

## The Baxter Simulator

Wyatt Newman

October, 2015

Our minimal robot illustrated the features of Gazebo, Rviz, a URDF file and joint controllers. Armed with this knowledge, we can begin to use a more realistically complex robot simulation. The example we will use is the Baxter simulator from Rethink Robotics, Inc.

On our systems, you will find a ROS subdirectory called “rethink.” This directory contains 4 repositories created by Rethink Robotics. Three of these are useful for interfacing to either the Baxter simulator or to the physical Baxter robot. The fourth repository, “baxter\_simulator”, is currently a private repository. Direct access to this repository can be obtained on request from Rethink Robotics. It is anticipated that this restriction will go away before the end of 2015.

A second directory we will use is “cwrubaxter.” This directory contains a package called “cwrubaxter\_sim”, which includes a modified version of the robot model, “cwrubaxter.urdf.” The modification is the addition of a Kinect sensor, beginning with the lines:

```
<gazebo reference="kinect_link">  
  <sensor type="depth" name="openni_camera_camera">  
and including the sensor library plug-in:  
  <plugin name="camera_controller" filename="libgazebo_ros_openni_kinect.so">
```

With these additions, Baxter can be launched with a Kinect sensor. As with our minimal robot example, we will also need to start a static-transform publisher to broadcast the relationship between the sensor frame and the Kinect enclosure link. This publisher will be started within the Baxter simulator launch file.

### Starting the Baxter simulator:

When we start up the Baxter gazebo simulation, we will not need to start a roscore. This will be performed automatically as part of the Gazebo launch. To start up the simulator, enter:

```
roslaunch cwrubaxter_sim baxter_world.launch
```

The file “baxter\_world.launch” starts up a default Gazebo “empty world”, it loads the robot model into the parameter server (thus making it available to Rviz), and it starts up the Baxter simulator with specified initial joint angles.

The default “empty world” Gazebo description is overridden with the argument:

```
<arg name="world_name" value="$(find cwrubaxter_sim)/baxter.world"/>
```

The file baxter.world makes the default world somewhat more interesting by introducing additional models, including a cafe table and a beer can. Specifications in this file describe where the models can be found, and where they should be placed in Gazebo, as follows:

(from baxter.world:)

```
<include>  
  <uri>model://cafe_table</uri>  
  <pose>0.756237 -0.288636 0 0 -0 0</pose>
```

```

</include>
<include>
  <uri>model://beer</uri>
  <pose>0.69112 0 0.775 0 0 0</pose>
</include>

<param name="robot_description"
  command="cat '$(find cwru_baxter_sim)/cwru_baxter.urdf'" />

```

The file `cwru_baxter.urdf` is a modified version of Rethink's `baxter.urdf`, which includes a Kinect link, a fixed joint securing the Kinect link to the robot's (movable) head, and a gazebo plugin:

```

<plugin name="camera_controller" filename="libgazebo_ros_openni_kinect.so">

```

to emulate a Kinect sensor.

The `baxter_world` launch file also includes the following launch file:

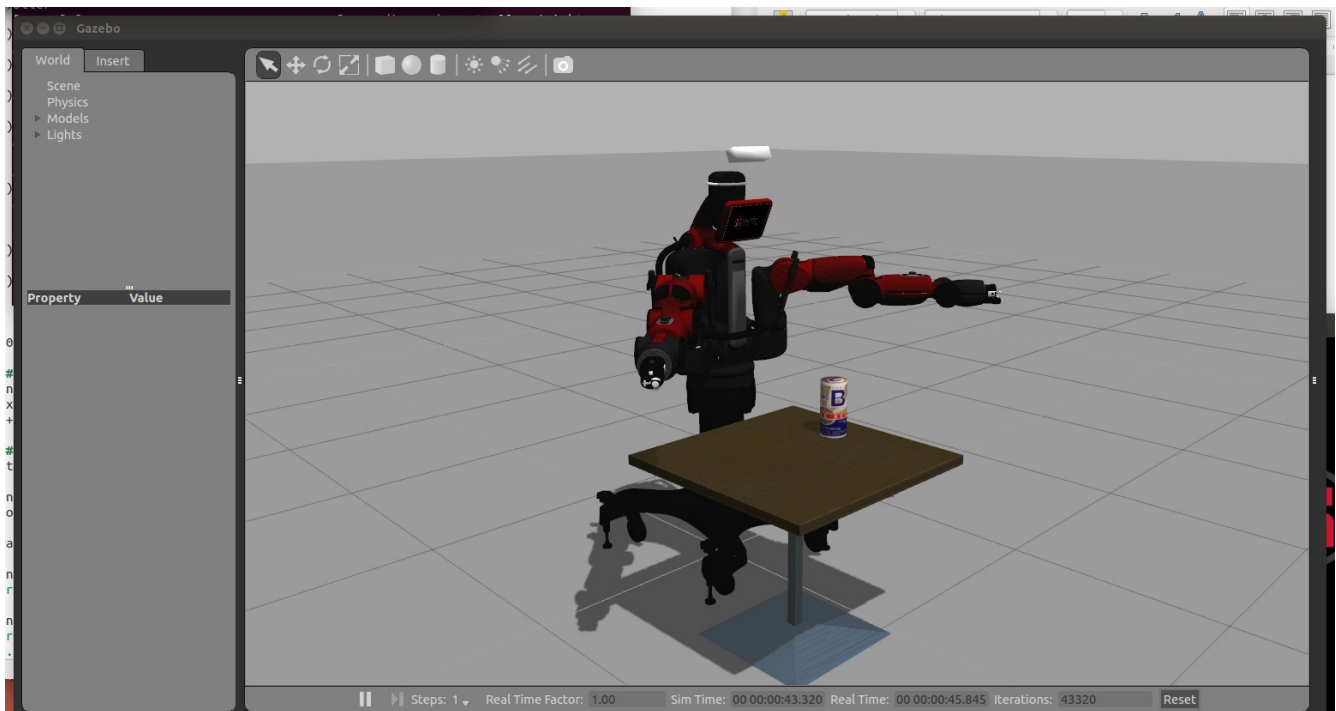
```

<!-- ros_control baxter launch file -->
<include file="$(find baxter_sim_hardware)/launch/baxter_sdk_control.launch" />

```

This launch file is responsible for bringing up Rethink code for a set of joint controllers. We will not need a version of “`minimal_joint_controller`” for Baxter (neither in simulation nor with the physical robot), since the joint controllers are provided as part of the simulator and as part of the robot controller.

Once the Baxter simulator is launched, it will appear in Gazebo as below:



### Getting Baxter joint and sensor states:

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, 16 “kinect” 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:

```
roscall list
```

we see that a robot\_state\_publisher is running. We thus will not need to start this node for Rviz visualization of joint poses. However, the robot\_state\_publisher needs access to a robot model and to robot joint states. We can see from a

```
roscall list
```

that our “robot\_description” is loaded on the parameter server. We can further examine the topic “/robot/joint\_states” with:

```
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:

```
roscall 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 below.

```
wyatt@AtlasUSB: ~
2242692769346e-06, 6.759778514034962e-08, 3.340446466643918e-06, -1.1454491156985906e-07, -2.1906271925719808e-07, -
1.1631384493633395e-08]
effort: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
---
t header:
  seq: 7984
  stamp:
    secs: 161
    nsecs: 176000000
  frame_id: ''
name: ['head_pan', 'left_e0', 'left_e1', 'left_s0', 'left_s1', 'left_w0', 'left_w1', 'left_w2', 'right_e0', 'right_e
1', 'right_s0', 'right_s1', 'right_w0', 'right_w1', 'right_w2']
position: [2.8733239787470666e-08, 1.2271041205380584e-06, 1.1997276662789602e-05, -1.697901389263734e-05, -7.591866
659684143e-05, -1.3453344380742749e-05, 3.012488263820501e-05, 7.193083240153442e-07, 5.559706441005119e-06, -0.0003
152983937670939, 3.763943322354635e-06, 0.0005241781615801244, -1.836458931059326e-05, -3.4931421713402244e-05, -1.3
034191512772964e-06]
velocity: [3.685317604671881e-08, -5.331257792984813e-09, 6.47819496962406e-08, -6.687495304331007e-08, -4.827398430
309769e-07, -8.355424173267948e-08, 1.9199202826708168e-07, -3.916104097719703e-11, 2.3763521777951177e-08, -1.93822
51325721195e-06, 6.759750591747239e-08, 3.340448051512521e-06, -1.145451197467815e-07, -2.1906273767324802e-07, -1.1
631241802400565e-08]
effort: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
---
^Cwyatt@AtlasUSB:~$
```

From our echo of the joint\_states topic, we see the following joint names:

```
name: ['head_pan', 'left_e0', 'left_e1', 'left_s0', 'left_s1', 'left_w0', 'left_w1', 'left_w2', 'right_e0',
'right_e1', 'right_s0', 'right_s1', 'right_w0', 'right_w1', 'right_w2']
```

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 tip. ROS, however, is generally indifferent to the order of joints, either for reporting states or accepting commands; the order used in the “name” list is the order ROS will use, and this may change from iteration to iteration. For joint\_states reporting, we can assume that the same order will be used every 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 camera topics, kinect topics, and laserscan topics will all be 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.

**Controlling the Baxter Robot:** The Baxter simulator (and actual Baxter robot) subscribes to the topic: /robot/limb/right/joint\_command

which carries messages of type baxter\_core\_msgs/JointCommand. We can examine this message type with:

```
rosmmsg 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 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.

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);
```

Where the publisher is defined as:

```
joint_cmd_pub_right =
nh.advertise<baxter_core_msgs::JointCommand>("/robot/limb/right/joint_command", 1);
```

Joint-angle commands should be published as a rapidly-updated stream of incrementally changing joint values to produce a smooth trajectory. However, the joint controller will only respond to a limited update rate of commands. Likely, the joint controller accepts commands at the same rate that joint\_states messages are published, i.e. 50Hz.

The topic /tf publishes large number of link-to-link transforms. All link-to-link relationships described by a connecting joint are updated rapidly. Some of these can be viewed by running:  
rostopic echo tf

With the Baxter simulator launch, we have provided all of the necessary information (robot model on parameter server, publication of joint\_states and execution of robot\_state\_publisher) for Rviz to correctly display all robot-model links. Starting up:

```
roslaunch rviz rviz
```

we can add a “robot model” item to view Baxter in Rviz. An example view is shown below. In the view below, we have included a “pointCloud2” topic, which subscribes to the kinect/depth/points topic. The resulting display shows what the Kinect sensor would see, given the Gazebo objects present in the virtual world. If the physical Baxter robot were running, and if a Kinect were added to the robot, and if the Kinect driver were streaming its data, and if the Kinect transform relative to the robot's head were being published, Rviz would be able to display the Kinect data similarly, only with reference to the physical rather than the virtual world.

