping_coefficient	0.0	1.0	0.1

rqt reconfigure joint torque springs

Select `rsdk_joint_torque_springs` from the left panel to view the control menu. The spring stiffness and damping coefficient can be varied for each joint of the arm specified.

Be careful when changing these values!

If you experiment with Baxter's control system, you should record the initial values and reset the values to the originals after changing them.

Demonstrating joint velocity

Rethink provides a simple `baxter_examples` program to demonstrate the joint velocity control mode for Baxter's arms. This program begins by moving the arms into a neutral position. The joint velocity puppet program simply mirrors the movement of Baxter's arm when the other arm is moved in Zero-G mode. This `baxter_examples` program is executed with the following command, specifying either right or left for the arm that is to be moved:

```
$ rosrun baxter_examples joint_velocity_puppet.py -l <right or left>
```

A parameter for amplitude can be passed with this command to change the velocity applied to the puppet response. For more information on this command, refer to Rethink's wiki site at `http://sdk.rethinkrobotics.com/wiki/Puppet_Example`.

Additional examples

The `baxter_examples` programs also include programs for gripper control, camera control, and analog and digital input/output control. Refer to the Rethink wiki Baxter examples program site to get details on these programs: `http://sdk.rethinkrobotics.com/wiki/Examples`.

In addition, Rethink offers a series of video tutorials that provide information on everything from setting up Baxter to running the example programs. Referring to these videos may provide some help if you have problems with executing the example programs (`http://sdk.rethinkrobotics.com/wiki/Video_Tutorials`).

Visual servoing and grasping

One of the greatest features of a real Baxter is the capability to detect and grasp an object. This capability is called **visual servoing control**. Baxter's cuff camera and gripper combination makes this a determined objective.

Baxter's cuff camera provides 2D camera images that can be processed by computer vision software such as OpenCV. OpenCV provides a vast library of functions for processing real-time image data. Computer vision algorithms for thresholding, shape recognition, feature detection, edge detection, and many more are useful for 2D (and 3D) perception.

An example of visual servoing from the Rethink website is available at `http://sdk.rethinkrobotics.com/wiki/Worked_Example_Visual_Servoing`.

This is a basic implementation linking object detection with the autonomous movement of the arm to grasp the object. This project is a good example of the technique described previously. Unfortunately, this example works with ROS Hydro and uses OpenCV functions that have been deprecated.

Only using Baxter's 2D cameras limits the accuracy of grasping objects, making the depth of objects in the entire scene hard to determine. Typically, programs such as the one previously mentioned require a setup phase, in which an infrared sensor measurement to the table surface is required. An alternative is to use an external 3D camera such as the Kinect, ASUS, PrimeSense, or RealSense to detect the depth of objects and match that information with the RGB camera data. This requires calibrating the two image data streams. The Open Source Robotics Foundation has demo software for both 2D perception and manipulation and 3D perception at `https://github.com/osrf/baxter_demos`.

The calculation of inverse kinematics to move the gripper to the desired location is also crucial to this process.

Inverse kinematics

Using forward kinematics, we can determine the position of the gripper at any time. The inverse kinematic problem is to place the gripper at a desired location and orientation. This requires the calculation of the joint angles, then sending Baxter the seven joint angles and commanding the arm to move.

Rethink Robotics provides an **Inverse Kinematic** (**IK**) example that sets a specific endpoint position and orientation in the script and solves the required joint angles. The example and the Python script are described on these websites:

- `http://sdk.rethinkrobotics.com/wiki/IK_Service_Example`
- `http://sdk.rethinkrobotics.com/wiki/IK_Service_-_Code_Walkthrough`

To run the IK example to find the joint angles of the left limb (arm) for the fixed position and orientation in the Python script, type this command:

```
$ rosrun baxter_examples ik_service_client.py -l left
```

The pose of the left end-effector taken from the `ik_service_client.py` script is as follows:

```
'left': PoseStamped(
            header=hdr,
            pose=Pose(
                position=Point(
                    x=0.657579481614,
                    y=0.851981417433,
                    z=0.0388352386502,
                ),
                orientation=Quaternion(
                    x=-0.366894936773,
                    y=0.885980397775,
                    z=0.108155782462,
                    w=0.262162481772,
                ),
            ),
        ),
```

Executing this yields the joint angles to move Baxter's left arm to the pose defined in the script. These angles would be used to move Baxter's arm to this pose from an arbitrary position, as shown by the example in the next section.

The script also sets an initial pose for the right arm. The endpoint position and orientation of the right arm can be found using the same command but with the `right` option. See the code to find the specific pose assigned for this.

Moving Baxter's arms with IK

To demonstrate the IK service example using the real Baxter's left arm, we will perform the following steps:

1. Power up Baxter and untuck both arms. This is the home position for the arms.
2. Record the endpoint state in the position and orientation of the left arm.
3. Move Baxter's left arm to an arbitrary position.
4. Modify the `ik_service_client.py` script in the `baxter_examples` package by entering the position and orientation of the untucked left arm and save the file under a different name in `catkin_workspace`.

5. Execute the script to get the joint angles of the left arm.
6. Type the angles into a modified `home_arms.py` script and execute it.
7. Record the new endpoint positions and orientations and compare them to the original values recorded in step 2.

First, execute the script to move Baxter's arms to the untucked position:

```
$ cd baxter_ws
$ ./baxter.sh
$ rosrun baxter_tools tuck_arms.py -u
```

Then, display the left-arm endpoint pose position and orientation and record the values to two decimal places with the following command:

```
$ rostopic echo /robot/limb/left/endpoint_state/pose -n1 -w4
```

Our output for the pose of the left arm is as follows:

```
position:

  x: 0.57
  y: 0.18
  z: 0.10
orientation:
  x: 0.13
  y: 0.99
  z: 0.00
  w: 0.02
```

The endpoint of Baxter should be out about 0.57 meters in `x`, 0.18 meters to the left of Baxter's vertical centerline in `y`, and about 0.10 meters up from the base in `z`. Next, by hand, move Baxter's arms arbitrarily so that you can test the IK server routine.

To modify the script `ik_service_client.py`, first use the following command:

```
$ roscd baxter_examples/scripts
```

Find the Python script to modify in this directory. To use the IK service with the endpoints of the untucked position and get angles for the left limb, put the `x`, `y`, `z` values and the orientation into the script `ik_service_client.py` file by editing the script with the values shown here, or use the values you obtained:

```
poses = {
        'left': PoseStamped(
            header=hdr,
            pose=Pose(
                position=Point(
                    x=0.57,
                    y=0.18,
                    z=0.10,
                ),
                orientation=Quaternion(
                    x=0.13,
                    y=0.99,
                    z=0.00,
                    w=0.02,
                ),
            ),
        ),
```

After editing `ik_service_client.py`, you should rename the file. Our new file was named `ik_home_arms_ch6RealBaxter.py`. To make it executable, type the following command:

```
$ chmod +x ik_home_arms_ch6RealBaxter.py
```

To run this script to find the joint angles of the left arm that would move Baxter's arm to the specific endpoint position, type this:

```
$ python ik_home_arms_ch6RealBaxter.py -l left
```

The output should be similar to the following:

```
SUCCESS - Valid Joint Solution Found from Seed Type: Current Joint Angles


IK Joint Solution:
{'left_w0': -1.8582664616409326, 'left_w1': -1.460468102595922,
'left_w2': 2.2756459061545797, 'left_e0': -1.6081637990992477, 'left_
e1': 1.9645288022495901, 'left_s0': 0.044896665837355125, 'left_s1':
-0.3326492980686455}
------------------
```

[Your results will probably be different, but the end position of Baxter's arms should be the same as in this Python example, which moves the arms to the home (untucked) position.]

Use the resulting angles to move Baxter's arms using the edited Python script, `home_arms.py`. Change the values of the left arm joints and save the file with a new name. We used the `MoveLeftArmToHome.py` filename. Make the file executable using this command:

```
$ chmod +x MoveLeftArmToHome.py
```

Execute the new script and watch Baxter's left endpoint return to the desired position, if all goes well:

```
$ python MoveLeftArmToHome.py
```

Finally, display the left arm endpoint pose position and orientation and record the values to compare them to the original values for position and orientation:

```
$ rostopic echo /robot/limb/left/endpoint_state/pose -n1 -w4
```

After Baxter's arm moved, our values were fairly close to the originals:

```
position:
  x: 0.57
  y: 0.18
  z: 0.09
orientation:
  x: 0.12
  y: 0.99
  z: 0.00
  w: 0.02
```

Using a state machine to perform YMCA

Finite-state machines are powerful mechanisms for controlling the behavior of a system, especially robotic systems. ROS has implemented a state machine structure and behaviors in a Python-based library called **SMACH**. The SMACH library is independent of ROS and can be used with any Python project. SMACH provides an architecture for implementing hierarchical tasks and mechanisms to define transitions between these tasks. The advantages of using SMACH for a system include the following:

- Rapid prototyping of a state machine for testing and use
- Defining complex behaviors using a clear, straightforward method for design, maintenance, and debugging
- Introspection of the state machine, its transitions, and data flow using SMACH tools

For a complete set of documentation and tutorials on SMACH, examine these websites:

- `http://wiki.ros.org/smach`
- `http://wiki.ros.org/smach/Tutorials`

Some basic rules for implementing state machines on a robot are as follows:

- A robot can be in one—and only one—state at a time.
- A finite number of states must be identified.
- The state that a robot transitions to, will depend on the state that just completed. These behaviors are encapsulated in the states to which they correspond.
- Transitions between states are specified by the structure of the state machine.
- All possible outcomes of a state should be identified and corresponding behaviors should address those outcomes.
- States that only have one transition condition cannot fail and only have one outcome.

To underscore the usefulness of the SMACH package, we devised a simple and fun example for Baxter, which has been implemented by Mikal Cristen, a recent UHCL graduate. Because the UHCL Baxter is such a main attraction on our campus, this project was to endow the robot with an entertainment skill, specifically, dancing to YMCA.

This state machine has five states corresponding to Baxter's arm poses for each letter: Y, M, C, A, and a fifth state for a neutral pose. When one pose of the arms completes, the state will successfully complete and the next state will begin. The code for this state machine is implemented in the `YMCAStateMach.py` code that follows and will be described in subsequent paragraphs.

The code for `YMCAStateMach.py` and `MoveControl.py` can be found in the `Chapter 6` folder of the Packt GitHub website for this book or at the website `https://github.com/FairchildC/ROS-Robotics-By-Example-2nd-Edition`

SMACH compels state machines to be implemented using Python procedures to provide flexibility in their implementation. Notice in the following code, the ROS convention for state machines is that the state names are identified in ALL_CAPS and the transition names are in lowercase:

```
#!/usr/bin/env python

import rospy
from smach import State,StateMachine

from time import sleep
from MoveControl import Baxter_Arms

class Y(State):
  def __init__(self):
    State.__init__(self, outcomes=['success'])

    self.letter_y = {
        'letter': {
           'left':  [0.0, -1.0, 0.0, 0.0,  0.0, 0.0, 0.0],
           'right': [0.0, -1.0, 0.0, 0.0,  0.0, 0.0, 0.0]
                       } }
             #DoF Key [s0,s1,e0,e1,w0,w1,w2]

  def execute(self, userdata):
    rospy.loginfo('Give me a Y!')
    barms.supervised_move(self.letter_y)
    sleep(2)
    return 'success'

class M(State):
  def __init__(self):
    State.__init__(self, outcomes=['success'])

    self.letter_m = {
        'letter': {
            'left': [0.0, -1.50, 1.0, -0.052, 3.0, 2.094, 0.0],
            'right':[0.0, -1.50, -1.0, -0.052, -3.0, 2.094, 0.0]
                       } }
               #DoF Key [s0,s1,e0,e1,w0,w1,w2]

  def execute(self, userdata):
    rospy.loginfo('Give me a M!')
    barms.supervised_move(self.letter_m)
    sleep(2)
    return 'success'

class C(State):
  def __init__(self):
    State.__init__(self, outcomes=['success'])

    self.letter_c = {
        'letter': {
            'left': [0.80, 0.0, 0.0, -0.052,  3.0, 1.50, 0.0],
            'right':[0.0, -1.50, -1.0, -0.052, -3.0, 1.0, 0.0]
                       } }
            #DoF Key [s0,s1,e0,e1,w0,w1,w2]

  def execute(self, userdata):
    rospy.loginfo('Give me a C!')
    barms.supervised_move(self.letter_c)
    sleep(2)
    return 'success'

class A(State):
  def __init__(self):
    State.__init__(self, outcomes=['success'])

    self.letter_a = {
        'letter': {
            'left': [0.50, -1.0, -3.0, 1.0, 0.0, 0.0, 0.0],
            'right':[-0.50, -1.0, 3.0, 1.0,  0.0, 0.0, 0.0]
                       } }
```

```
                #DoF Key [s0,s1,e0,e1,w0,w1,w2]

      def execute(self, userdata):
        rospy.loginfo('Give me an A!')
        barms.supervised_move(self.letter_a)
        sleep(2)
        return 'success'

    class Zero(State):
      def __init__(self):
        State.__init__(self, outcomes=['success'])

        self.zero = {
            'letter': {
                'left': [0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00],
                'right':[0.00, 0.00, 0.00, 0.00,  0.00, 0.00, 0.00]
                           } }
                             #DoF Key [s0,s1,e0,e1,w0,w1,w2]

      def execute(self, userdata):
        rospy.loginfo('Ta-da')
        barms.supervised_move(self.zero)
        sleep(2)
        return 'success'

    if __name__ == '__main__':

      barms = Baxter_Arms()
      rospy.on_shutdown(barms.clean_shutdown)

      sm = StateMachine(outcomes=['success'])
      with sm:
        StateMachine.add('Y', Y(), transitions={'success':'M'})
        StateMachine.add('M', M(), transitions={'success':'C'})
        StateMachine.add('C', C(), transitions={'success':'A'})
        StateMachine.add('A', A(), transitions={'success':'ZERO'})
        StateMachine.add('ZERO', Zero(),        transitions={'success':'s
    uccess'})

      sm.execute()
```

This Python script imports the following packages:

- `rospy`: This ROS package is used for information messages and to control Baxter's arms in the event a system shutdown occurs.
- `smach`: This ROS package imports the `State` and `StateMachine` classes and their methods.
- `time`: This Python package imports the `sleep` function.
- `MoveControl`: This package imports the `Baxter_Arms` class and the methods for interacting with Baxter's arms. Look for this Python script in the `Chapter06` folder of the Packt GitHub website for this book.

Next, classes are defined for each of the states of the state machine: `Y`, `M`, `C`, `A`, and `Zero`. Each of these classes creates a new, initialized instance of the SMACH `State` class, identifying all of the possible outcomes for that state. For these states, `success` is the only outcome. Next, each of Baxter's 14 arm joints are assigned a value (in radians). These values are assigned in the specific order they are expected (ultimately) by the `baxter_interface` package. This order, as indicated in the code comment, is S0, S1, E0, E1, W0, W1, W2.

Each of the states also implements an execute method, where the actual work is done. In each of these state execute methods, the `supervised_move` function for moving Baxter's arms is called passing the argument with all of the joint angles for Baxter's arms. After the function is called, the process sleeps for 2 seconds to allow all of Baxter's arm movements to stop. When the state finishes, the `success` flag is returned as an outcome to the calling function.

The main program of `YMCAStateMach.py` creates an instance of the `Baxter_Arms` class and initializes the attributes for the instance. An instance of a `StateMachine` class is also created and a list of possible outcomes is passed as an argument. The empty `StateMachine` (`sm`) instance is opened and populated with the different states we defined. Each state is added with the add function. For example, the first `StateMachine.add` function adds the state `Y`, with an instance of the class `Y()` and, on completion with a successful outcome, the `StateMachine` class will transition to state `M`. Similarly, the state `M` is added with an instance of the class `M()` and, with a successful outcome, will transition to state `C`. The states `C` and `A` are added in a similar manner, as is the last state `ZERO`. Upon completion, the state `ZERO` will return the successful outcome for `StateMachine`.

Be sure that the `MoveControl.py` and `YMCAStateMach.py` scripts are in your directory and have execute permissions. Real Baxter or Baxter Simulator should be running and enabled (preferably in an untucked pose). Then, run the state machine with the following command:

```
$ python YMCAStateMach.py
```

You should see Baxter's arms transition to a Y pose, then an M pose, then a C pose, then an A pose, and end in a pose similar to the `arms_to_zero_angles.py` pose. The `INFO` messages to the terminal window should be similar to the following:

```
[  INFO ] : State machine starting in initial state 'Y' with userdata:
[]
[INFO] [1502141862.054272]: Give me a Y!
[INFO] [1502141862.054742]: Movement in progress.
[INFO ] : State machine transitioning 'Y':'success'-->'M'
[INFO] [1502141867.569880]: Give me a M!
[INFO] [1502141867.570851]: Movement in progress.
[INFO] [1502141872.782220]: Robot Disabled
[  INFO ] : State machine transitioning 'M':'success'-->'C'
[INFO] [1502141874.784842]: Give me a C!
[INFO] [1502141874.785353]: Movement in progress.
[INFO] [1502141876.906888]: Robot Disabled
[INFO ] : State machine transitioning 'C':'success'-->'A'
[INFO] [1502141878.909487]: Give me an A!
[INFO] [1502141878.910383]: Movement in progress.
[INFO ] : State machine transitioning 'A':'success'-->'ZERO'
[INFO] [1502141889.262132]: Ta-da
[INFO] [1502141889.262614]: Movement in progress.
[ INFO ] : State machine terminating 'ZERO':'success':'success'
```

Summary

In this chapter, we described a real and popular robot called Baxter, manufactured by Rethink Robotics Corporation. Many of the details of the robot can be discovered using Baxter Simulator, which displays a simulated Baxter using the Gazebo program. Since Baxter has two movable arms, much of the chapter describes the arms and control of them.

The chapter started with a description of Baxter in both the research and the manufacturing versions. Baxter's arms and sensors and the control modes for the arms were described.

After downloading the Baxter Simulator software, the simulator was used to demonstrate various examples of controlling Baxter with Python scripts supplied by Rethink Robotics. Baxter can be controlled using ROS commands, a keyboard, or a joystick. We have also included several Python scripts that will make the control of Baxter easier if joint angles are specified for the movement of the arms.

The ROS frame transform package tf was used to show the relationship between the coordinate frames of Baxter's base and other elements of the arms. The view in rviz displays these frames.

MoveIt! is another package that works with a simulated Baxter, as well as with the real robot. MoveIt! was explained and the method of planning and executing arm trajectories even with obstacles in Baxter's path was discussed.

Finally, we were introduced to the real Baxter by explaining the setup procedure for communication between a workstation and Baxter. Various examples showing control of the real Baxter were also discussed. A state machine using the ROS SMACH package was implemented to move Baxter's arms into different positions.

In the next chapter, a ROS view of quadrotors will be described using both simulated and real air vehicles. The simulated air vehicle is a generalized representation of a quadrotor and is great for learning to control the craft before you decide to try flying a real one.

In addition to this action packed chapter for Baxter, *Chapter 10*, *Controlling Baxter with MATLAB©* presents the **Robotics System Toolbox**. This MATLAB toolbox allows ROS commands to be used with MATLAB scripts to control Baxter.