
Gazebo Simulator

Unit 3: Connect to ROS

- Summary -

Estimated time to completion: **6 hours**

This unit presents concepts around creating a new world in Gazebo. You will learn to create new worlds from scratch with different models, ground, animated objects, and actors. In addition, you will learn how to use plugins and how to control worlds programmatically.

- End of Summary -

3.1 Gazebo's ROS node

In the next sections, you will need the empty world running:

► Execute

In []: `roslaunch robot_description empty_world.launch`



- Gazebo node -

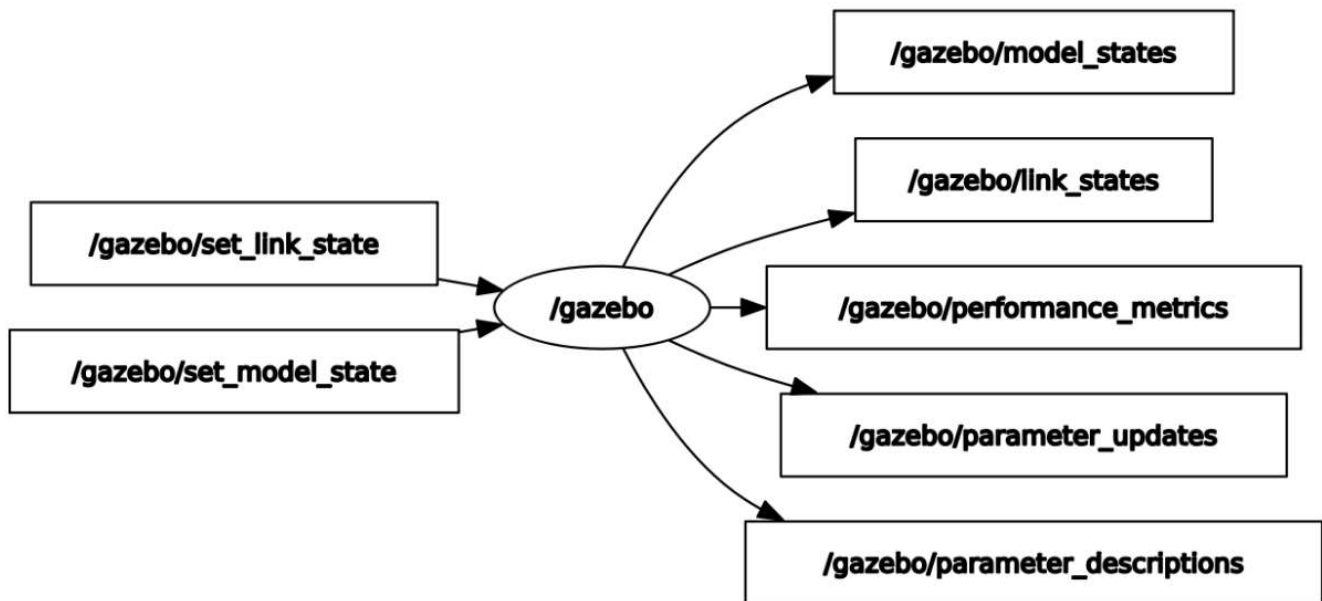
You learned how to launch a Gazebo simulation using ROS launch files. You are working like that because it creates ROS instances that represent Gazebo in your ROS environment.

The simulation is represented by a node called **/gazebo**.

You can see more details about that node using **rqt*:

► Execute

In []: rqt_graph



- End of Gazebo node -

- Gazebo topics -

The topics that Gazebo is subscribing to (**set_link_state** and **set_model_state**) can be used to change the simulation links and model positions in runtime.

It is possible to get information about the simulation in runtime using the published topics: **model_states**, **link_states**, and others.

Exercise 3.1

Insert a box in the empty world, using one terminal to echo **/gazebo/model_states** and another to publish to **/gazebo/set_model_state**. The goal is to change the position of the box using Gazebo topics.

Hints

- Check the type of messages you need to publish to **set_model_state**
- Use the auto-complete of the terminals to fill the necessary fields and update the ones you think are necessary
- Use your ROS knowledge for this exercise!

End of Exercise 3.1

Solution for Exercise 3.1

[Solution \(https://s3.eu-west-1.amazonaws.com/readme.theconstructsim.com/_others_/course_gazebo_intro/sol-ex-3.1.gif\)](https://s3.eu-west-1.amazonaws.com/readme.theconstructsim.com/_others_/course_gazebo_intro/sol-ex-3.1.gif)

End of Solution for Exercise 3.1

- End of Gazebo topics -

- Gazebo services -

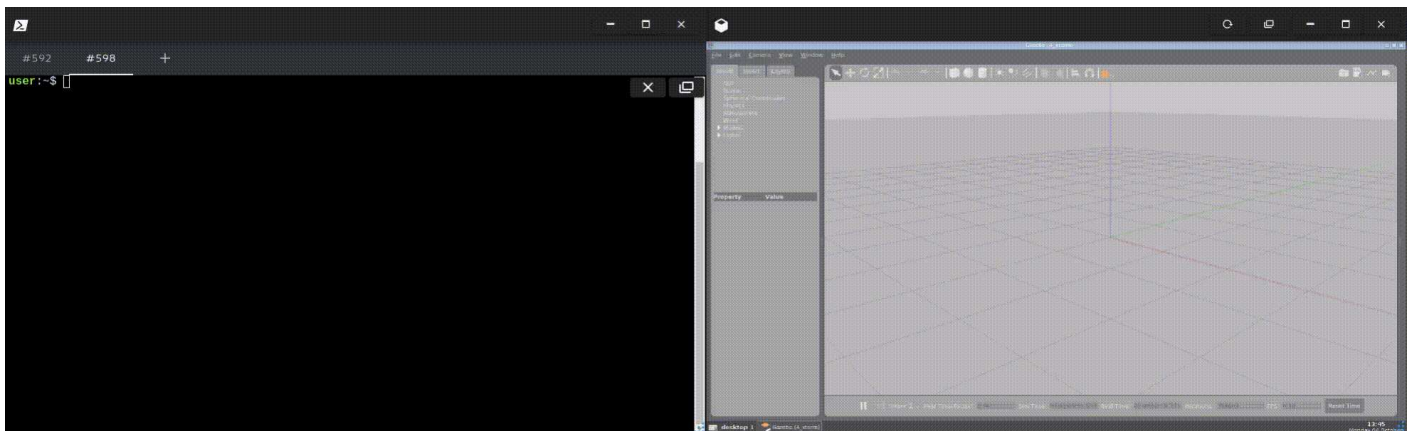
Gazebo starts with several services available. However, most of them are complex, for example, the **/gazebo/spaw_urdf_model**, which is internally used when you spawn a robot in a launch file. Therefore, it is not practical to use it from the CLI.

There are other useful and simple services worth mentioning because they can help you to develop and work with the simulator:

- **/gazebo/pause_physics**
- **/gazebo/unpause_physics**
- **/gazebo/reset_simulation**
- **/gazebo/reset_world**

Pause and unpause

The first two are used to pause and unpause the simulation:



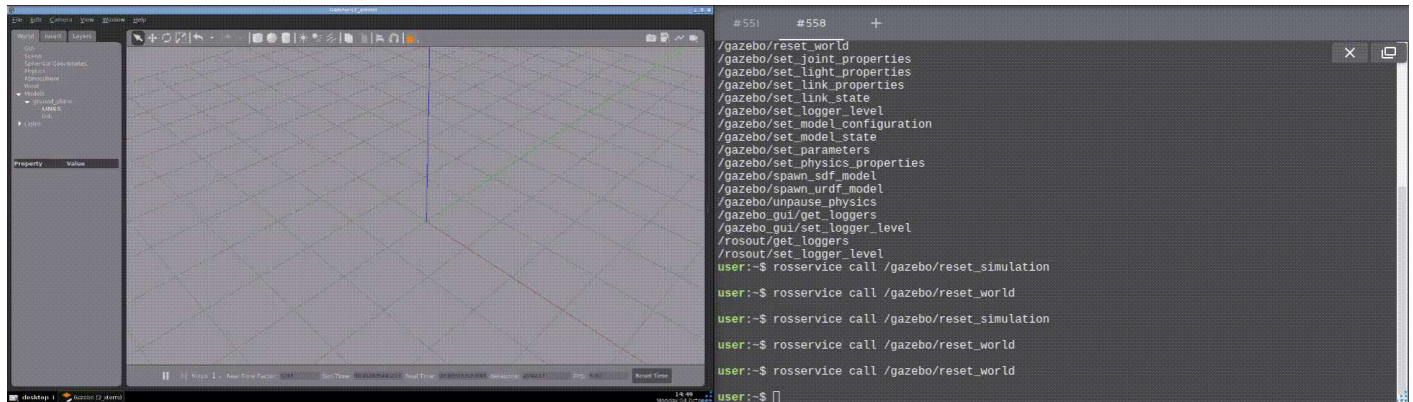
Reset world and simulation

Reset world will put all objects back in their initial places. So even if you create an object after the simulation has started, a reset command will put it back, instead of removing it.

Reset simulation does the same, except it will also reset the simulation time. You can see the simulation time at the bottom bar.

- Practice -

Add some simple objects to the empty simulation and call the services you have just learned to see the effects.



- End of Practice -

- End of Gazebo services -

3.2 Robot Gazebo Tag

In the robot's **URDF** file, every time a URDF model is spawned, Gazebo converts it internally to SDF. Due to that, you need to specify some attributes directly to Gazebo. For example, you currently do not have any colors set for the robot. That is because the material you set inside the **visual** tag only works for tools prepared to work with URDF, and Gazebo conversion does not parse it!

You need to set up the material using, let us say, Gazebo's language. There is a special tag for it in URDF. The official documentation is here: https://classic.gazebosim.org/tutorials?tut=ros_urdf (https://classic.gazebosim.org/tutorials?tut=ros_urdf).

- Material -

Start changing the colors of the robot. You need to set up a new tag **<gazebo>** for every link you want to change color. Add the following to the beginning of the robot description:

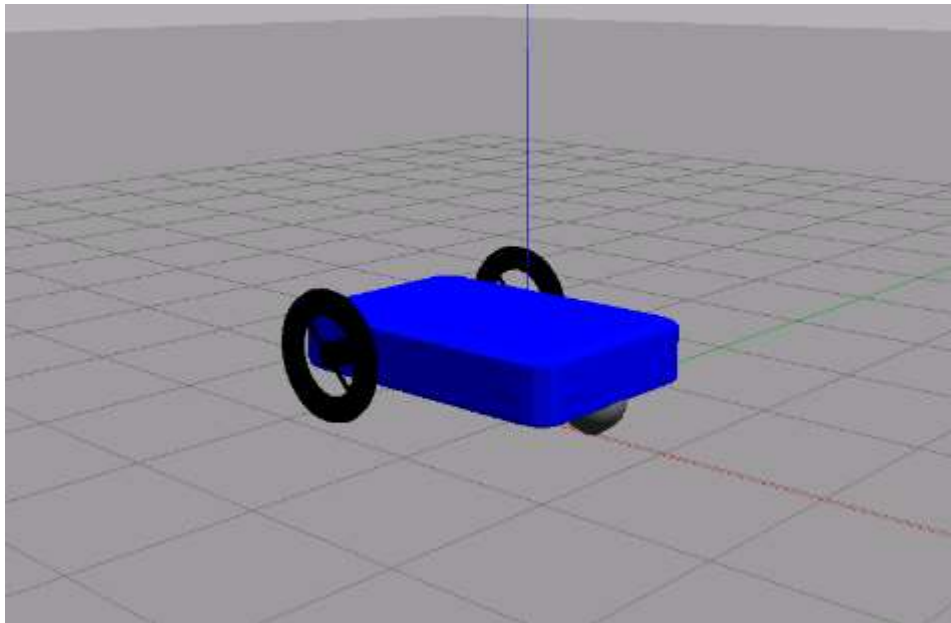
 Copy and Paste

```
In [ ]: <robot name="robot">
        <gazebo reference="link_chassis">
            <material>Gazebo/Blue</material>
        </gazebo>
        <gazebo reference="link_caster_wheel">
            <material>Gazebo/Grey</material>
        </gazebo>
        <gazebo reference="link_left_wheel">
            <material>Gazebo/Black</material>
        </gazebo>
        <gazebo reference="link_right_wheel">
            <material>Gazebo/Black</material>
        </gazebo>

        ...

    </robot>
```

Delete the model from the simulation if you have one there and relaunch the spawn. The robot should look like the below:



So, which colors do you have available in Gazebo?

There is no precise answer because what you have used is not a color, but a script. Gazebo has many pre-defined scripts that you can find here:

► Execute

```
In [ ]: vim /usr/share/gazebo-11/media/materials/scripts/gazebo.material
```

You can even define new materials. You are not going deep into this matter in the course. You can get more information here: https://classic.gazebosim.org/tutorials?tut=ros_urdf#Materials:Usingpropercolorsandtextures (https://classic.gazebosim.org/tutorials?tut=ros_urdf#Materials:Usingpropercolorsandtextures).

- Practice -

Read the file **/usr/share/gazebo-11/media/materials/scripts/gazebo.material** and change the materials setup to the links of the robot. Spawn it again and see the changes. You can customize your robot using the available materials.

- End of Practice -

- End of Material -

- Friction -

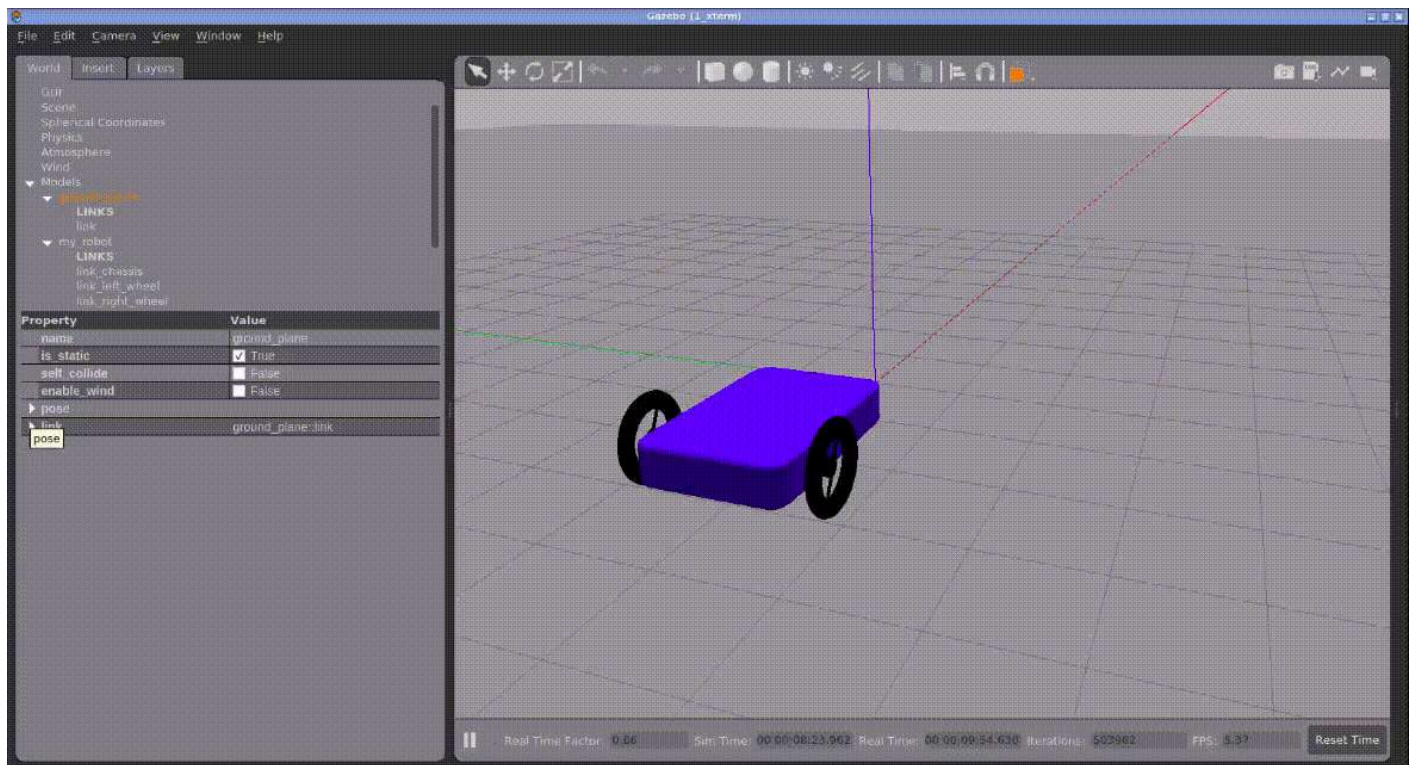
The friction of the robot parts is significant to the dynamics you want to implement. In the current example, where you have a two-wheeled robot, you need both wheels with high friction and the caster wheel with no friction.

Check the difference that it makes in the robot model. Open the empty world and launch the current robot. Click on the model, select one of the wheel links and check its surface attribute:

▼ my_robot	
LINKS	
link_chassis	
link_left_wheel	
link_right_wheel	
JOINTS	
joint_chassis_left_wheel	
joint_chassis_right_wheel	
Property	Value
▶ collision	my_robot::link_right_wheel::link_ri...
▼ collision	my_robot::link_right_wheel::link_ri...
name collision	my_robot::link_right_wheel::link_ri...
laser_retro	0.00
▶ pose	
▶ geometry	
▼ surface	
restitution_coefficient	0.00
bounce_threshold	100,000.00
soft_cfm	0.00
soft_erp	0.20
kp	2,147,483,647.00
kd	1.00
max_vel	0.01
min_depth	0.00
▼ friction	
mu	1.00
mu2	1.00
slip1	0.00
slip2	0.00

You have **mu**, **mu2**, **slip1**, and **slip2**. These are the levels of friction coefficient and slip. Check official SDF documentation for more details about the physics and calculation. You are just using values that are good enough for this demo. (<http://gazebosim.org/tutorials?tut=friction>)

If you apply a force to the X-axis of the robot now, it will not move. This is because even though you have joints in the wheels, the caster wheel has high friction. You need to make it with a smaller friction coefficient to allow movement in the robot. Try to apply the force like it is shown below:



Add the following code to the robot description:

```
<gazebo reference="joint_chassis_caster_wheel">
  <preserveFixedJoint>true</preserveFixedJoint>
</gazebo>
```

This will make the SDF conversion keep the chassis, and the caster wheel separated. Even though they are two different links in the URDF description, Gazebo merges links bound by fixed joints, preventing friction of the caster wheel, without setting the friction of the chassis.

Delete the model and spawn again. You must have the following in the model links menu:



- Exercise -

You will set up the friction for both wheels and the caster wheel. How to do that?

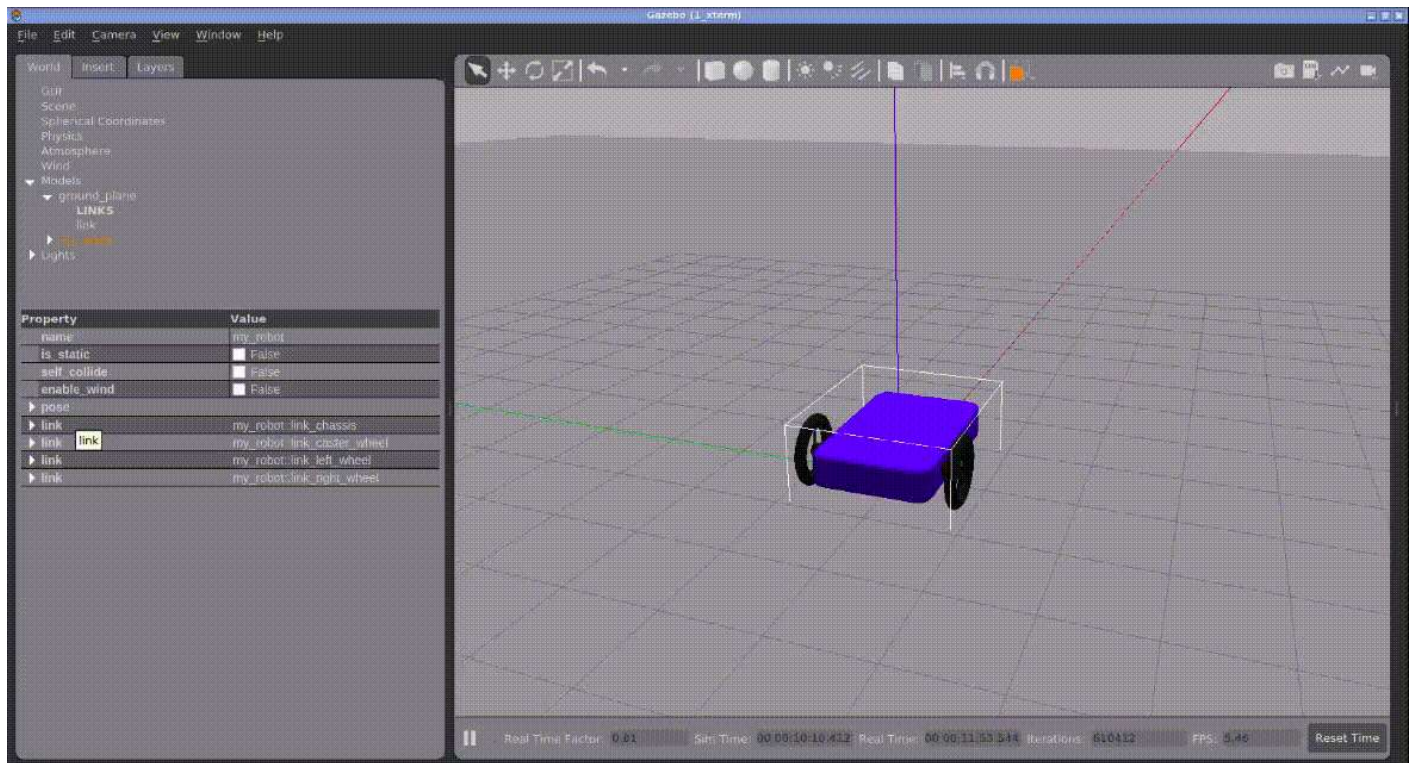
- Use the Gazebo tags you have already created for the links (to set the colors) and set up the attributes below:

```
<mu1></mu1>
```

```
<mu2></mu2>
```

- The accepted values go from **0** to **1**.
- Set it up for **link_caster_wheel**, **link_left_wheel** and **link_right_wheel**
- The highest value, the less slippery the link

Expected result with no friction in the caster wheel:



- End of Exercise -

- Solution -

Do not check the solution before trying to completing the exercise on your own! Instead, review the solution when you have finished the exercise. It is an excellent resource to compare with your solution.

If you have problems solving the exercise, use our forum support!

[robot.urdf \(unit-03/robot-friction.urdf.txt\)](#)

- End of Solution -

- End of Friction -

3.2 ROS Plugin - Robot differential driver

You have the robot ready to move! Your next step is to connect it to ROS is to make it possible to move the robot by it publishing to a ROS topic. There is a plugin for it, the **gazebo_ros_diff_drive**, and it can be included in the URDF model using the **Gazebo** tag. This is how it goes:

 Copy and Paste

In []: 

```
<gazebo>
  <plugin filename="libgazebo_ros_diff_drive.so" name="differential_drive_cc
    <alwaysOn>true</alwaysOn>
    <updateRate>20</updateRate>
    <leftJoint>joint_chassis_left_wheel</leftJoint>
    <rightJoint>joint_chassis_right_wheel</rightJoint>
    <wheelSeparation>1.66</wheelSeparation>
    <wheelDiameter>0.8</wheelDiameter>
    <torque>10</torque>
    <commandTopic>cmd_vel</commandTopic>
    <odometryTopic>odom</odometryTopic>
    <odometryFrame>odom</odometryFrame>
    <robotBaseFrame>link_chassis</robotBaseFrame>
  </plugin>
</gazebo>
```



Paste that instruction after the others you have just added. Then, inside the robot URDF definition, delete the robot and spawn it again.

What has changed? Nothing new to the robot model, but check the available ROS topics! Go to the terminal:

► Execute

In []: 

```
rostopic list
```

```

user:~$ rostopic list
/clock
/cmd_vel
/gazebo/link_states
/gazebo/model_states
/gazebo/parameter_descriptions
/gazebo/parameter_updates
/gazebo/performance_metrics
/gazebo/set_link_state
/gazebo/set_model_state
/odom
/rosout
/rosout_agg
/tf

```

You can move the robot by publishing to **/cmd_vel** as you have just configured. The angular velocities will take into account the separation between the wheels. The linear velocity takes into account the wheel diameter configured. That is why you used two times the value of the radius when you created the wheels in the robot description!

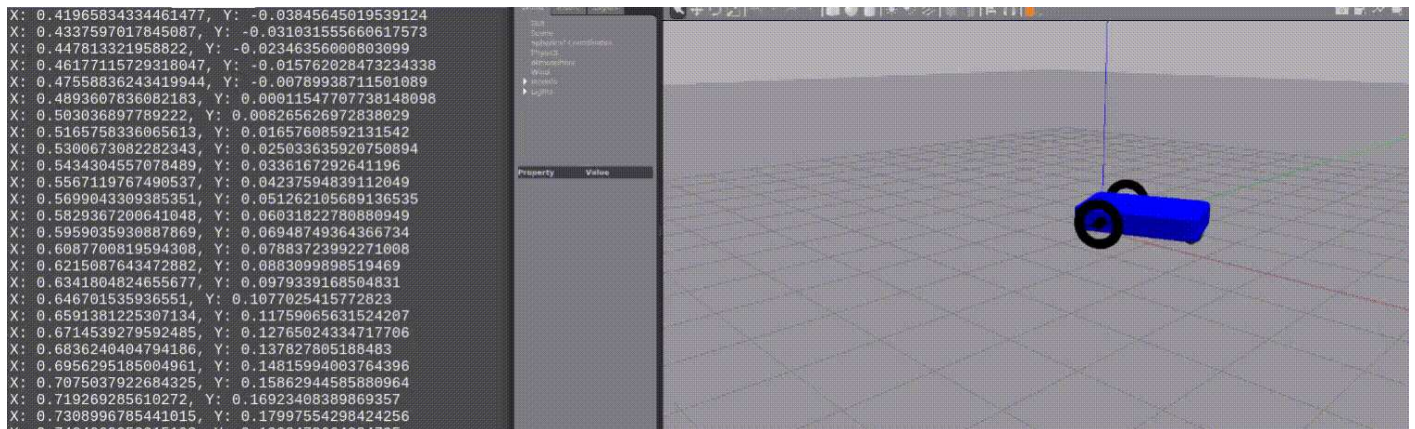
You also have the **/odom** topic that receives messages from this Gazebo plugin containing information about the robot's position.

- Exercise -

Create a simple **Python script** that moves the robot based on two arguments: **linear** and **angular** velocities.

While the robot is moving, the program must display the robot's position on the **X** and **Y**-axis.

Expected result:



- End of Exercise -

- Solution -

Do not check the solution before trying to complete the exercise on your own! Instead, review the solution when you have finished the exercise. It is an excellent resource to compare with your solution.

If you have problems solving the exercise, use our forum support!

[move-display-odom.py \(unit-03/move-display-odom.py.txt\)](#)

- End of Solution -

3.3 ROS Plugin - Robot laser sensor

Another popular Gazebo plugin is the **Laser**. See how to add a laser scan to the robot.

First, create a new link that represents the scan sensor. Then, add the following code to the robot definition:

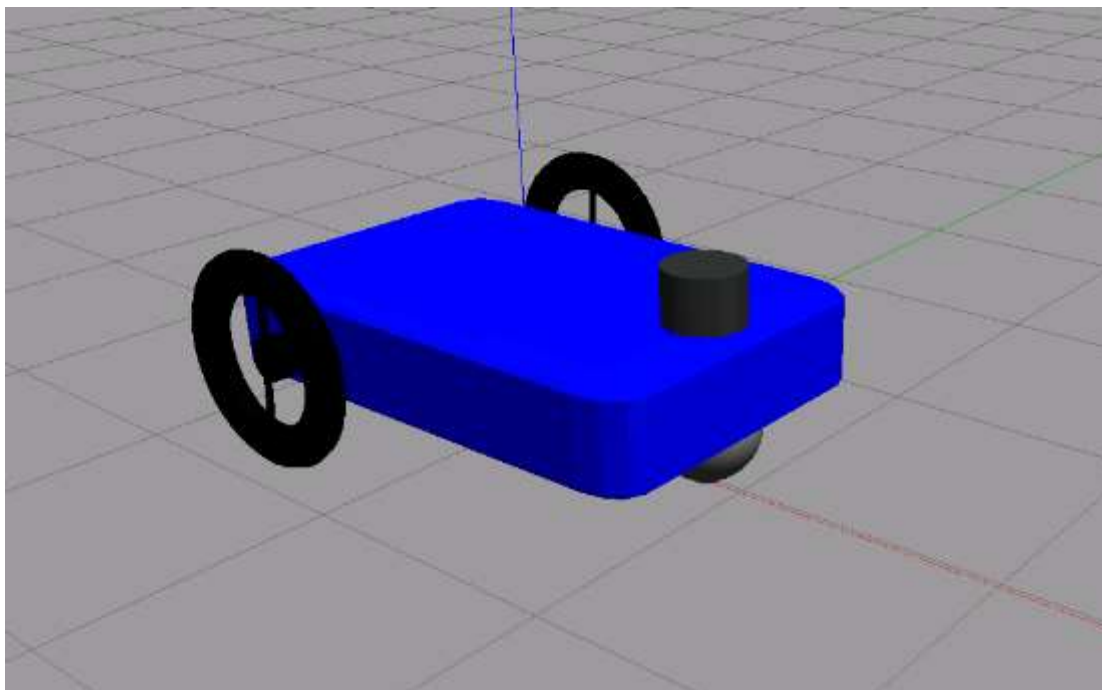
In []:

```
<!-- Gazebo tags - Laser scan -->
<gazebo reference="joint_laser_scan_chassis">
  <preserveFixedJoint>true</preserveFixedJoint>
</gazebo>
<gazebo reference="link_laser_scan">
  <material>Gazebo/DarkGrey</material>
</gazebo>

<!-- Laser scan -->
<joint name="joint_laser_scan_chassis" type="fixed">
  <origin rpy="0 0 0" xyz="0.8 0 0.3" />
  <child link="link_laser_scan" />
  <parent link="link_chassis" />
  <joint_properties damping="1.0" friction="1.0" />
</joint>
<link name="link_laser_scan">
  <inertial>
    <origin xyz="0 0 0" rpy="0 0 0" />
    <mass value="0.5" />
    <inertia ixx="0.000252666666667" ixy="0" ixz="0" iyy="0.000252666666667"
  </inertial>
  <visual>
    <origin xyz="0 0 0" rpy="0 0 0" />
    <geometry>
      <cylinder radius="0.15" length="0.20"/>
    </geometry>
    <material name="Red">
      <color rgba="0.7 0.1 0.1 1" />
    </material>
  </visual>
  <collision>
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <geometry>
      <cylinder radius="0.15" length="0.20"/>
    </geometry>
  </collision>
</link>
```



Remove the model in the simulation and spawn it again. You must have something like the below:



And finally, to make it work, add a new Gazebo tag, referencing the **link_laser_scan** link. It means the sensor reading will start from the initial point of this link.

In []:

```
<gazebo reference="link_laser_scan">
  <sensor type="ray" name="head_hokuyo_sensor">
    <pose>0 0 0 0 0 0</pose>
    <visualize>true</visualize>
    <update_rate>20</update_rate>
    <ray>
      <scan>
        <horizontal>
          <samples>720</samples>
          <resolution>1</resolution>
          <min_angle>-1.570796</min_angle>
          <max_angle>1.570796</max_angle>
        </horizontal>
      </scan>
      <range>
        <min>0.20</min>
        <max>10.0</max>
        <resolution>0.01</resolution>
      </range>
      <noise>
        <type>gaussian</type>
        <mean>0.0</mean>
        <stddev>0.01</stddev>
      </noise>
    </ray>
    <plugin name="gazebo_ros_head_hokuyo_controller" filename="libgazebo_r
      <topicName>/laser/scan</topicName>
      <frameName>sensor_laser</frameName>
    </plugin>
  </sensor>
</gazebo>
```



The parameter **visualize** is helpful in understanding how the scan rays will act in the simulation. You can set it up to false later if you prefer. Finally, **pose** sets the position of the rays with respect to the origin of the referenced link.

The param **update_rate** is where you set up the frequency that Gazebo will publish the data.

Inside **ray/scan/horizontal**, it is defined as the number of samples taken between **min_angle** and **max_angle**, as well as its resolution. Take note of these data because it defines where the readings begin, the steps, and where it finishes.

range defines the minimum and maximum points and the **resolution**. **IMPORTANT!** The minimum reading cannot be smaller than the link that contains the sensor. Otherwise, the readings would always hit the internal shape of the link, which does not make sense. So, in this example, the minimum reading is **0.20**, and the radius of the cylinder representing your sensor is **0.15**, so it starts reading outside the sensor geometry.



There are other parameters you can explore, checking its official documentation:

http://wiki.ros.org/gazebo_ros_pkgs (http://wiki.ros.org/gazebo_ros_pkgs).

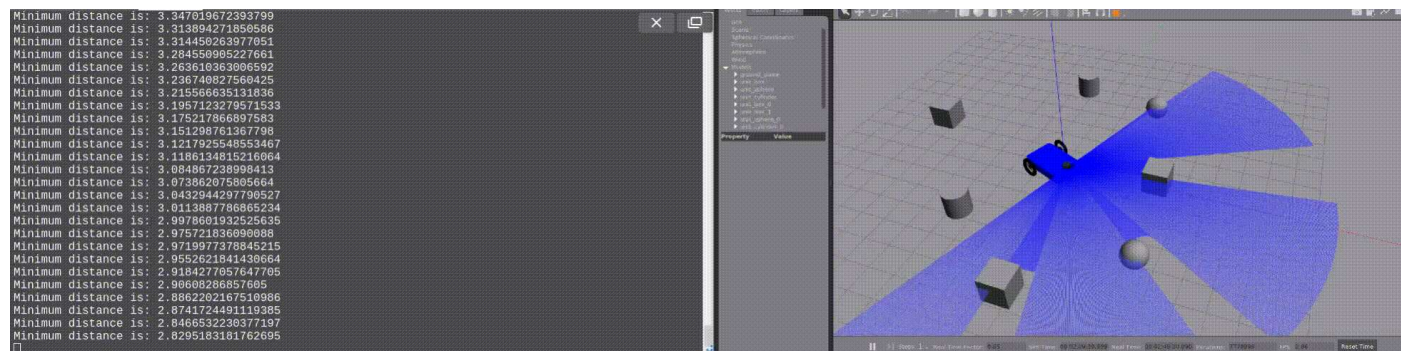
Remove the current robot and spawn it again with the scan plugin. Add some blocks around the robot and check how it works. Check the rostopic as well.

- Exercise -

Create a simple **Python script** that moves the robot based on two arguments: **linear** and **angular** velocities.

While the robot moves, the program must display the minimum distance read by the laser scan sensor.

Expected result:



- End of Exercise -

- Solution -

Do not check the solution before trying to completing the exercise on your own! Instead, review the solution when you have finished the exercise. It is an excellent resource to compare with your solution.

If you have problems solving the exercise, use our forum support!

[move-laser.py \(unit-03/move-laser.py.txt\)](#)

- End of Solution -

3.4 XACRO = XML + Macro

- XACRO files -

Xacro makes it possible to create more organized and shorter XML files. For example, you can make these files more organized since you are using XML to describe robots in URDF files.

Including files

For example, you have a big section to describe Gazebo integration with the robot. You can put it into a separate file. Check how!

- Create a new file **robot.gazebo**
- Rename the original file **robot.urdf** to **robot.xacro**

Cut all **<gazebo>** tags from **robot.xacro** and paste them into the new **robot.gazebo** file, inside the following template:

 Copy and Paste

```
In [ ]: <?xml version="1.0"?>
        <robot>
            <!-- Paste here -->
        </robot>
```

Finally, include this new file in the robot description file. You also need to add this new Xacro definition in the **robot** tag:

 Copy and Paste

```
In [ ]: <?xml version="1.0" ?>

        <robot name="robot" xmlns:xacro="http://www.ros.org/wiki/xacro">

            <xacro:include filename="$(find robot_description)/urdf/robot.gazebo" />

            <!-- Link - chassis -->
            ...

        </robot>
```

To make this XACRO file understandable for Gazebo, convert it just before spawning the model. Open the **spawn.launch** file and replace the parameter **robot_descript** with the one below:

```
In [ ]: <param name="robot_description" command="$(find xacro)/xacro --inorder '$(find robot_description)/urdf/robot.gazebo' --verbose --print" />
```

This converts from XACRO to URDF, so the description returns to its original state, as URDF, and the developer can use the best of XACROs!

Math Expressions

Another useful feature is using variables and math expressions. For example, if you have the chassis weight defined as 10 kg, you can put it into a property:

 Copy and Paste

In []:

```
<!-- Link - chassis -->
<robot name="robot" xmlns:xacro="http://www.ros.org/wiki/xacro">

    <xacro:include filename="$(find robot_description)/urdf/robot.gazebo" />

    <!-- Parameters -->
    <xacro:property name="chassis_mass" value="10" />

    <!-- Link - chassis -->
    <link name="link_chassis">
        <inertial>
            <mass value="${chassis_mass}" />
```

If you want to use a common value, for example, **half_chassis**, you could assign a math expression like:

```
<mass value="${2 * chassis_mass}" />
```

There is much more you can do with XACRO. Check the official reference: <http://wiki.ros.org/xacro> (<http://wiki.ros.org/xacro>).

- End of XACRO files -

- Exercise -

Now that you know how to use XACROs, replace all the places where you used PI factors with a math expression.

- Create a new property called **pi**
- Replace, for example, 1.57 with an expression of **pi**

The advantage is that you can use many decimal places for **pi**, to have a more precise model.

- End of Exercise -

- Solution -

Do not check the solution before trying to complete the exercise on your own! Instead, review the solution when you have finished the exercise. It is a good resource to compare with your solution.

If you have problems solving the exercise, use our forum support!

- [robot.xacro \(unit-03/robot.xacro-ex.txt\)](#)

- End of Solution -

3.5 ROS Plugin - Joint Control

So far, you created the robot and controlled the wheels using the `diff_driver` plugin, although sometimes you need to control a specific joint. First, create a new link called **tail** and attach it to the robot. It will not have a role in the robot tasks, but you will learn how to control joints with it!

Add the tail to the **robot.xacro** file:

 Copy and Paste

```
In [ ]: <!-- Robot tail -->
<joint name="joint_tail" type="continuous">
  <origin rpy="0 0 0" xyz="-1 0 0" />
  <child link="link_tail" />
  <axis rpy="0 0 0" xyz="0 1 0" />
  <parent link="link_chassis" />
  <joint_properties damping="1.0" friction="1.0" />
</joint>
<link name="link_tail">
  <inertial>
    <origin xyz="0 0 0" rpy="0 0 0" />
    <mass value="0.5" />
    <inertia ixx="0.000252666666667" ixy="0" ixz="0" iyy="0.000252666666667"
    </inertial>
  <visual>
    <origin xyz="-0.2 0 0" rpy="0 ${pi/2} 0" />
    <geometry>
      <cylinder radius="0.05" length="0.40"/>
    </geometry>
    <material name="Red">
      <color rgba="0.7 0.1 0.1 1" />
    </material>
  </visual>
  <collision>
    <origin xyz="-0.2 0 0" rpy="0 ${pi/2} 0"/>
    <geometry>
      <cylinder radius="0.05" length="0.40"/>
    </geometry>
  </collision>
</link>
```

And update the **rviz.launch** file to use **XACRO** as well:

 Copy and Paste

In []:

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

  <param name="robot_description"
    command="$(find xacro)/xacro --inorder '$(find robot_description)/urdf

  <!-- send fake joint values -->
  <node name="joint_state_publisher_gui" pkg="joint_state_publisher_gui" type="

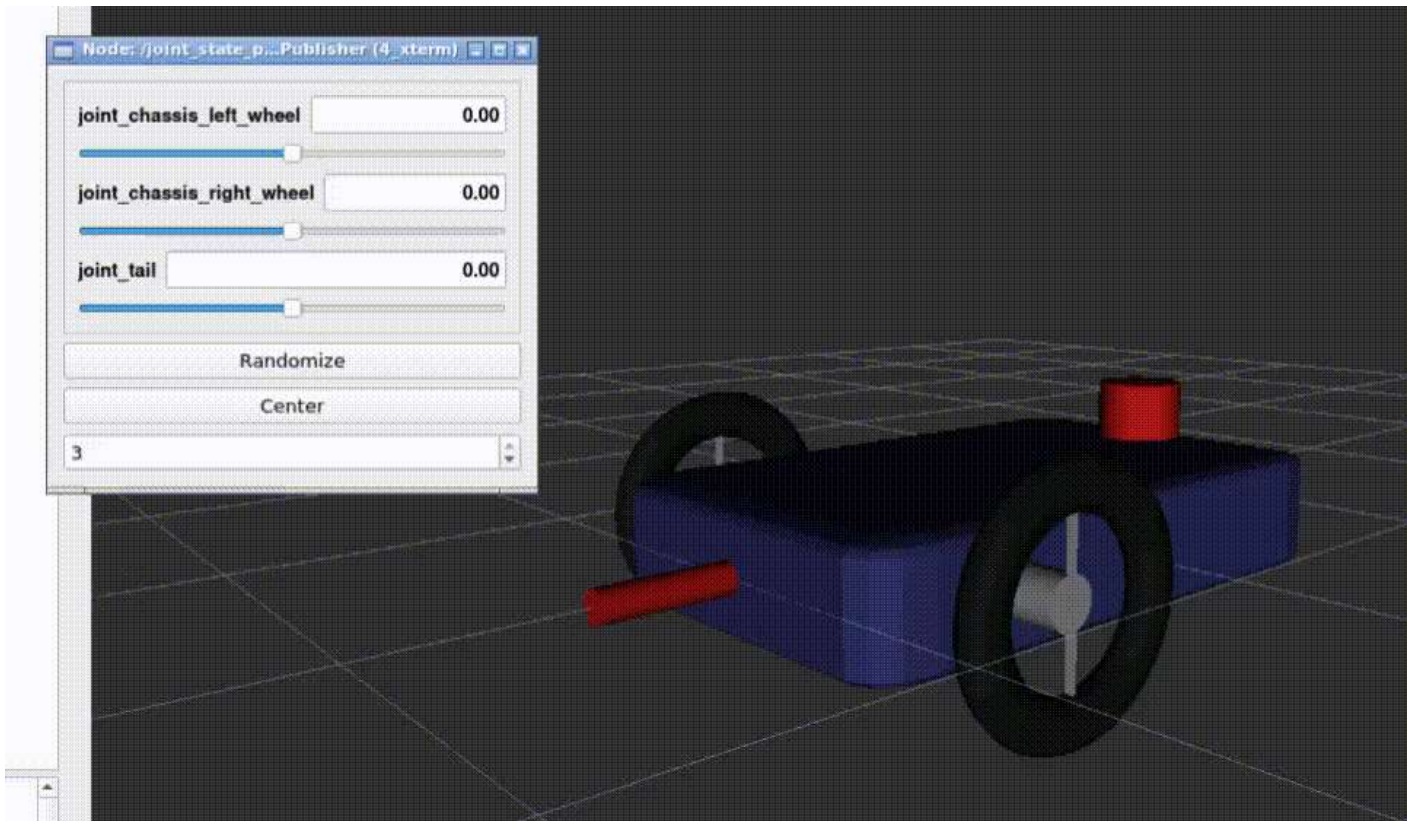
  <!-- Combine joint values -->
  <node name="robot_state_publisher" pkg="robot_state_publisher" type="robot

  <!-- Show in Rviz -->
  <node name="rviz" pkg="rviz" type="rviz" />

</launch>
```



Launch RVIZ and the robot tail should look like the below:



To control the joint, you must have a new element, a **transmission**. Add the following after the joint/link definitions:

 Copy and Paste

```
In [ ]: <transmission name="tran_tail">
        <type>transmission_interface/SimpleTransmission</type>
        <joint name="joint_tail">
            <hardwareInterface>hardware_interface/EffortJointInterface</hardwareIr
        </joint>
        <actuator name="motor_joint_tail">
            <hardwareInterface>hardware_interface/EffortJointInterface</hardwareIr
            <mechanicalReduction>1</mechanicalReduction>
        </actuator>
    </transmission>
```

The keyword **joint_tail** is the reference you will use outside the robot description. Create a new file to configure the controller parameters:

► Execute

```
In [ ]: mkdir -p ~/catkin_ws/src/robot_description/config
```




```
In [ ]: touch ~/catkin_ws/src/robot_description/config/robot_control.yaml
```



```
In [ ]: And paste the configuration below:
```



 Copy and Paste

```
In [ ]: robot:
  # Joint state controller publisher
  joint_state_controller:
    type: joint_state_controller/JointStateController
    publish_rate: 50

  # Controllers
  joint1_position_controller:
    type: effort_controllers/JointPositionController
    joint: joint_tail
    pid: {p: 2.0, i: 0.01, d: 0.01}
```



This is a PID configuration for the controller. Now, launch the controller. Create a new launch file **robot_control.launch**:

► Execute

```
In [ ]: touch ~/catkin_ws/src/robot_description/launch/robot_control.launch
```



 Copy and Paste

In []:

```
<launch>

  <!-- Load joint controller configurations from YAML file to parameter serv
  <rosparam file="$(find robot_description)/config/robot_control.yaml" comma

  <!-- load the controllers -->
  <node
    name="controller_spawner"
    pkg="controller_manager"
    type="spawner"
    respawn="false"
    output="screen"
    ns="/robot"
    args="joint_state_controller joint1_position_controller"
  />

  <!-- convert joint states to TF transforms for rviz, etc -->
  <node name="robot_state_publisher" pkg="robot_state_publisher" type="robot
    respawn="false" output="screen">
    <remap from="/joint_states" to="/robot/joint_states" />
  </node>

  <node name="rqt_reconfigure" pkg="rqt_reconfigure" type="rqt_reconfigure"
  <node name="rqt_publisher" pkg="rqt_publisher" type="rqt_publisher" />

</launch>
```

Finally, add **gazebo_ros_control** to the file **robot.gazebo**:

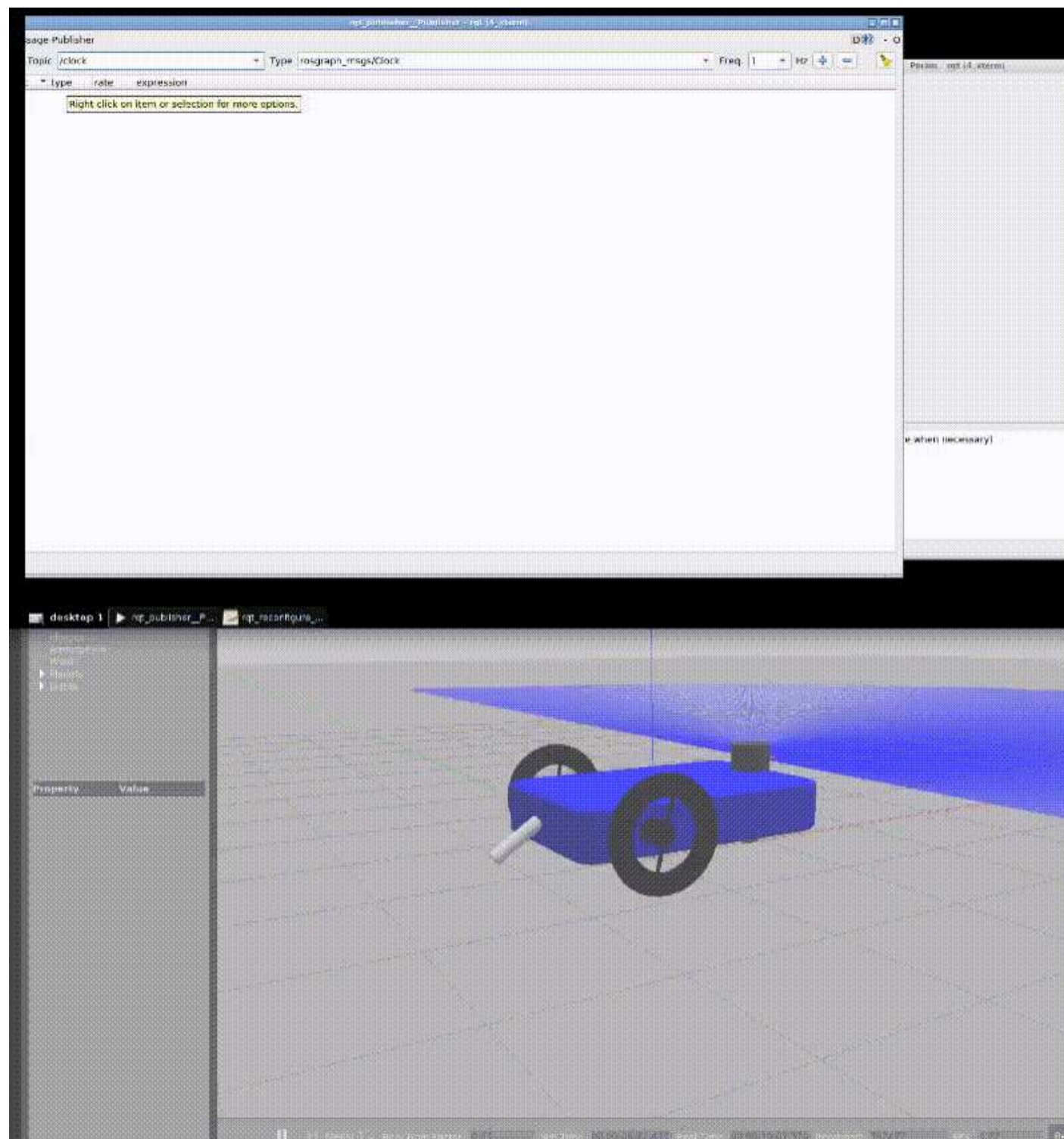
 Copy and Paste

In []:

```
<!-- Control plugin -->
<gazebo>
  <plugin name="gazebo_ros_control" filename="libgazebo_ros_control.so">
    <robotNamespace>/robot</robotNamespace>
    <robotSimType>gazebo_ros_control/DefaultRobotHWSim</robotSimType>
  </plugin>
</gazebo>
```

Launch the simulation, spawn the robot and launch the new launch file to load the controllers.

Use both **rqt_publisher** and **rqt_reconfigure** to try the controller. You can send commands and tune the PID controller. See the difference in the simulation accordingly.



- End of Practice -

3.6 References

It was shown the commonly used plugins for basic robots. You can check all plugins available here:

- http://wiki.ros.org/gazebo_ros_pkgs (http://wiki.ros.org/gazebo_ros_pkgs)
- https://github.com/ros-simulation/gazebo_ros_pkgs (https://github.com/ros-simulation/gazebo_ros_pkgs)