Introduction to Dynamic Simulation with Gazebo

Wyatt Newman
July, 2015

One of the most important features of ROS is the ability to switch running code between a simulated robot and a real robot.

For robot motion control, a ROS interface is defined at the joint level, which is the lowest common denominator for robots. With respect to robot joints, the ROS interface between a robot to user code in ROS consists primarily of joint commands (from ROS to the robot) and feedback of joint states (from the robot to ROS). Some robot companies, including Rethink Robotics, Fetch Robotics and Boston Dynamics, provide such interfaces, making them immediately ROS compatible. Some industrial robots have been retrofit to publish joint states and accept joint commands via ROS topics (see http://rosindustrial.org/). Custom and retrofit robots require the developer to create a “bridge” capable of reading the robot's joint sensors and publishing these values in ROS-compatible messages with sufficiently high sample rate and low latency. Similarly, robot joint commands published on a joint-command topic must be received, interpreted and converted to suitable commands to the hardware. Such bridge nodes provide a ROS interface, making the custom robot compatible with existing ROS software.

It should be noted that low-level controls (e.g. PID servos) typically are not suitable for ROS implementation. Since Linux is not a hard real-time operating system, and since network communications may be insufficiently reliable for demanding real-time control, critical low-level controls should be implemented on dedicated hardware. Streams of position commands from ROS typically are fast enough and smooth enough for a suitable interface to a dedicated joint position controller.

If the robot interface is abstracted such that the robot publishes its joint-sensor values and subscribes to joint-position commands, then the physical robot is indistinguishable from from a corresponding robot simulator. ROS control code can switch between a physical robot and a simulated robot merely by changing topic names or pointing to a different IP address. As such, no changes are required in the ROS code, thus eliminating sources of errors originating from re-porting code from simulator to robot.

When developing ROS code on a robot simulator, the quality of the results depends on the fidelity of the simulator. To realize a useful quality of simulation, one must specify a fair amount of detail modeling the dynamics of the robot. In addition, to simulate realistic interactions with objects in the world, the simulator must consider contact dynamics, including possible collisions between any surface of the robot and the surface of any obect in the environment (e.g., a forearm hitting the edge of a table). To incorporate these dynamic effects, we must augment the robot's URDF model, we must include a solid model of an environment of interest, and we must invoke a physics engine to compute the dynamics resulting from actuator efforts, gravity effects, and forces/torques from contact interactions.

Adding Dynamic Information to the URDF: We formerly saw that we could construct a robot model for visualization in the URDF format. Additional terms can be included to create a dynamic model. A more complete URDF file, “minimal_robot_description.urdf”, is contained in the package “minimal_robot_description.” From our original one-DOF robot model, we specified link2's appearance as:

```
  <visual>
    <origin xyz="0 0 0.5" rpy="0 0 0"/>
    <geometry>
      <cylinder length="1" radius="0.1"/>
    </geometry>
  </visual>
```

Within the same <link\> scope, we can add some mass properties, e.g.:

```
  <inertial>
    <origin xyz="0 0 0.5" rpy="0 0 0"/>
    <mass value="1"/>
    <inertia
      ixx="0.1" ixy="0.0" ixz="0.0"
      iyy="0.1" iyz="0.0"
      izz="0.005"/>
  </inertial>
```

The above mass properties are crudely based on a cylindrical link of mass 1 kg. In general, one should attempt to model each robot link as well as possible to get reasonable estimates of mass, center of mass and mass moments. In this case, the center of mass of the cylinder is specified to be in the center of the cylinder—which is appropriate. The mass of the cylinder is specified to be 1 (kg). The values of “rpy” specify a roll-pitch-yaw orientation corresponding to principal axes of rotation (about the center of mass). Since the cylinder is symmetric, aligning the principal axes with the link's reference frame is correct. The six values ixx through izz specify components of the matrix inertia tensor for this link. Since this 3x3 matrix is symmetric, it is only necessary to specify 6 values. For Ixx and Iyy, the inertia should be approximately $(1/12)m*L^2$. For m=1 and L=1, these terms are assigned the approximate value 0.1 kg-m^2. The rotational inertia about the z axis should be $(1/2)mr^2$, or approximately 0.005 kg-m^2. By symmetry, the off-diagonal elements are zero.

For link1, the inertial terms are not important, since link1 is “glued” to the world frame. However, the simulator still requires specifying values—and these values should be non-zero, to avoid numerical instability. For link1, mass properties are arbitrarily set to:

```
 <inertial>
    <origin xyz="0 0 0.5" rpy="0 0 0"/>
    <mass value="1"/>
    <inertia
      ixx="1.0" ixy="0.0" ixz="0.0"
      iyy="1.0" iyz="0.0"
      izz="1.0"/>
  </inertial>
```

Additional properties can be added to joint1, e.g. to model viscous or Coulomb friction. Further, joint limits and actuator torque limits can be specified. However, the minimal model that we have is adequate for simulating robot dynamics subject to specified joint (actuator) torques and the influence of gravity. To also include the influence of contact forces, we need to specify additional geometric detail corresponding to a “skin” on the movable link(s). This is call the “collision” model. It is used to compute intersections between solids within a world model, which lead to reaction forces/torques at the points of contact.

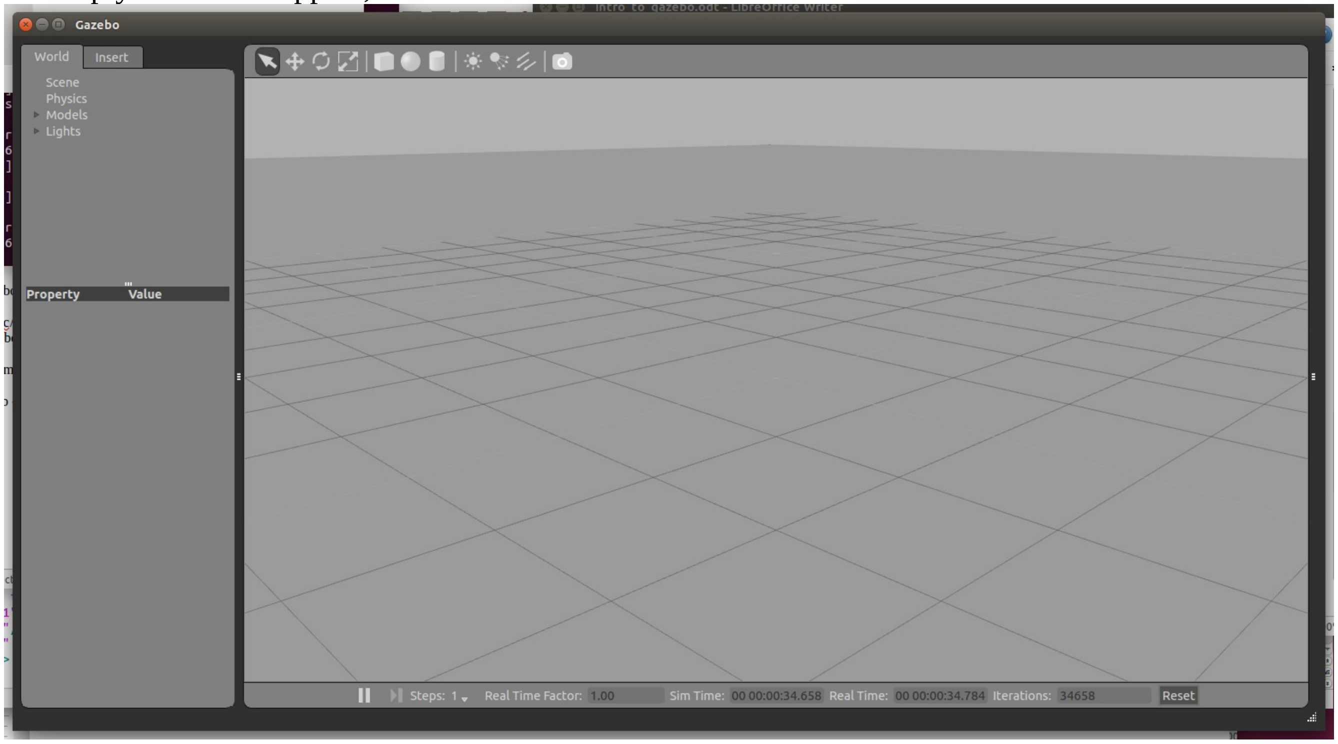
Often, the collision model is identical to the visual model. However, collision-checking can be a slow process, and therefore the collision model should be as sparse as possible. This can be done by reducing the number of triangles in a tessellated surface model, or by creating a primitive collision model based on geometric solids, e.g. rectangular prisms, cylinders or spheres.

Another concern for the collision model is that a collision model that does not offer adequate clearance between the links can result in simulation instability, as the simulator repeatedly determines that the links are colliding with each other. For our crude model, we will set the collision model of link2 to be identical to the visual model of link2—a simple cylinder. However, we will set the collision model of link1 to be a shorter box, thus providing clearance for link2. Since link1 is stationary, fidelity of its collision model is not a concern.

The Gazebo simulator: “Gazebo” is the simulator used with ROS (see http://gazebosim.org/). Gazebo offers options for alternative physics engines, with a default of “ODE” (Open Dynamics Engine—see http://opende.sourceforge.net/). To start up Gazebo, with a roscore running, enter:

```
        rosrun gazebo_ros gazebo
```

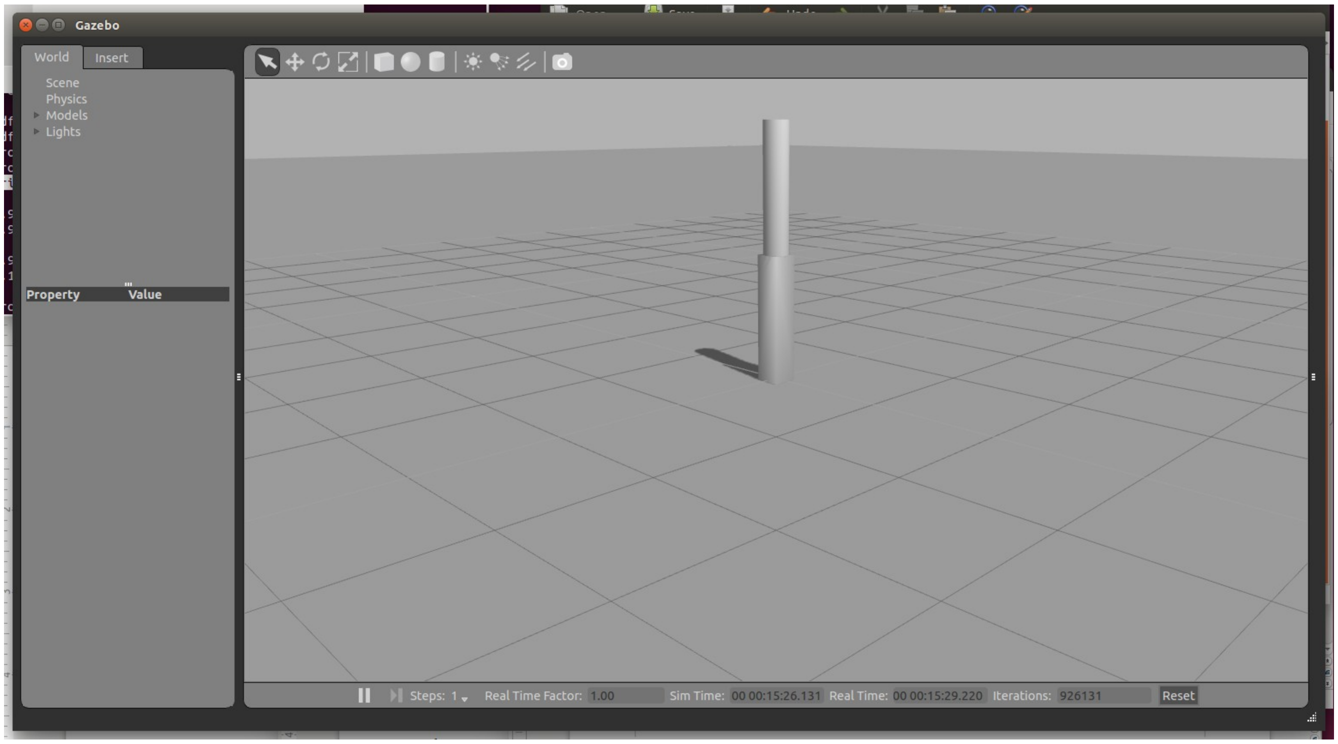
An empty world will appear, as below:



Next, we want to load our robot model. First, navigate to the folder “minimal_robot_description” and enter the following:

```
  rosrun gazebo_ros spawn_model -file minimal_robot_description.urdf -urdf -model one_DOF_robot
```

The Gazebo window then looks looks like this:



Our robot is displayed in its initial position. Since no controller is running, you may see it fall over under the influence of gravity. With the robot in Gazebo, new topics and services exist, as can be seen from the response to: rostopic list and rosservice list.

A minimal joint controller: A minimal joint controller is described here, which interacts with Gazebo and creates a ROS interface (similar to constructing the bridges necessary to interact with a real robot). Please note that the example controller described here normally would not be used. For a physical robot, the PD controller would be contained within dedicated control hardware. Similarly, for Gazebo, there are pre-defined Gazebo "plug-ins" that perform the equivalent of the present joint-controller example. Thus, you would not need to use the minimal_joint_controller package; this is presented only to illustrate the equivalent of what a gazebo controller plug-in performs.

In the source code "minimal_joint_controller.cpp", service clients of both /gazebo/get_joint_properties and /gazebo/apply_joint_effort are instantiated. Corresponding service message types (from gazebo_msgs) are populated with request values.

The service "/gazebo/get_joint_properties" can be invoked with:

```
rosservice call /gazebo/get_joint_properties "joint1"
```

which gives the response:

```
type: 0
damping: []
position: [0.0]
rate: [0.0]
success: True
status_message: GetJointProperties: got properties
```

A service client of this service within "minimal_joint_controller.cpp" uses such calls to repeatedly get the joint position and velocity from the dynamic simulator.

A second service that is useful is: /gazebo/apply_joint_effort. This service uses a "srv" message defined in: …/gazebo_msgs/srv/ApplyJointEffort.srv, which contains the following:

```
# set urdf joint effort
string joint_name          # joint to apply wrench (linear force and torque)
float64 effort             # effort to apply
time start_time            # optional wrench application start time (seconds)
                           # if start_time < current time, start as soon as pos ible
duration duration          # optional duration of wrench application time (seco nds)
                           # if duration < 0, apply wrench continuously without end
                           # if duration = 0, do nothing
                           # if duration < step size, assume step size and
                           #           display warning in status_message
---
bool success               # return true if effort application is successful
string status_message      # comments if available
```

A service client of /gazebo/apply_joint_effort sets the request fields of joint_name to "joint1" and "effort" to a desired joint torque, enabling our controller node to impose joint torques on our simulated robot.

ROS publishers for "jnt_trq", "jnt_vel" and "jnt_pos" are instantiated in our minimal joint controller node, so that these details will be published for interpretation. A subscriber to the topic "pos_cmd" is set up, ready to accept user input for desired joint1 position values.

Additionally, a publisher is set up to publish messages of type: sensor_msgs/JointState to topic "joint_states." This takes the place of the joint_state_publisher, used previously, relaying the joint state values from Gazebo to topic joint_states in the expected format (message type).

The main loop of our controller node does the following:

- get the current joint position and velocity from Gazebo;
- compare the (virtual) joint-sensor value to the commanded joint angle (from the pos_cmd callback);
- compute a torque P-D torque response
- send this effort to Gazebo via the apply_joint_effort service

Running the minimal controller node with:

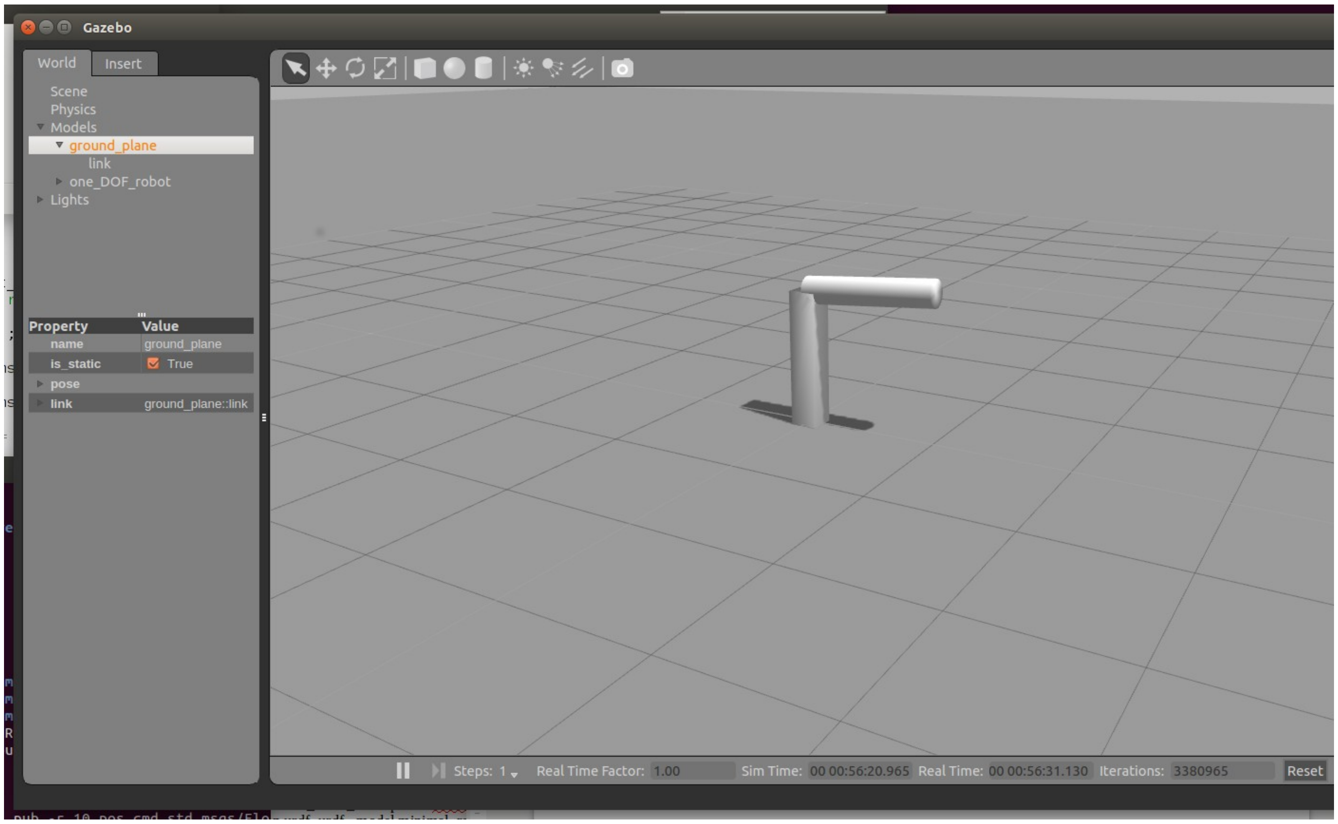
```
rosrun minimal_joint_controller minimal_joint_controller
```

starts the control loop running.

At this point, there is no noticeable effect on Gazebo, since the start-up desired angle is 0, and the robot is already at zero angle. However, we can command a new desired angle manually from the command line (and later, under program control) with the command:

```
rostopic pub pos_cmd std_msgs/Float64 1.0
```

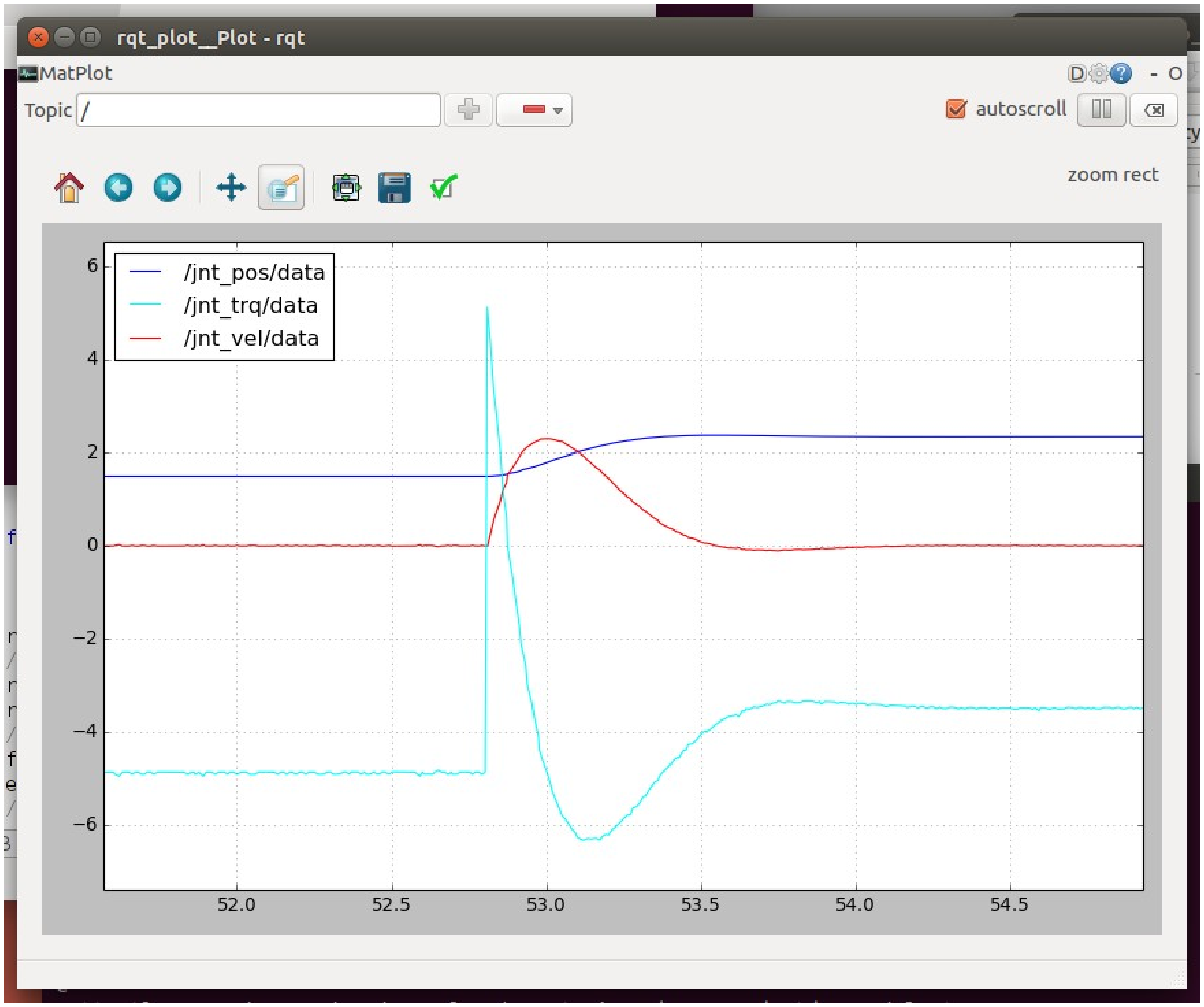
which commands a new joint angle of 1.0 radians. The robot then moves to the position shown below:



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

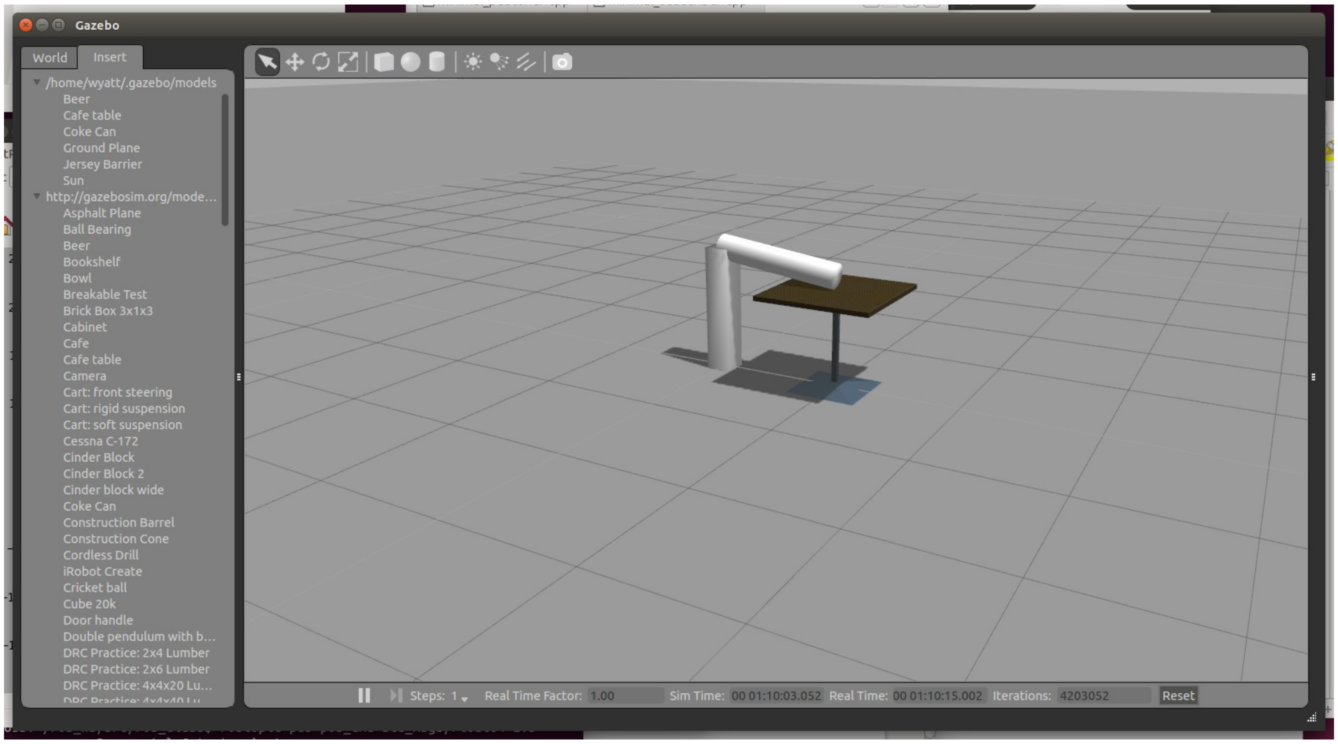
Real Time Factor: 1.00 Sim Time: 00 00:56:20.965 Real Time: 00 00:56:31.130 Iterations: 3380965

We can record the dynamic response of an input command by plotting out the published values of joint torque, velocity and position, using “rqt_plot.” Enter the command “rqt_plot”, then add topics of /jnt_pos/data, /jnt_trq/data and /jnt_vel/data. The plot below shows the transient response, starting from a position command of 1.0, then responding to: rostopic pub pos_cmd std_msgs/Float64 2.0



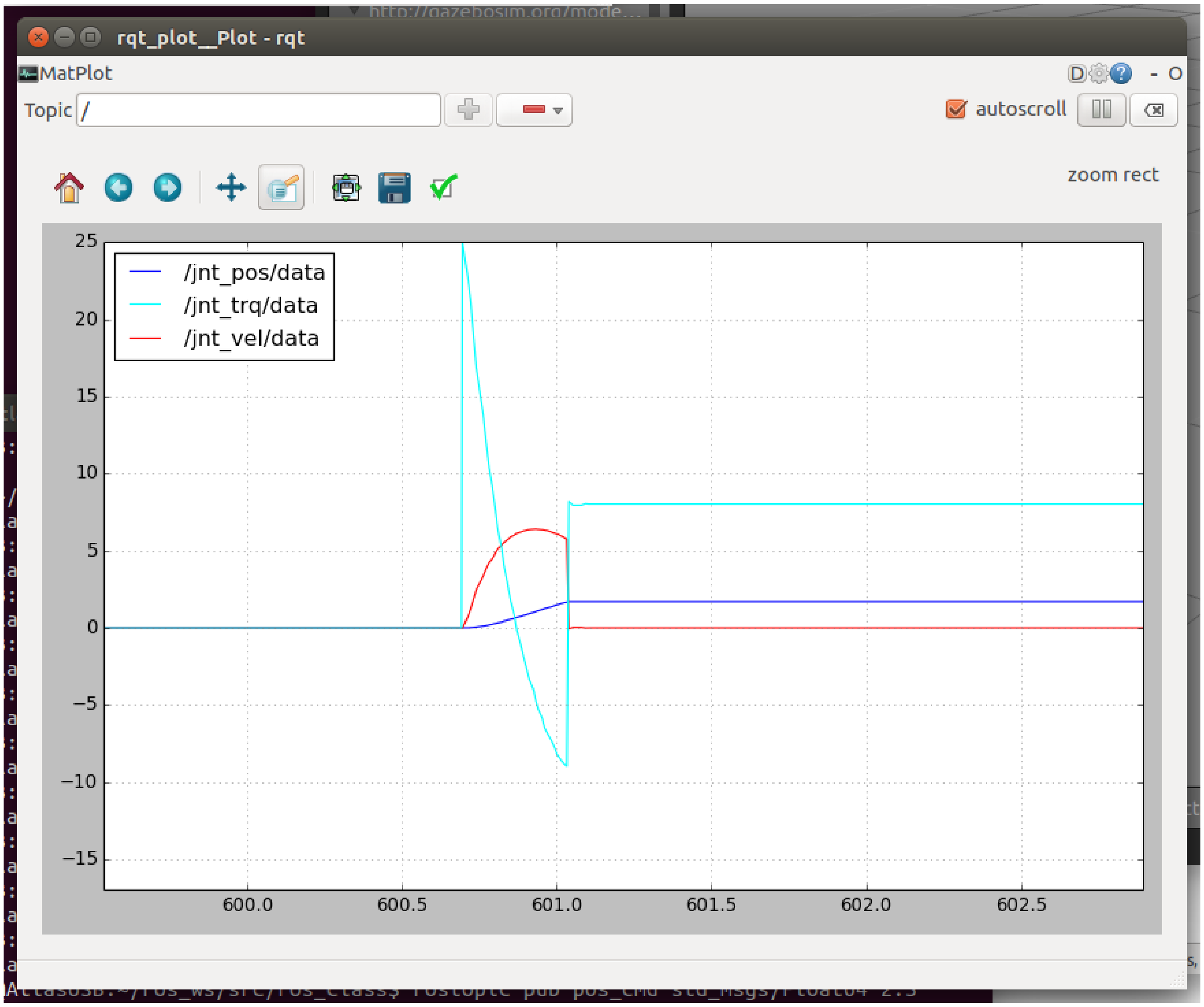
As shown, the initial join angle is larger than the commanded 1.0. This is due to a low feedback proportional gain and theinfluence of gravity, causing droop relative to the desired angle. At approximately t=52.8, the input position command is changed to 2.0 rad, resulting in a transient in joint torque as the link accelerates towards the new goal. The goal of 2.0 is overshot, again due to gravity load. As the link settles, a sustained torque of approximately -3.5N-m is required to hold the link against gravity.

To illustrate the influence of contact dynamics, an additional model is added to Gazebo, as shown below.



Here, a “cafe table” (from a list of pre-defined models) is inserted using the “insert” menu of Gazebo. Gazebo offers the user the capability of moving the table to a desired location, which was chosen to be within reach of the robot.

Next, the robot was commanded to position 0 (straight up), then commanded to position 2.5—which is unreachable with the table in the way. The transient dynamics from this command are shown in the rqt_plot image below.



The collision-model of the link and the collision model of the table are used by Gazebo to detect that contact has occurred. This results in a reaction force from the table to the robot (and from the robot to the table). As a result, the robot does not reach its goal angle, and equilibrium occurs with the robot's joint actuator exerting an effort downwards on the table, while struggling to reach the desired angle of 2.5 rad.

Displaying the robot in Rviz: Since our minimal joint controller publishes the joint angle to the topic joint_states, we do not need to run "joint_state_publisher" to view the robot in Rviz. However, we do still need to load the robot model into the parameter server. Our augmented model in "minimal_robot_description.urdf" can be loaded into the parameter server with the command:

```
roslaunch minimal_robot_description minimal_robot_description.launch
```

Our newer robot description contains inertial information. However, this is ignored by rviz, since only the visualization is rendered by rviz. Only Gazebo needs to know dynamic properties.

Initially in Rviz, link2 does not appear in the display, since Rviz does not know its pose in space. By running:

```
rosrun robot_state_publisher robot_state_publisher
```

Rviz is informed of the full transform for this link. The view in rviz then shows the same pose as the robot in Gazebo. However, the cafe table does not show up in Rviz. This is an important distinction. “Gazebo” is meant to take the place of reality—emulating robot dynamics, including interactions with the environment. Rviz, on the other hand, is intended for sensor display. At this point, the only sensor available to Rviz is the value of the joint1 angle (and the corresponding transform for link2, as computed and published by robot_state_publisher).

Ultimately, the robot simulator (gazebo) gets replaced by a physical robot interacting with a physical world. At this point, gazebo is not running, but Rviz is still useful. Rviz can continue to display the state of the robot by rendering the model according to the joint values published.

Rviz becomes more valuable as sensors are added to the robot. For development purposes, virtual sensors are added to the robot model, and sensor values are synthesized as part of the Gazebo simulation. Adding and visualizing sensors is covered next.