

# ROS/Gazebo

# Getting Started - Installation

- Install ROS
- Install Simulator\_Gazebo package

```
$ sudo apt-get install ros-melodic-simulators
```

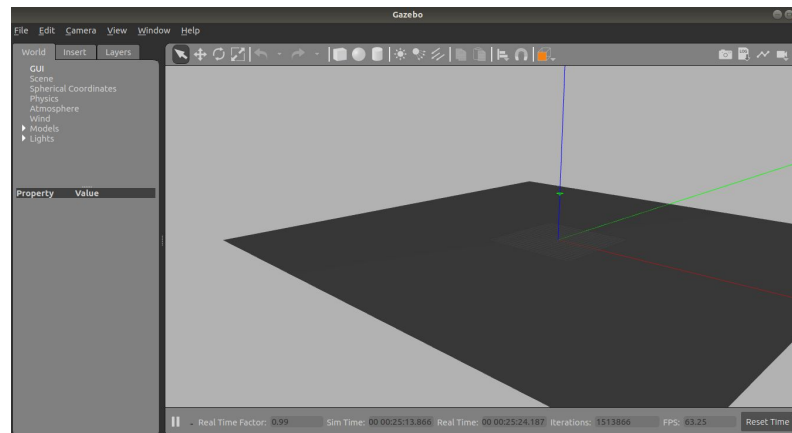
- IF ROS is configured properly, you can launch gazebo by running the following command:

```
$ roslaunch gazebo_ros empty_world.launch
```

- You can see the launch file within the gazebo\_ros package:

```
$ roscd gazebo_ros
```

```
$ cat launch/empty_world.launch
```



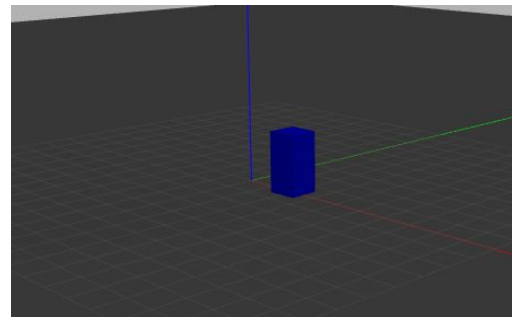
# Creating and Spawning Custom URDF Objects in Simulation

```
<robot name="simple_box">
  <link name="my_box">
    <inertial>
      <origin xyz="2 0 0" />
      <mass value="1.0" />
      <inertia ixx="1.0" ixy="0.0" ixz="0.0"
iyy="100.0" iyz="0.0" izz="1.0" />
    </inertial>
    <visual>
      <origin xyz="2 0 1"/>
      <geometry>
        <box size="1 1 2" />
      </geometry>
    </visual>
    <collision>
      <origin xyz="2 0 1"/>
      <geometry>
        <box size="1 1 2" />
      </geometry>
    </collision>
  </link>
  <gazebo reference="my_box">
    <material>Gazebo/Blue</material>
  </gazebo>
</robot>
```

- Create a file called 'object.urdf' and copy the content shown on the left and save it in the current directory.
- Use the following command to spawn (load) this object into our Gazebo empty world environment.

```
$ rosrun gazebo_ros spawn_model `pwd`/object.urdf
-urdf -z 1 -model my_object
```

- It spawns a blue box into the gazebo environment.



# Building and Controlling a Mobile Robot in Gazebo

- Objectives

- Building a simulated mobile robot from a scratch
- Controlling the robot motion from ROS
- Visualizing with RViz
- Adding Sensors to the robot

- Prerequisites:

- Working ROS and Gazebo installation
- Catkin Workspace '~/.catkin\_ws' is created
- Tested on Ubuntu 18.04 with ROS-Melodic with Gazebo version 9.0

- Credits:

- [Generation Robots Blog](#)
- [Moorerobots.com Blog](#)

- Create 3 ros packages inside the ~/catkin\_ws/src folder

```
$ catkin create pkg skbot gazebo gazebo_ros  
$ catkin create pkg skbot description  
$ catkin_create_pkg skbot_control
```

- Create your own world

```
$ roscd skbot gazebo  
$ mkdir launch worlds  
$ cd worlds  
$ gedit skbot.world
```

It has a ground and basic illumination source as shown in the adjacent block.

```
<?xml version="1.0.7" ?>  
<sdf version="1.4">  
  <world name="myworld">  
  
    <include>  
      <uri> model://sun </uri>  
    </include>  
  
    <include>  
      <uri> model://ground_plane </uri>  
    </include>  
  
  </world>  
</sdf>
```

- Create a launch file to load this world into the gazebo simulator:

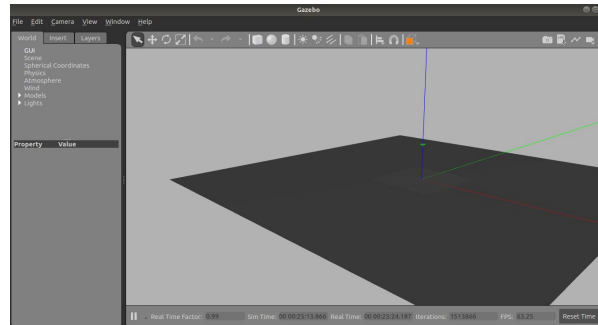
```
$ roscd skbot gazebo/launch  
$ gedit skbot_world.launch
```

And insert the following:

```
<launch>  
  <include file="$(find gazebo_ros)/launch/empty_world.launch">  
    <arg name="world_name" value="$(find skbot_gazebo)/worlds/skbot.world"/>  
    <arg name="gui" value="true"/>  
  </include>  
</launch>
```

- This gazebo environment can now be loaded using the following command:

```
$ roslaunch skbot_gazebo skbot_world.launch
```



- Create a Robot Model
  - Robot models are described using an XML file called Universal Robot Description Format (URDF) which is the native format for describing all elements of a robot. Xacro (XML macro) is an XML macro language that is useful for making shorter and clearer robot descriptions.
  - Robot description files are created inside the 'skbot\_description' folder

```
$ roscd skbot_description  
$ mkdir urdf  
$ cd urdf  
$ gedit skbot.xacro
```

The basic structure of this file looks as follows.

```
<?xml version="1.0"?>  
<robot name="skbot" xmlns:xacro="http://www.ros.org/wiki/xacro">  
  <!-- put robot description here -->  
</robot>
```

## XACRO Concepts:

- **xacro:include** import content from other files. This helps in dividing the content in different xacros and merge them using xacro:include
- **xacro:property** is used to define constant variables and use them later using `${property_name}`
- **xacro:macro** are macros with variable values. These are like functions which help in reusing the same piece of code at multiple places with new set of variables passed to the macro.
- `xmlns:xacro="http://www.ros.org/wiki/xacro"` specifies that a particular file will use xacro.



## File: skbot.xacro

```
<?xml version="1.0"?>
<robot name="skbot" xmlns:xacro="http://www.ros.org/wiki/xacro">

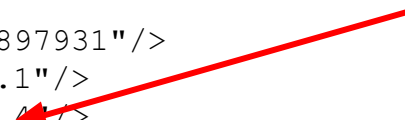
  <xacro:property name="PI" value="3.1415926535897931"/>
  <xacro:property name="chassisHeight" value="0.1"/>
  <xacro:property name="chassisLength" value="0.4"/>
  <xacro:property name="chassisWidth" value="0.2"/>
  <xacro:property name="chassisMass" value="50"/>

  <xacro:property name="wheelWidth" value="0.05"/>
  <xacro:property name="wheelRadius" value="0.1"/>
  <xacro:property name="wheelPos" value="0.25"/>
  <xacro:property name="wheelMass" value="5"/>

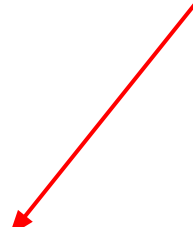
  <xacro:property name="casterRadius" value="0.05"/>
  <xacro:property name="casterMass" value="5"/>
  <xacro:property name="cameraSize" value="0.05"/>
  <xacro:property name="cameraMass" value="0.1"/>

  <xacro:include filename="$(find skbot_description)/urdf/skbot.gazebo" />
  <xacro:include filename="$(find skbot_description)/urdf/materials.xacro" />
  <xacro:include filename="$(find skbot_description)/urdf/macros.xacro" />
</robot>
```

Define variables to  
be used by various  
macros.



Include other  
files



## Rectangular Base for the robot

```
<link name="chassis">
  <pose> 0 0 0.1 0 0 0 </pose>
  <collision>
    <origin xyz="0 0 ${wheelRadius}" rpy="0 0 0"/>
    <geometry>
      <box size="${chassisLength} ${chassisWidth} ${chassisHeight}"/>
    </geometry>
  </collision>
  <visual>
    <origin xyz="0 0 ${wheelRadius}" rpy="0 0 0"/>
    <geometry>
      <box size="${chassisLength} ${chassisWidth} ${chassisHeight}"/>
    </geometry>
    <material name="orange"/>
  </visual>
  <inertial>
    <origin xyz="0 0 ${wheelRadius}" rpy="0 0 0"/>
    <mass value="${chassisMass}"/>
    <box inertia m="${chassisMass}" x="${chassisLength}" y="${chassisWidth}" z="${chassisHeight}"/>
  </inertial>
</link>
</robot>
```

Defined in materials.xacro file

Defined in macros.xacro file

- Collision tags are used by the collision detection engine
- Visual tags are used by the visual rendering engine
- Inertial tags are used by physics engine.

File: skbot.gazebo

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

```
<robot>
```

```
  <gazebo reference="chassis">
```

```
    <material>Gazebo/Orange</material>
```

```
  </gazebo>
```

```
</robot>
```

File: materials.xacro

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

```
<robot>
```

```
  <material name="black">
```

```
    <color rgba="0.0 0.0 0.0 1.0"/>
```

```
  </material>
```

```
  <material name="blue">
```

```
    <color rgba="0.0 0.0 0.8 1.0"/>
```

```
  </material>
```

```
  <material name="green">
```

```
    <color rgba="0.0 0.8 0.0 1.0"/>
```

```
  </material>
```

```
  <material name="grey">
```

```
    <color rgba="0.2 0.2 0.2 1.0"/>
```

```
  </material>
```

```
  <material name="orange">
```

```
    <color rgba="${255/255} ${108/255} ${10/255} 1.0"/>
```

```
  </material>
```

```
  <material name="white">
```

```
    <color rgba="1.0 1.0 1.0 1.0"/>
```

```
  </material>
```

```
</robot>
```

File: macros.xacro

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

```
<robot>
```

```
<macro name="cylinder_inertia" params="m r h">  
  <inertia ixx="{m*(3*r*r+h*h)/12}" ixy = "0" ixz = "0"  
    iyy="{m*(3*r*r+h*h)/12}" iyz = "0"  
    izz="{m*r*r/2}"  
/>
```

```
</macro>
```

```
<macro name="box_inertia" params="m x y z">  
  <inertia ixx="{m*(y*y+z*z)/12}" ixy = "0" ixz = "0"  
    iyy="{m*(x*x+z*z)/12}" iyz = "0"  
    izz="{m*(x*x+y*y)/12}"  
/>
```

```
</macro>
```

```
<macro name="sphere_inertia" params="m r">  
  <inertia ixx="{2*m*r*r/5}" ixy = "0" ixz = "0"  
    iyy="{2*m*r*r/5}" iyz = "0"  
    izz="{2*m*r*r/5}"  
/>
```

```
</macro>
```

```
</robot>
```

All formulas, repetitive functions could be defined in this file using macro tags.

## File:skbot\_world.launch

```
<launch>
  <include file="$(find gazebo_ros)/launch/empty_world.launch">
    <arg name="world_name" value="$(find
skbot_gazebo)/worlds/skbot.world"/>
    <arg name="gui" value="true"/>
  </include>

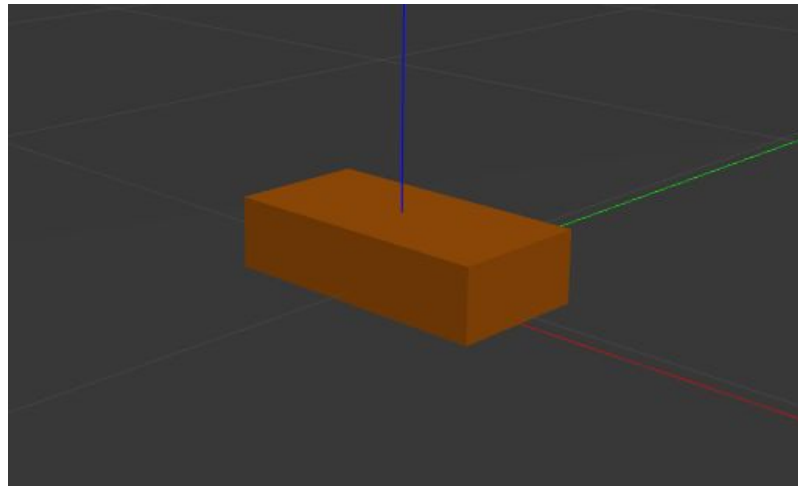
  <!-- urdf xml robot description loaded on the Parameter Server, converting
the xacro into a proper urdf file-->
  <param name="robot_description" command="$(find xacro)/xacro.py '$(find
skbot_description)/urdf/skbot.xacro'" />

  <!-- push robot description to factory and spawn robot in gazebo -->
  <node name="skbot_spawn" pkg="gazebo_ros" type="spawn_model"
output="screen"
  args="-urdf -param robot_description -model skbot " />

</launch>
```

```
$ roslaunch skbot skbot_gazebo skbot_world.launch
```

- We should see an orange box in our empty world Gazebo environment as shown in the adjacent image.
- The next step is to add caster wheels and other two main wheels to the robot chassis.
- We make modifications to the main URDF file “`skbot.xacro`” after the chassis link definition within the `<robot>` `</robot>` tags.
- We also define another macro called `<wheel>` inside the `macros.xacro` file to keep the main URDF file clean and concise.
- We will add a gazebo reference related to caster wheels in the `skbot.gazebo` file.



```

<joint name="fixed" type="fixed">
  <parent link="chassis"/>
  <child link="caster_wheel"/>
</joint>

```

Add to the `skbot.xacro` file after the chassis link.

```

<link name="caster_wheel">
  <collision>
    <origin xyz="{casterRadius-chassisLength/2} 0 {casterRadius-chassisHeight+wheelRadius}" rpy="0 0 0"/>
    <geometry>
      <sphere radius="{casterRadius}"/>
    </geometry>
  </collision>
  <visual>
    <origin xyz="{casterRadius-chassisLength/2} 0 {casterRadius-chassisHeight+wheelRadius}" rpy="0 0 0"/>
    <geometry>
      <sphere radius="{casterRadius}"/>
    </geometry>
    <material name="red"/>
  </visual>
  <inertial>
    <origin xyz="{casterRadius-chassisLength/2} 0 {casterRadius-chassisHeight+wheelRadius}" rpy="0 0 0"/>
    <mass value="{casterMass}"/>
    <sphere_inertia m="{casterMass}" r="{casterRadius}"/>
  </inertial>
</link>
<wheel lr="left" tY="1"/>
<wheel lr="right" tY="-1"/>
</robot>

```

Defined in `macros.xacro` file

```

<macro name="wheel" params="lr tY">
  <link name="\${lr}_wheel">
    <collision>
      <origin xyz="0 0 0" rpy="0 \${PI/2} \${PI/2}" />
      <geometry>
        <cylinder length="\${wheelWidth}" radius="\${wheelRadius}" />
      </geometry>
    </collision>
    <visual>
      <origin xyz="0 0 0" rpy="0 \${PI/2} \${PI/2}" />
      <geometry>
        <cylinder length="\${wheelWidth}" radius="\${wheelRadius}" />
      </geometry>
      <material name="white"/>
    </visual>
    <inertial>
      <origin xyz="0 0 0" rpy="0 \${PI/2} \${PI/2}" />
      <mass value="\${wheelMass}" />
      <cylinder_inertia m="\${wheelMass}" r="\${wheelRadius}" h="\${wheelWidth}" />
    </inertial>
  </link>

  <gazebo reference="\${lr}_wheel">
    <mu1 value="1.0" />
    <mu2 value="1.0" />
    <kp value="10000000.0" />
    <kd value="1.0" />
    <fdirl value="1 0 0" />
    <material>Gazebo/White</material>
  </gazebo>

```

Additions to file: macros.xacro



```

<joint name="${lr}_wheel_hinge" type="continuous">
  <parent link="chassis"/>
  <child link="${lr}_wheel"/>
  <origin xyz="${-wheelPos+chassisLength}
${tY*wheelWidth/2+tY*chassisWidth/2} ${wheelRadius}" rpy="0 0 0" />
  <axis xyz="0 1 0" rpy="0 0 0" />
  <limit effort="100" velocity="100"/>
  <joint_properties damping="0.0" friction="0.0"/>
</joint>

```

```

<transmission name="${lr}_trans">
  <type>transmission_interface/SimpleTransmission</type>
  <joint name="${lr}_wheel_hinge">
    <hardwareInterface>EffortJointInterface</hardwareInterface>
  </joint>
  <actuator name="${lr}Motor">
    <hardwareInterface>EffortJointInterface</hardwareInterface>
    <mechanicalReduction>10</mechanicalReduction>
  </actuator>
</transmission>
</macro>

```

```
</robot>
```

Transmission element is used by  
ros\_control, required for controlling  
the robot.

Addition to File: skbot.gazebo

```

<gazebo
reference="caster_wheel">
  <mu1>0.0</mu1>
  <mu2>0.0</mu2>

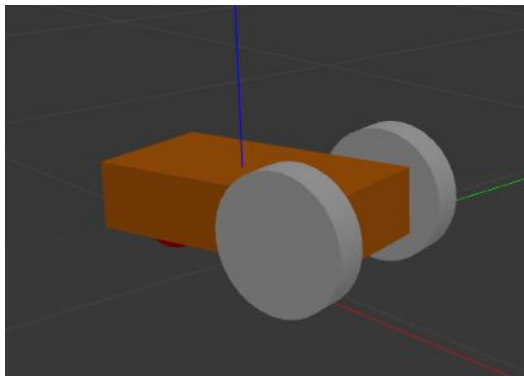
```

```

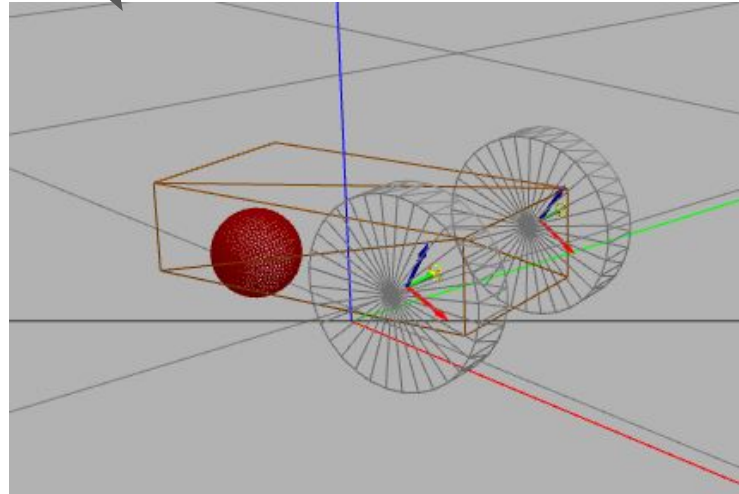
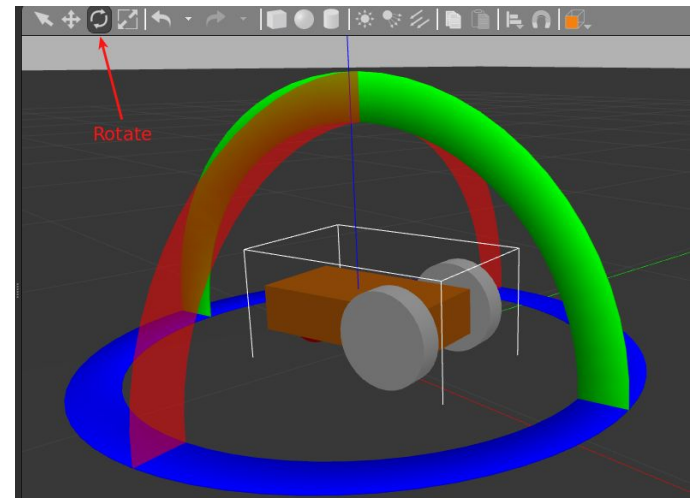
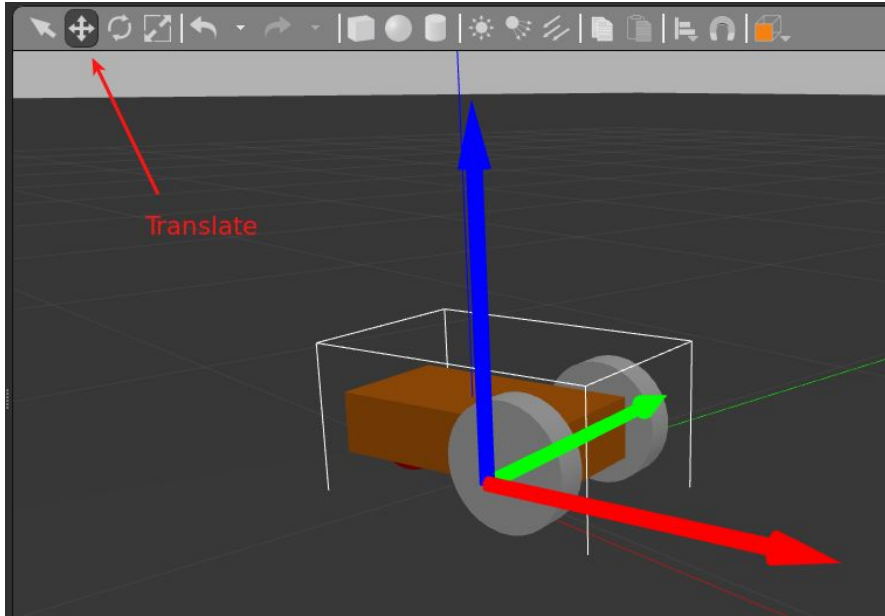
<material>Gazebo/Red</material>
</gazebo>

```

```
</robot>
```



view->wireframes



- Moving the robot
  - For this, we need to connect Gazebo to ROS using gazebo plugins.
  - There are different kinds of plugins:
    - World: Dynamic changes to the world like illumination, gravity, inserting models.
    - Model: Manipulation of models (robots) - moving the robot
    - Sensor: Feedback from virtual sensor like camera, laser scanner etc.
    - System: plugins that are loaded by GUI, like saving images
  - To activate plugin, add the following to the file: `skbot.gazebo`

```
<gazebo>
  <plugin name="gazebo_ros_control" filename="libgazebo_ros_control.so">
    <robotNamespace>/skbot</robotNamespace>
  </plugin>
</gazebo>
```

- In order to use this plugin, some additional configurations are required. This is defined in the package “`skbot_control`” as described next.

```
$ roscd skbot_control
$ mkdir config
$ cd config
$ gedit skbot_control.yaml
```

We define three controllers: one for each Wheel and one for publishing joint states.

```
$ roscd myrobot_control
$ mkdir launch
$ gedit myrobot_control.launch
```

Launch the controller by adding the following Line to the 'skbot\_world.launch' file within The <launch> </launch> tags

File: skbot\_control.yaml

```
skbot:
  # Publish all joint states -----
  joint_state_controller:
    type: joint_state_controller/JointStateController
    publish_rate: 50

  # Effort Controllers -----
  leftWheel_effort_controller:
    type: effort_controllers/JointEffortController
    joint: left_wheel_hinge
    pid: {p: 100.0, i: 0.1, d: 10.0}
  rightWheel_effort_controller:
    type: effort_controllers/JointEffortController
    joint: right_wheel_hinge
    pid: {p: 100.0, i: 0.1, d: 10.0}
```

```
<!-- ros_control skbot launch file -->
<include file="$(find skbot_control)/launch/skbot_control.launch" />
```

## File: skbot\_control.launch

```
<launch>
  <!-- Load joint controller configurations from YAML file to parameter server -->
  <rosparam file="$(find skbot_control)/config/skbot_control.yaml" command="load"/>

  <!-- load the controllers -->
  <node name="controller_spawner"
    pkg="controller_manager"
    type="spawner" respawn="false"
    output="screen" ns="/skbot"
    args="joint state controller
    rightWheel_effort_controller
    leftWheel_effort_controller"
  />

  <!-- convert joint states to TF transforms for rviz, etc -->
  <node name="robot_state_publisher" pkg="robot_state_publisher"
    type="robot_state_publisher" respawn="false" output="screen">
    <param name="robot_description" command="$(find xacro)/xacro.py '$(find
    skbot_description)/urdf/skbot.xacro'" />
    <remap from="/joint_states" to="/skbot/joint_states" />
  </node>
</launch>
```

This file performs two functions:

- Loads necessary controllers and its parameters as defined in the yaml file.
- Another node to provide 3D transformations (/tf) for the robot

In one terminal, launch the gazebo environment

```
$ roslaunch skbot_gazebo skbot_world.launch
```

In other terminal, type the following command to move the robot

```
$ rostopic list
```

```
$ rostopic pub -1  
/skbot/leftWheel_effort_controller/command  
std_msgs/Float64 "data: 1.0"
```

```
$ rostopic pub -1  
/skbot/rightWheel_effort_controller/command  
std_msgs/Float64 "data: 1.5"
```

You can also see the robot joint states by using the following command:

```
$ rostopic echo /skbot/joint_states
```

```
$ rostopic list  
/clock  
/gazebo/link_states  
/gazebo/model_states  
/gazebo/parameter_descriptions  
/gazebo/parameter_updates  
/gazebo/set_link_state  
/gazebo/set_model_state  
/rosout  
/rosout_agg  
/skbot/joint_states  
/skbot/leftWheel_effort_controller/command  
/skbot/rightWheel_effort_controller/command  
/tf  
/tf_static
```

## Teleoperation of the Robot

- It is not convenient to control each wheel separately. So, we will make use of another plugin called “differential drive” to make it easier to control the robot. Add the following into the skbot.gazebo file inside the <robot> </robot> tags:

```
<gazebo>
  <plugin name="differential_drive_controller" filename="libgazebo_ros_diff_drive.so">
    <alwaysOn>true</alwaysOn>
    <updateRate>100</updateRate>
    <leftJoint>left_wheel_hinge</leftJoint>
    <rightJoint>right_wheel_hinge</rightJoint>
    <wheelSeparation>${chassisWidth+wheelWidth}</wheelSeparation>
    <wheelDiameter>${2*wheelRadius}</wheelDiameter>
    <torque>20</torque>
    <commandTopic>skbot/cmd_vel</commandTopic>
    <odometryTopic>skbot/odom</odometryTopic>
    <odometryFrame>skbot/odom</odometryFrame>
    <robotBaseFrame>chassis</robotBaseFrame>
  </plugin>
</gazebo>
```

- Now relaunch the skbot\_world.launch file and run the following command a separate terminal:

```
$ rosrun turtlesim turtle_teleop_key /turtle1/cmd_vel:=/ skbot/cmd_vel
```

- Now you should be able to control the robot motion using the arrow keys.

## Visualization with Rviz

- Create a new launch file for RViz

```
$ roscd skbot description
$ mkdir launch
$ cd launch
$ gedit skbot_rviz.launch
```

- Now insert the content shown on right hand side. It essentially creates new node to publish joint states to be used by Rviz.
- Launch this file on a separate terminal while gazebo environment is running

```
$ roslaunch
skbot description
skbot_rviz.launch
```

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

```
  <param name="robot_description" command="$(find
xacro)/xacro.py '$(find
skbot_description)/urdf/skbot.xacro'"/>
```

```
  <!-- send fake joint values -->
  <node name="joint_state_publisher"
pkg="joint_state_publisher"
type="joint_state_publisher">
    <param name="use_gui" value="False"/>
  </node>
```

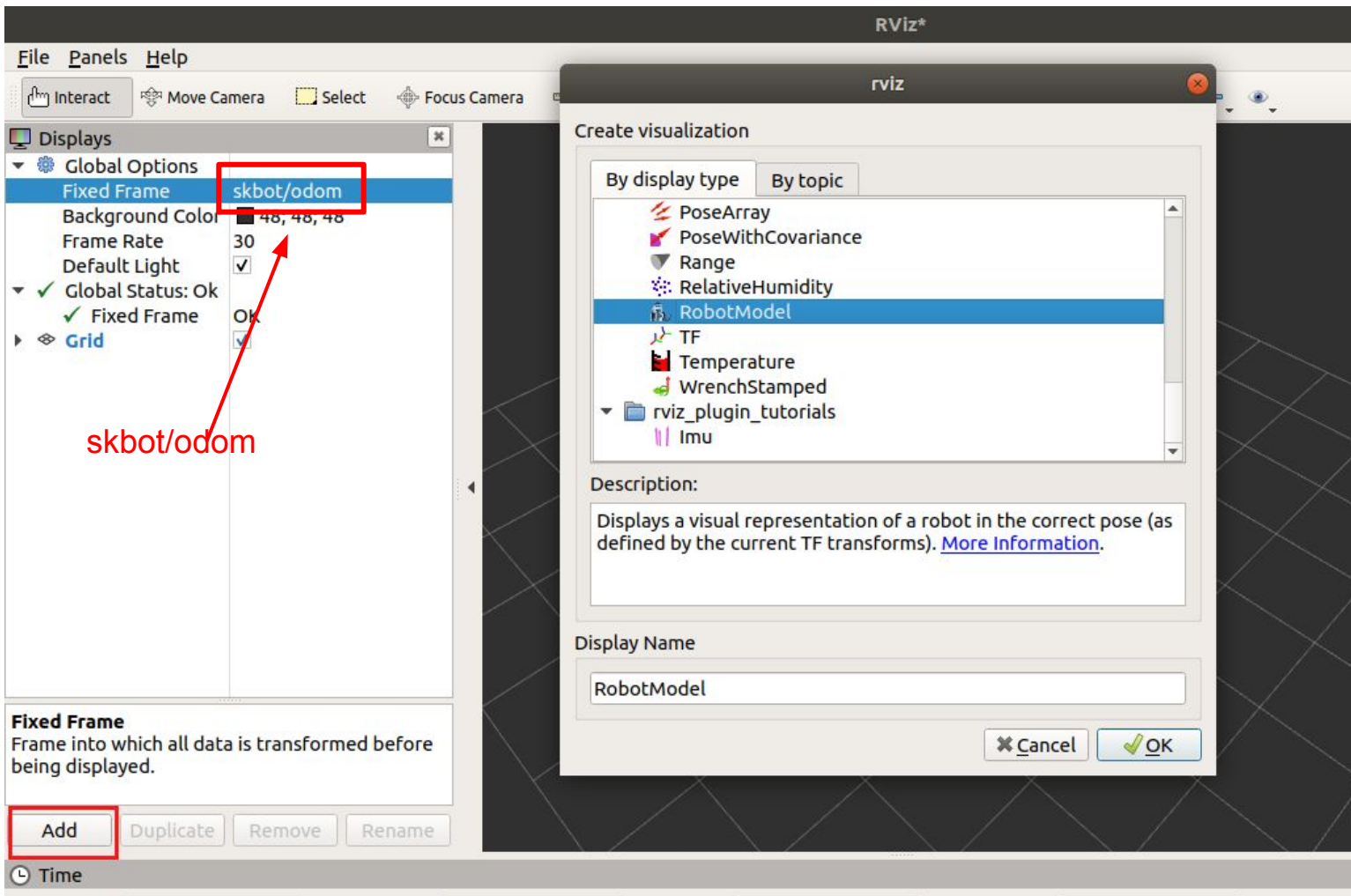
```
  <!-- Combine joint values -->
  <node name="robot_state_publisher"
pkg="robot_state_publisher" type="state_publisher"/>
```

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

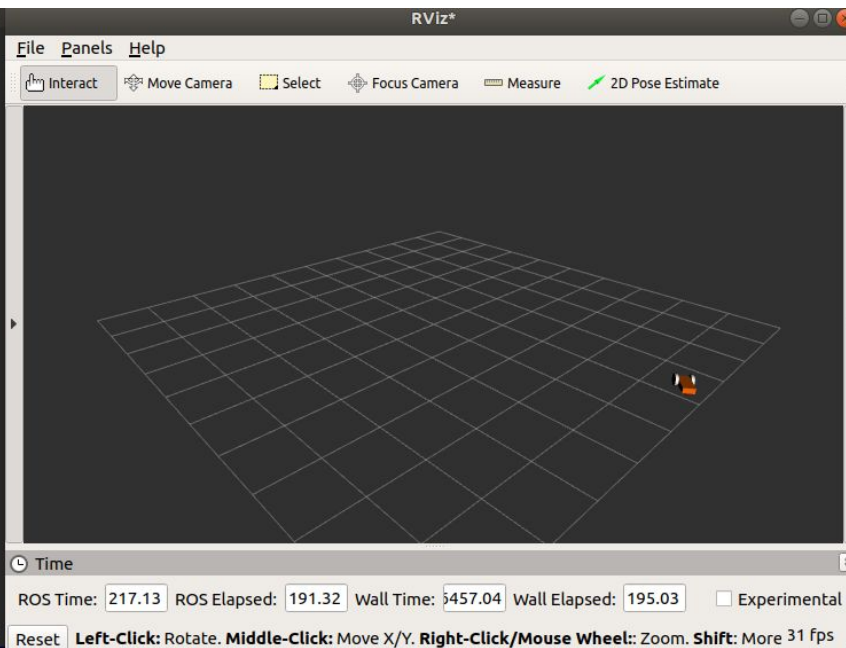
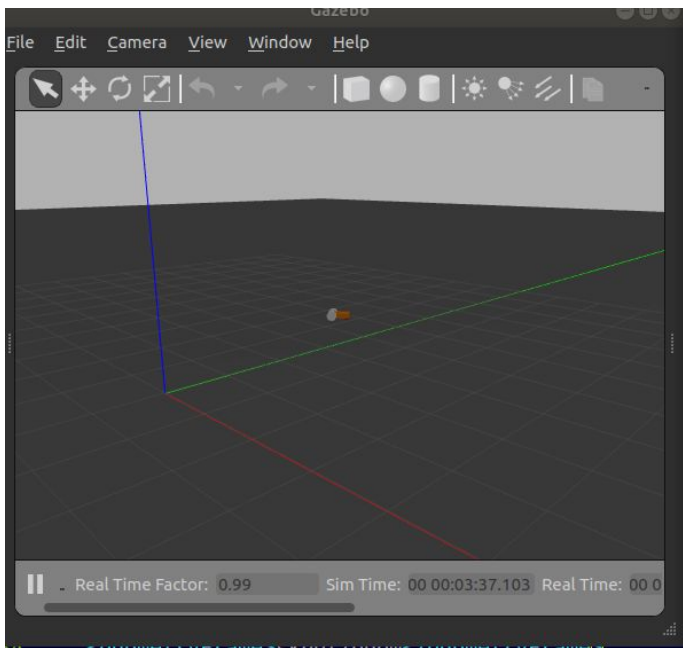
```
</launch>
```

File: skbot\_rviz.launch





Add Robot Model and select skbot/odom as the fixed frame.

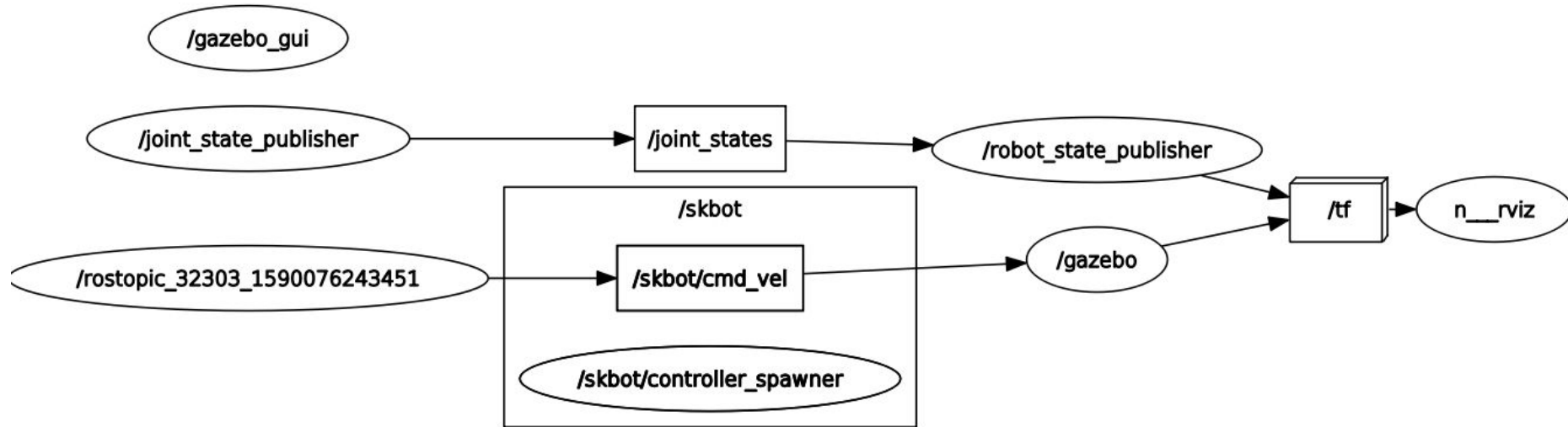


```
$ rostopic pub /skbot/cmd_vel geometry_msgs/Twist "linear:
  x: 0.2
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.1"
```

Robot should start moving in a circular trajectory both in Gazebo and Rviz.

- Using rqt\_graph we can visualize various nodes, topics and publishers.
- Note that joint\_state\_publisher (defined in rviz launch file) publishes joint\_states which are used by Rviz to update robot states.
- Gazebo receives the command velocity from a rostopic publisher started by the user.
- Rviz uses data obtained from robot state publisher and gazebo to update robot states.

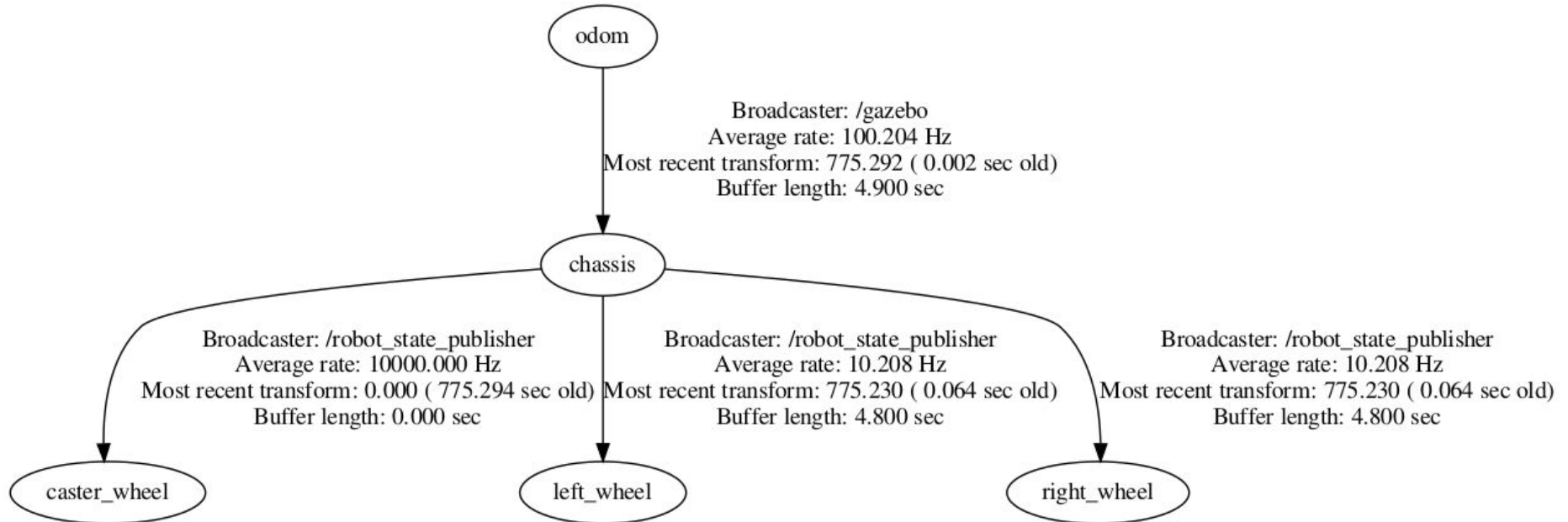
`$ rqt_graph`



view\_frames Result

Recorded at time: 775.294

```
$ rosrun tf view_frames  
$ evince frames.pdf
```



# Adding Sensors to the Robot

## Adding Camera Sensor to the Robot

- Add a camera related description, link and joint information in the main urdf file: “skbot.xacro”
- Add Gazebo related information and suitable camera plugin into the file: “skbot.gazebo”
- Add some objects to see through camera using <include> tags in the file: “skbot.world”
- Now reload the launch file “skbot\_world.launch”
- Use the default image\_view tool of ROS to view the scene as seen by the robot.
- Use the rotate tool of Gazebo GUI to rotate the robot towards object to view

```
$ rosrun image_view image_view image:=/skbot/camera1/image_raw
```

## File: skbot.xacro

```
<!-- Adding a Camera -->
<joint name="camera base" type="fixed">
  <parent link="chassis"/>
  <child link="camera" />
  <origin xyz="{chassisLength/2-cameraSize/2} 0 {chassisHeight+wheelRadius}" rpy="0 0 0" />
</joint>

<link name="camera">
  <collision>
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <geometry>
      <box size="{cameraSize} {cameraSize} {cameraSize}"/>
    </geometry>
  </collision>
  <visual>
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <geometry>
      <box size="{cameraSize} {cameraSize} {cameraSize}"/>
    </geometry>
    <material name="blue"/>
  </visual>
  <inertial>
    <mass value="{cameraMass}" />
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <box_inertia m="{cameraMass}" x="{cameraSize}" y="{cameraSize}" z="{cameraSize}" />
  </inertial>
</link>
</robot>
```

```
<gazebo reference="camera">
```

## File: skbot.gazebo

```
<material>Gazebo/Blue</material>
<sensor type="camera" name="camera1">
  <update_rate>30.0</update_rate>
  <camera name="head">
    <horizontal_fov>1.3962634</horizontal_fov>
    <image>
      <width>320</width>
      <height>240</height>
      <format>R8G8B8</format>
    </image>
    <clip>
      <near>0.02</near>
      <far>300</far>
    </clip>
  </camera>
  <plugin name="camera_controller" filename="libgazebo_ros_camera.so">
    <alwaysOn>true</alwaysOn>
    <updateRate>0.0</updateRate>
    <cameraName>skbot/camera1</cameraName>
    <imageTopicName>image_raw</imageTopicName>
    <cameraInfoTopicName>camera_info</cameraInfoTopicName>
    <frameName>camera_link</frameName>
    <hackBaseline>0.07</hackBaseline>
    <distortionK1>0.0</distortionK1>
    <distortionK2>0.0</distortionK2>
    <distortionK3>0.0</distortionK3>
    <distortionT1>0.0</distortionT1>
    <distortionT2>0.0</distortionT2>
  </plugin>
</sensor>
</gazebo> </robot>
```

## File: skbot.world

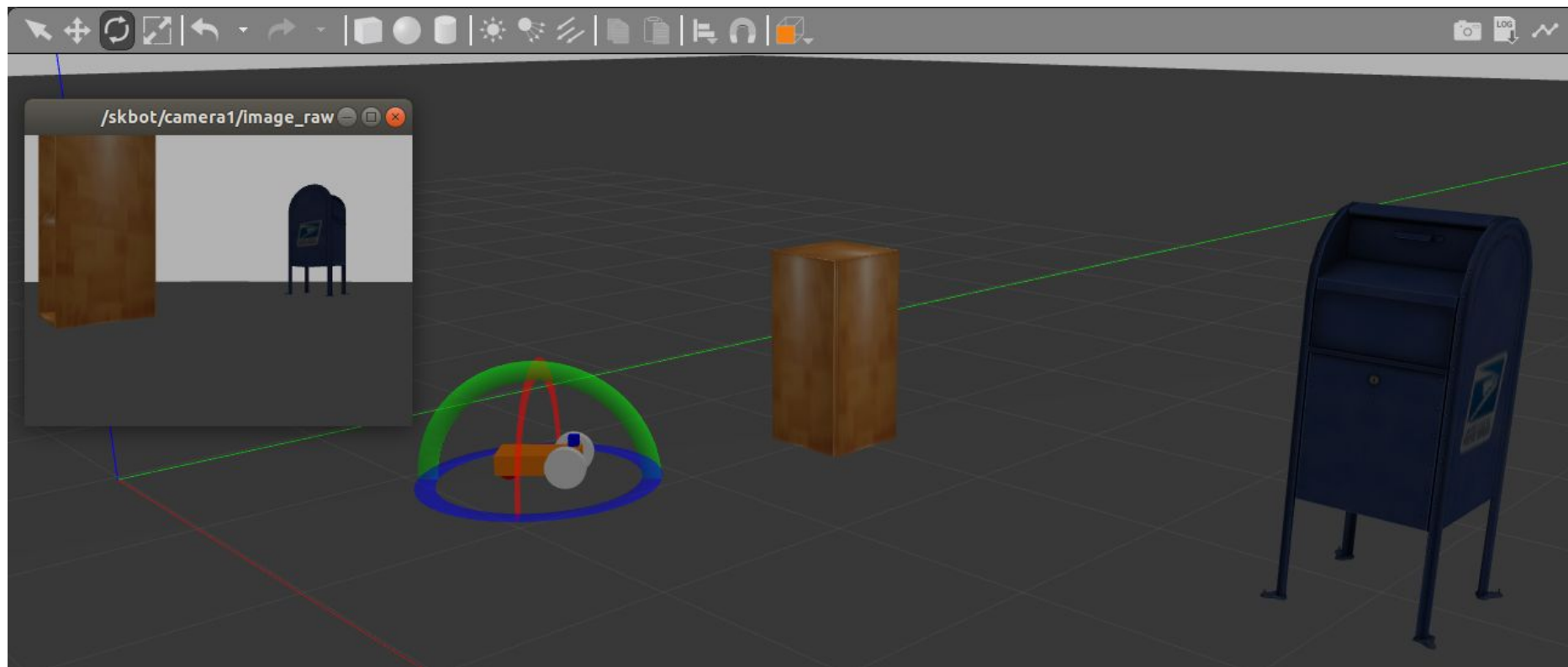
```
<include>
  <uri> model://cabinet </uri>
  <pose>2 3 0 0 0 0</pose>
  <static>true</static>
</include>

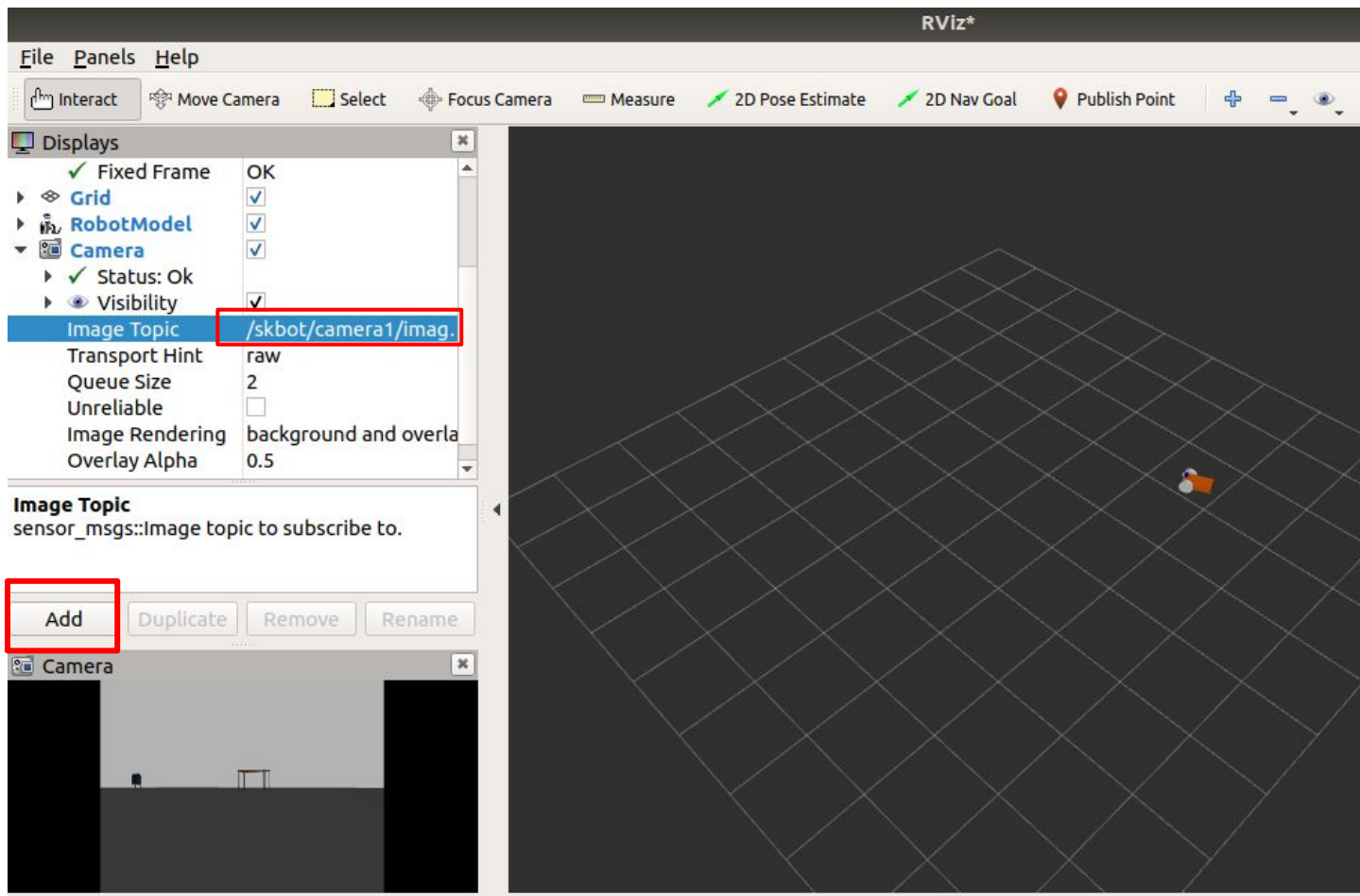
<include>
  <uri> model://postbox </uri>
  <pose>5 3 0 0 0 0 </pose>
  <static>true</static>
</include>
</world>
```

Many built-in objects are already available with Gazebo which can be loaded as shown above. An exhaustive list is available at [this link](#).



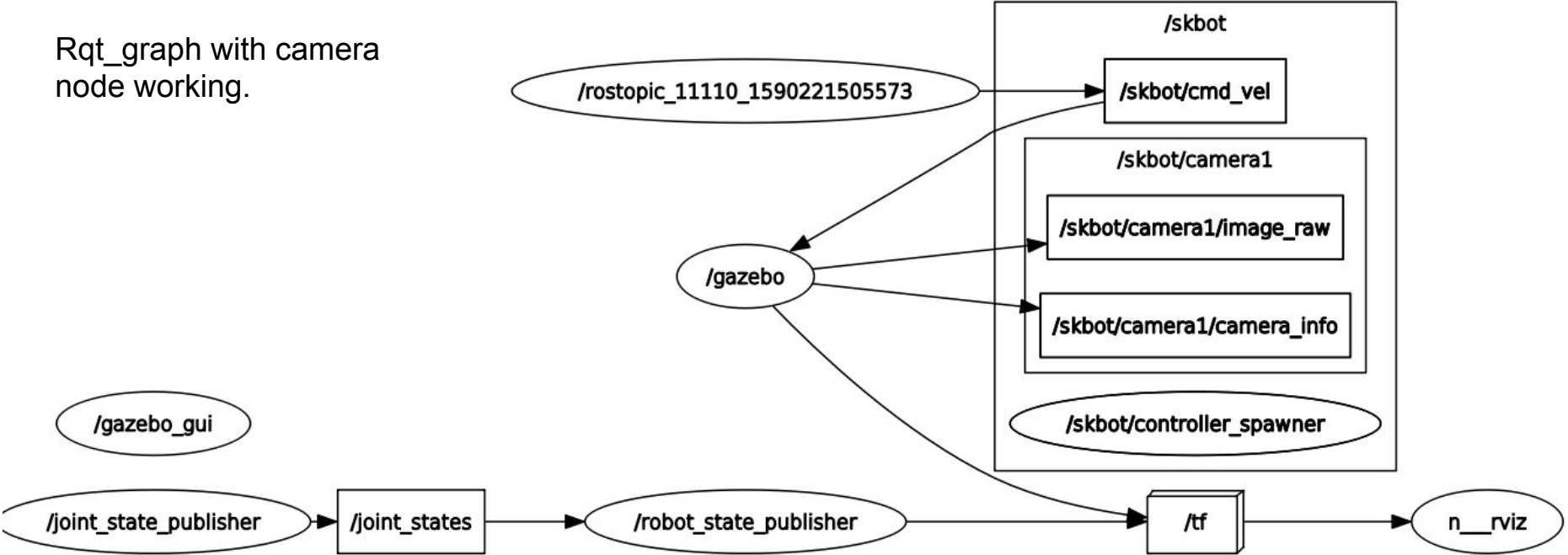
Seeing the world through Robot's Camera.



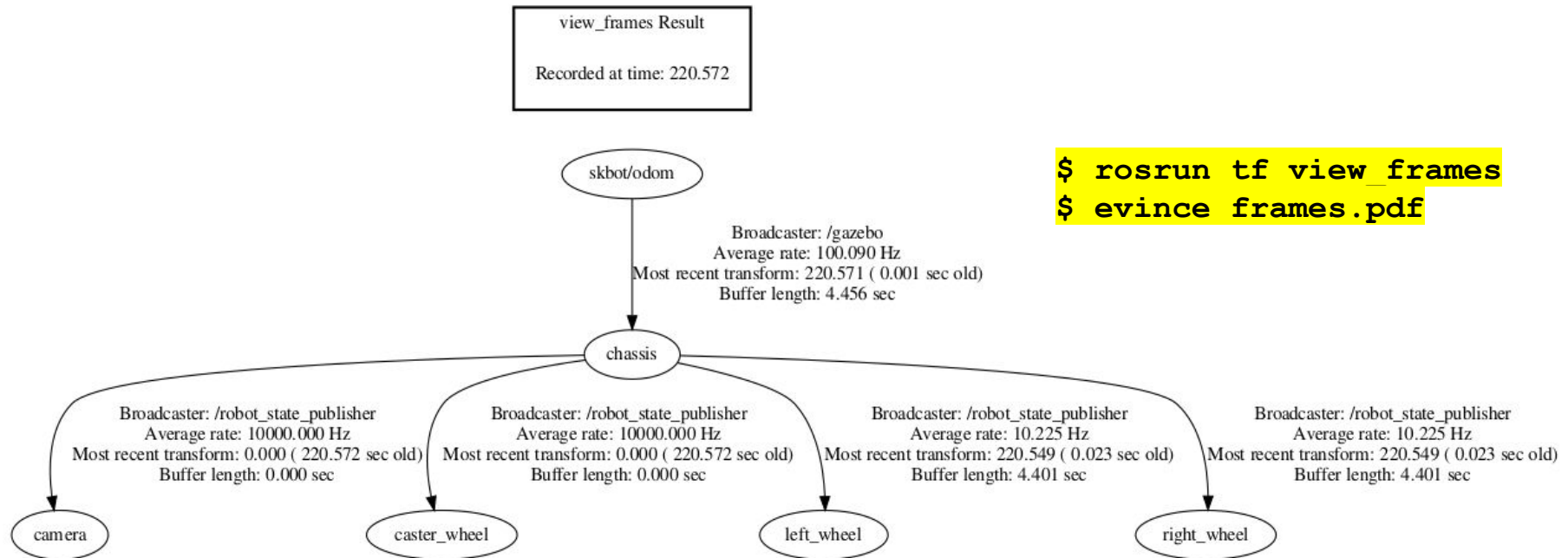


A camera can be similarly added. Change the image Topic to `/skbot/camera1/image_raw` to access the video feed of on-board camera.

Rqt\_graph with camera node working.



- Transform tree shows the geometric and physical relationship between various parts of the robot.
- skbot/odom provides the position of the chassis which is connected to all other parts like camera, caster\_wheel, left and right wheels.



## Adding a Laser Scan Sensor

- Modify “skbot.xacro” file to include the following descriptions:
  - Add a link named “hokuyo” corresponding to laser scanner to the robot.
  - Add a corresponding joint with “chassis” as the parent link and “hokuyo” as the child link.
- Modify “skbot.gazebo” file to include a gazebo reference tag and a gazebo plugin for this laser.
- You will need “hokuyo.dae” file provided by the manufacturers to be included in the package (/skbot\_description/meshes/)
- If you have a GPU, use gpu version of laser plugin library.
- More details on how to use various sensors is available at [this](#) link.

```
<!-- Adding a Laser -->
```

```
<joint name="hokuyo_joint" type="fixed">
```

```
<axis xyz="0 1 0" />
```

```
<origin xyz="{chassisLength/2-0.005} 0
```

```
{chassisHeight+wheelRadius}" rpy="0 0 0" />
```

```
<!--origin xyz=".15 0 .1" rpy="0 0 0"/-->
```

```
<parent link="chassis"/>
```

```
<child link="hokuyo"/>
```

```
</joint>
```

```
<!-- Hokuyo Laser -->
```

```
<link name="hokuyo">
```

```
<collision>
```

```
<origin xyz="0 0 0" rpy="0 0 0"/>
```

```
<geometry>
```

```
<box size="0.1 0.1 0.1"/>
```

```
</geometry>
```

```
</collision>
```

```
<visual>
```

```
<origin xyz="0 0 0" rpy="0 0 0"/>
```

```
<geometry>
```

```
<mesh
```

```
filename="package://skbot_description/meshes/hokuyo.dae"/>
```

```
</geometry>
```

```
</visual>
```

```
<inertial>
```

```
<mass value="1e-5" />
```

```
<origin xyz="0 0 0" rpy="0 0 0"/>
```

```
<inertia ixx="1e-6" ixy="0" ixz="0" iyy="1e-6" iyz="0"
```

```
izz="1e-6" />
```

```
</inertial>
```

```
</link>
```

```
</robot>
```

File: skbot.xacro

```

<!-- Adding a hokuyo Laser Scanner -->
<gazebo reference="hokuyo">
  <sensor type="ray" name="head_hokuyo_sensor">
    <pose>0 0 0 0 0 0</pose>
    <visualize>true</visualize>
    <update_rate>40</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>0.10</min>
        <max>30.0</max>
        <resolution>0.01</resolution>
      </range>
      <noise>
        <type>gaussian</type>
        <mean>0.0</mean>
        <stddev>0.01</stddev>
      </noise>
    </ray>
  </sensor>
</gazebo>

```

```

<plugin name="gazebo_ros_head_hokuyo_controller"
filename="libgazebo_ros_laser.so">
  <topicName>/skbot/laser_scan</topicName>
  <frameName>hokuyo</frameName>
</plugin>
</sensor>
</gazebo>

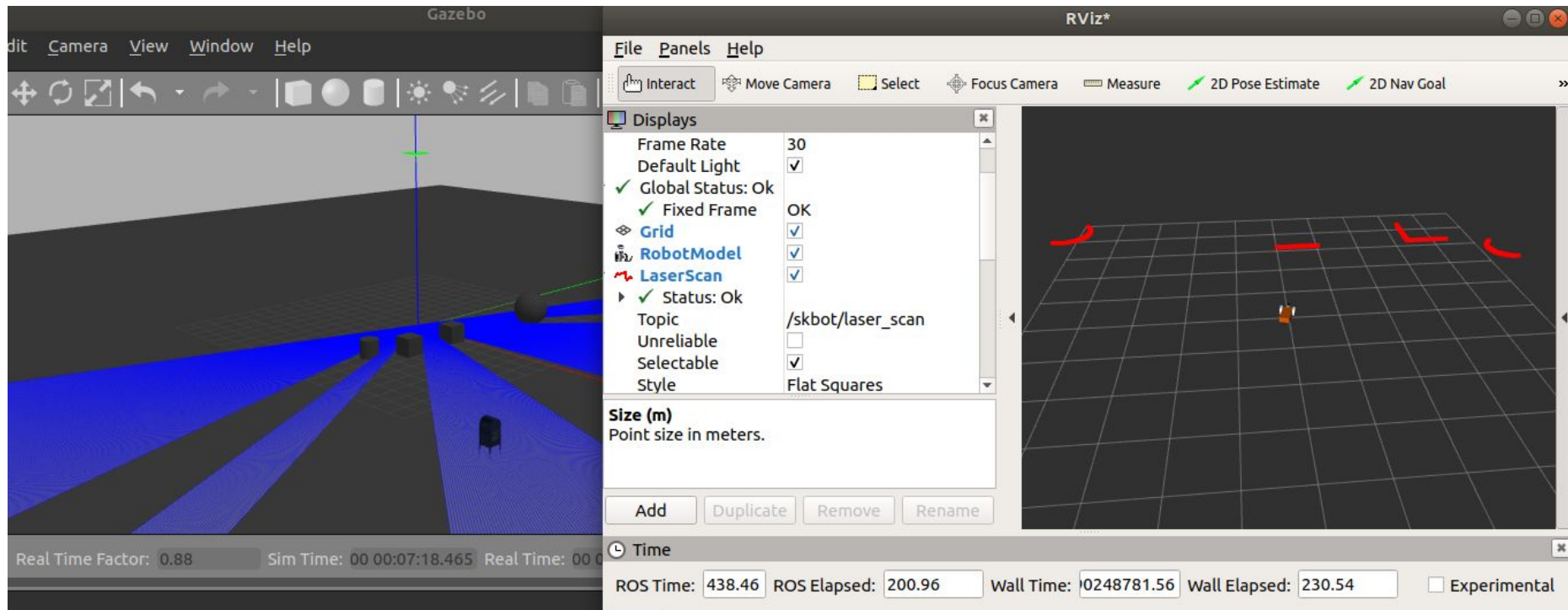
```

- If you have gpu, use “gpu\_ray” and “libgazebo\_ros\_gpu\_laser.so” instead.
- The laser data is published at topic /skbot/laser\_scan.
- Add a LaserScanner model in Rviz and subscribe to above topic to visualize laser scans

File: skbot.gazebo

```
$ roslaunch skbot_gazebo skbot_world.launch (terminal 1)
```

```
$ roslaunch skbot_description skbot_rviz.launch (terminal 2)
```





# Autonomous Navigation

# ROS navigation Stack with Turtlebot

- Github Turtlebot repository  
<https://github.com/turtlebot/turtlebot>
- On ROS-Melodic, you need to build it from source. The instructions are available here.
- You must activate the environment after installation

```
$ source ~/catkin_ws/devel/setup.bash
```

Or

```
$ source ~/catkin_ws/devel_isolated/setup.bash
```

- Step 1: Build the map using Gmapping

On Terminal 1:

```
$ roslaunch turtlebot_gazebo turtlebot_world.launch
```

On Terminal 2:

```
$ roslaunch turtlebot_gazebo gmapping_demo.launch
```

On Terminal 3:

```
$ roslaunch turtlebot_rviz_launchers view_navigation.launch
```

On Terminal 3:

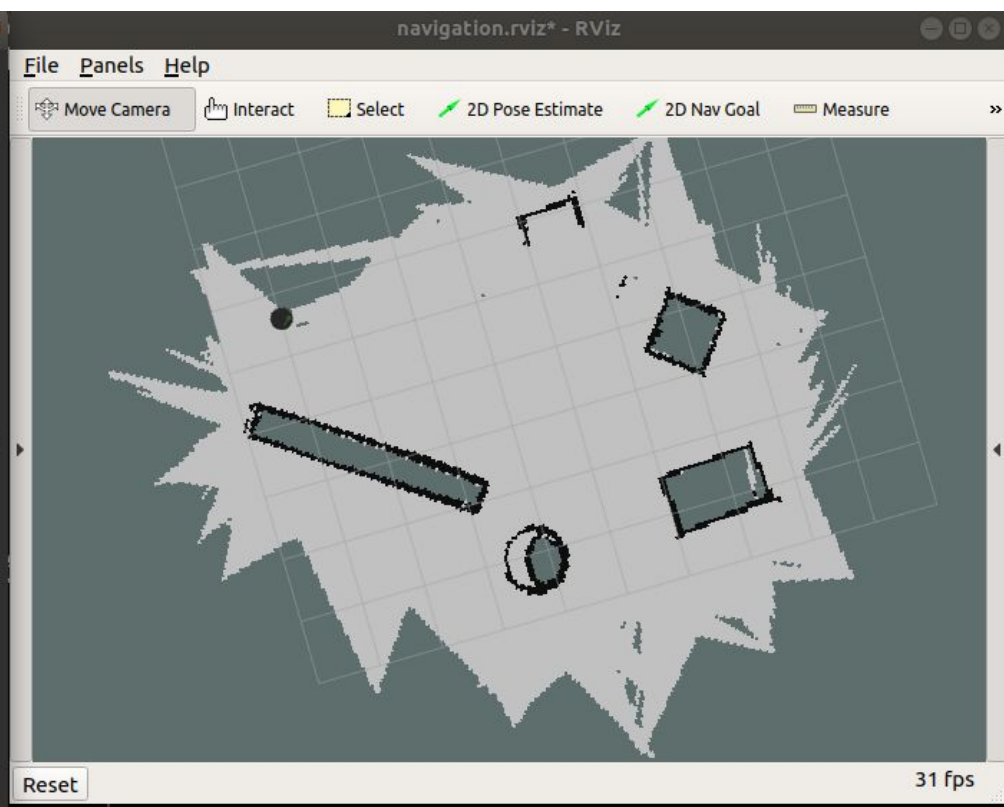
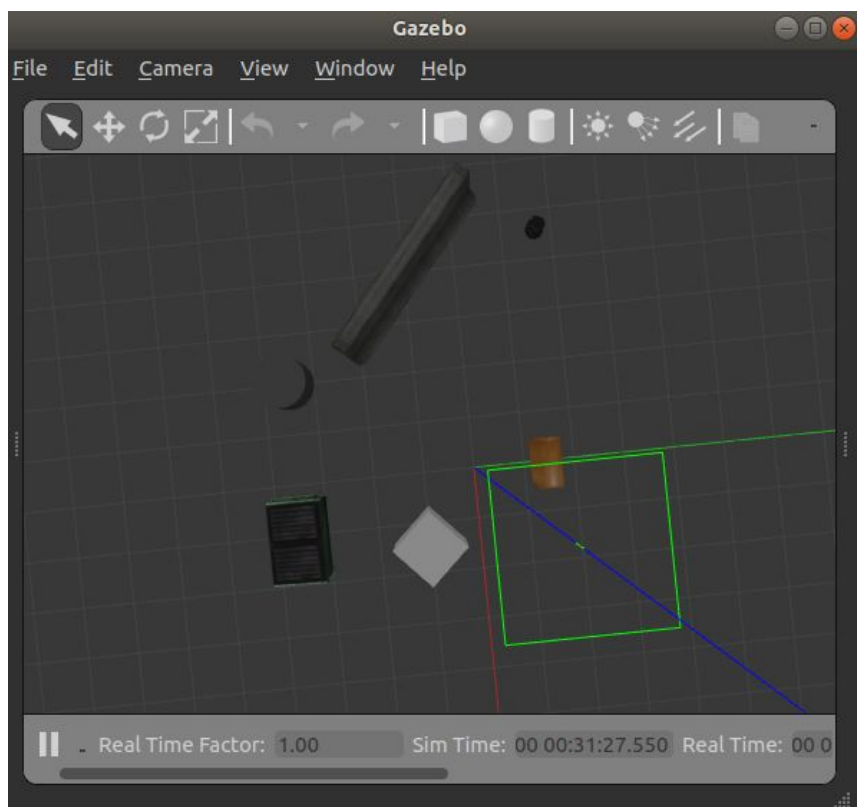
```
$ roslaunch turtlebot_teleop keyboard_teleop.launch
```

Now make the robot explore its environment by using keyboard and you can see the map developing on Rviz.

- Step 2: Save the map

```
$ rosrun map_server map_saver -f ~/catkin_ws/maps/test_map
```

It creates two files test\_map.png and test\_map.yaml within the above folder.



Step 3: Use the saved map to navigate autonomously

On terminal 1:

```
$ roslaunch turtlebot_gazebo turtlebot_world.launch
```

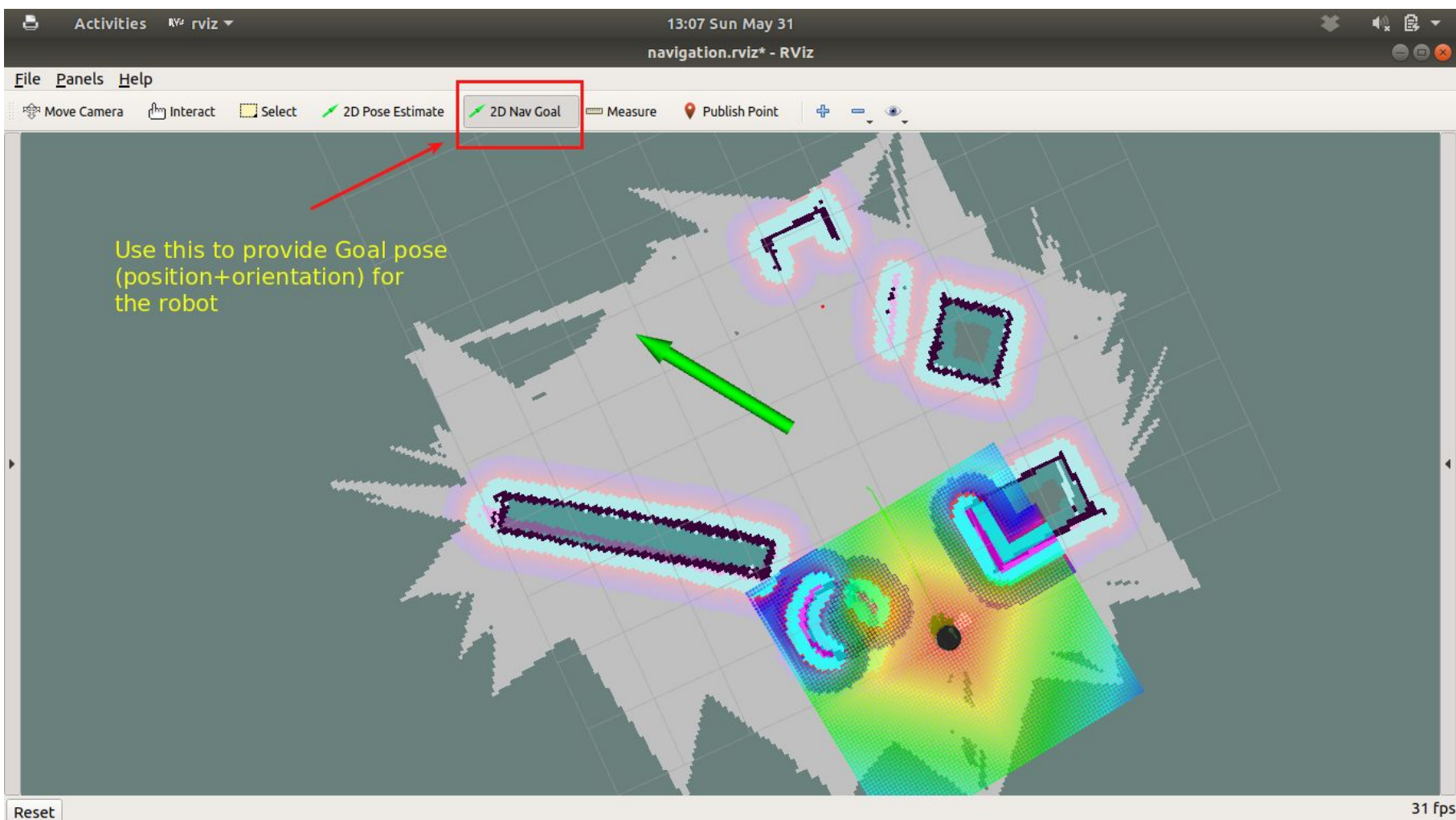
On Terminal 2:

```
$ roslaunch turtlebot_gazebo amcl_demo.launch  
map_file:=~/catkin_ws/maps/test_map.yaml
```

On Terminal 3:

```
$ roslaunch rviz_launchers view_navigation.launch
```

Use 2D Nav Goal tool to provide goal pose (position and orientation) for the robot. The AMCL planner generates a path (shown in red) for the robot to follow. Robot then follows this path autonomously by using a scan matching algorithm.





Move Camera



Interact



Select



2D Pose Estimate



2D Nav Goal



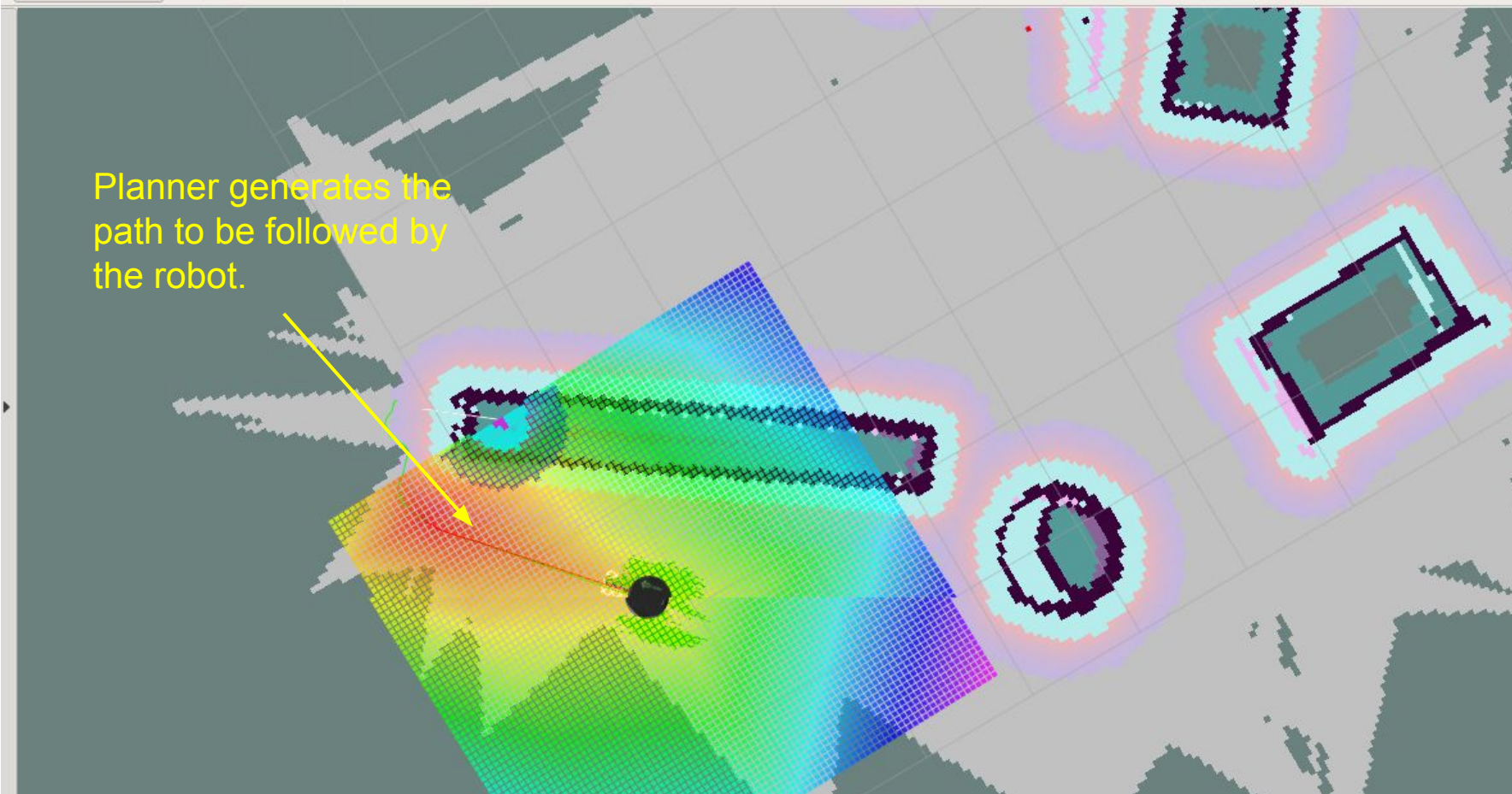
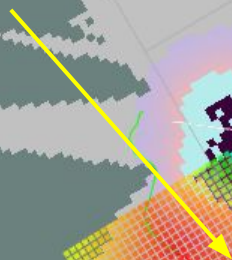
Measure



Publish Point



Planner generates the path to be followed by the robot.



# Autonomous Navigation with your own Robot

- Copy the 'playground.world' file from 'turtlebot\_gazebo' package to /skbot\_gazebo/worlds/ folder and rename it as 'turtlebot\_playground.world'.
- Modify the '/skbot\_gazebo/launch/skbot\_world.launch' to load this new world:

```
<include file="$(find gazebo_ros)/launch/empty_world.launch">
<!--arg name="world_name" value="$(find skbot_gazebo)/worlds/skbot.world"/-->
<arg name="world_name" value="$(find skbot_gazebo)/worlds/turtlebot_playground.world"/>
...
</include>
```

- Create a new catkin package named 'skbot\_navigation':

```
$ catkin_create_pkg skbot_navigation
```

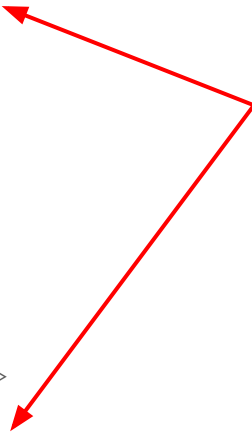


- Copy `/skbot_navigation/config/` folder containing various parameters files needed for tuning mapping functions (available with repo).
- Create a `/skbot_navigation/launch/` folder with the following files:
  - `gmapping_demo.launch`
  - `amcl_demo.launch`
  - `skbot_teleop.launch`
- First two files could be copied from `/turtlebot_navigation/launch'` folder while the last file could be copied from `/turtlebot_teleop/launch'` folder.
- We will also create the following custom rviz launch file in the folder `/skbot_description/launch/`
  - `skbot_rviz_gmapping.launch`

## /skbot\_navigation/launch/gmapping\_demo.launch

```
<?xml version="1.0"?>
<launch>
  <master auto="start"/>
  <param name="/use_sim_time" value="true"/>
  <!-- Run gmapping -->
  <node pkg="gmapping" name="slam_gmapping" type="slam_gmapping" output="screen">
    <param name="base_frame" value="chassis"/>
    <param name="odom_frame" value="/skbot/odom"/>
    <param name="delta" value="0.01"/>
    <param name="xmin" value="-20"/>
    <param name="xmax" value="20"/>
    <param name="ymin" value="-20"/>
    <param name="ymax" value="20"/>
    <remap from="scan" to="/skbot/laser/scan"/>
    <param name="base_frame" value="chassis" />
    <param name="linearUpdate" value="0.5"/>
    <param name="angularUpdate" value="0.436"/>
    <param name="temporalUpdate" value="-1.0"/>
    <param name="resampleThreshold" value="0.5"/>
    <param name="particles" value="80"/>
    <remap from="scan" to="/skbot/laser/scan"/>
  </node>
</launch>
```

Make sure, these variables  
point to right values.



/skbot\_navigation/launch/skbot\_teleop.launch

```
<?xml version="1.0"?>
<launch>
  <!-- turtlebot_teleop_key already has its own built in velocity smoother
-->
  <node pkg="turtlebot_teleop" type="turtlebot_teleop_key"
name="turtlebot_teleop_keyboard" output="screen">
    <param name="scale_linear" value="0.5" type="double"/>
    <param name="scale_angular" value="0.5" type="double"/>
    <remap from="turtlebot_teleop_keyboard/cmd_vel" to="/skbot/cmd_vel"/>
  </node>
</launch>
```

Requires turtlebot\_teleop package

## /skbot\_description/launch/skbot\_rviz\_gmapping.launch

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


  <param name="robot_description" command="$(find xacro)/xacro.py '$(find
skbot_description)/urdf/skbot.xacro'"/>

  <!-- send fake joint values -->
  <node name="joint_state_publisher" pkg="joint_state_publisher"
type="joint_state_publisher">
    <param name="use_gui" value="False"/>
  </node>

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

  <!-- Show in Rviz -->
  <node name="rviz" pkg="rviz" type="rviz" />
  <!--node name="rviz" pkg="rviz" type="rviz" args="-d $(find
skbot_description)/rviz/mapping.rviz"/-->
</launch>
```

It is possible to load a  
previously saved rviz file.



- Step 1: Build the map

On terminal 1:

```
$ roslaunch skbot_gazebo skbot_world.launch
```

On terminal 2:

```
$ roslaunch skbot_navigation gmapping_demo.launch
```

On terminal 3:

```
$ roslaunch skbot_navigation skbot_teleop.launch
```

On terminal 4:

```
$ roslaunch skbot_description skbot_rviz_gmapping.launch
```

Now use keyboard to make the robot explore its environment. The corresponding map develops on Rviz. Make sure to add components - RobotModel, Maps, LaserScan to Rviz.

- Step 2: Save the map

```
$ rosrund map_server map_saver -f ~/catkin_ws/maps/test_map
```

File Panels Help

Interact Move Camera Select

Displays

- Fixed Frame OK
- Grid ☒
- RobotModel ☒
- Map ☒
  - Status: Ok
  - Topic /map
  - Alpha 0.7
  - Color Scheme map
  - Draw Behind ☐
  - Resolution 0.01
  - Width 4000
  - Height 4000
  - Position -20; -20; 0
  - Orientation 0; 0; 1
  - Unreliable ☐
  - Use Timestamp ☐
- LaserScan ☒
  - Status: Ok
  - Topic /skbot/laser/scan
  - Unreliable ☐
  - Selectable ☒
  - Style Flat Squares

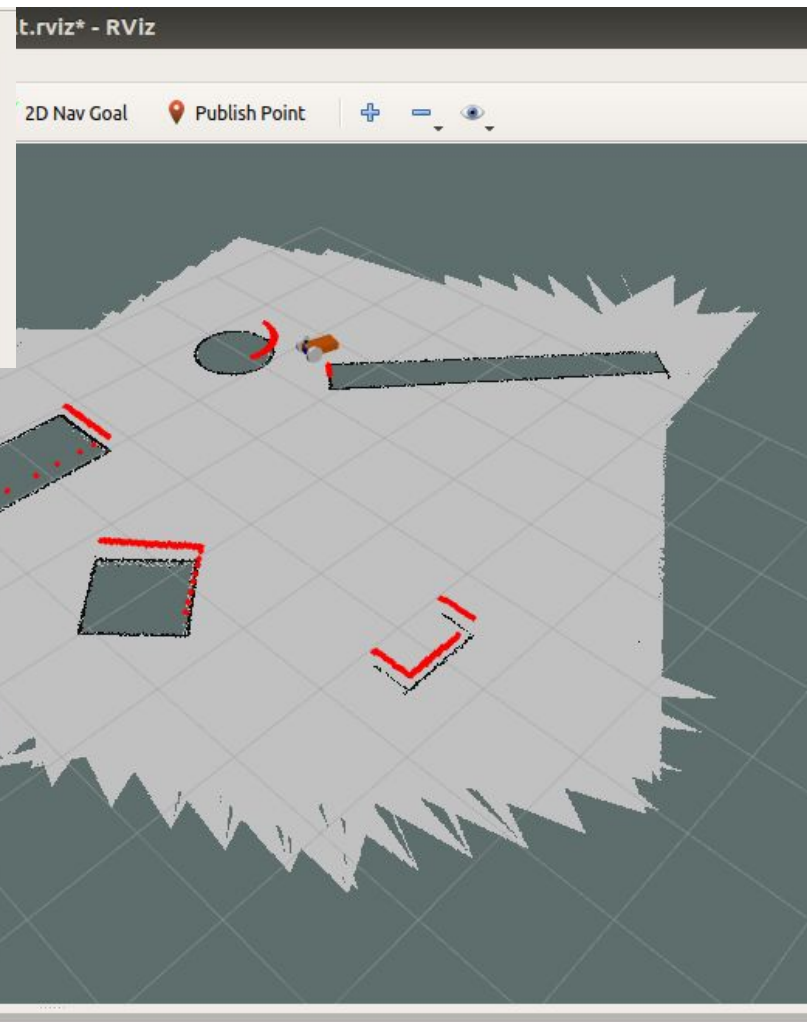
Alpha

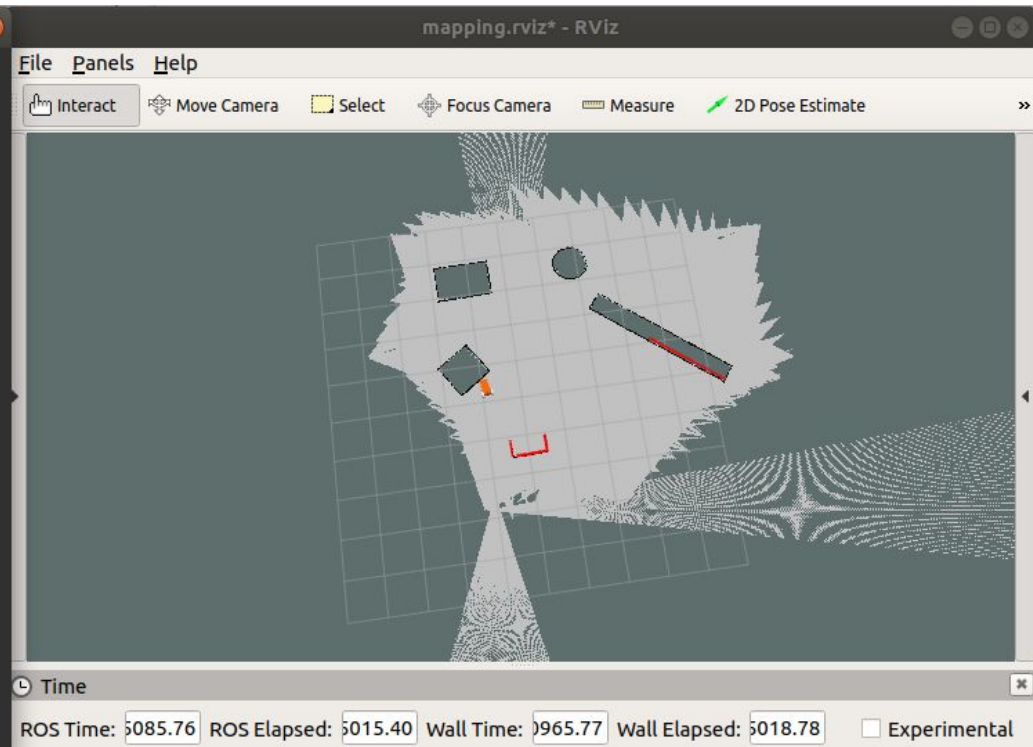
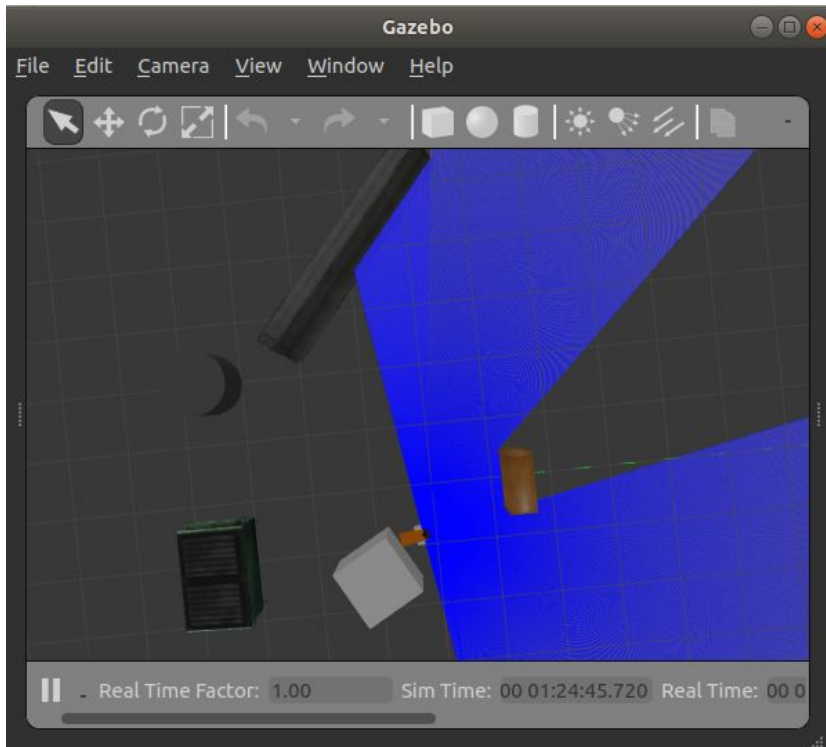
Amount of transparency to apply to the map.

Add Duplicate Remove Rename

Displays

- Global Options
  - Fixed Frame map
  - Background Color 48; 48; 48
  - Frame Rate 30
  - Default Light ☒
- Global Status: Ok
- Fixed Frame OK
- Grid ☒
- RobotModel ☒
- Map ☒
- LaserScan ☒





Add 'RobotModel', 'Map' and 'LaserScan' to Rviz. Point them to use suitable robot topics '/map' and '/skbot/laser/scan'. Use keyboard to move the robot around in the environment.

- Step 3: Use the saved map to navigate autonomously

After terminating all previous commands (Ctrl+C), rerun the following commands on separate terminals.

On terminal 1:

```
$ roslaunch skbot_gazebo skbot_world.launch
```

On terminal 2:

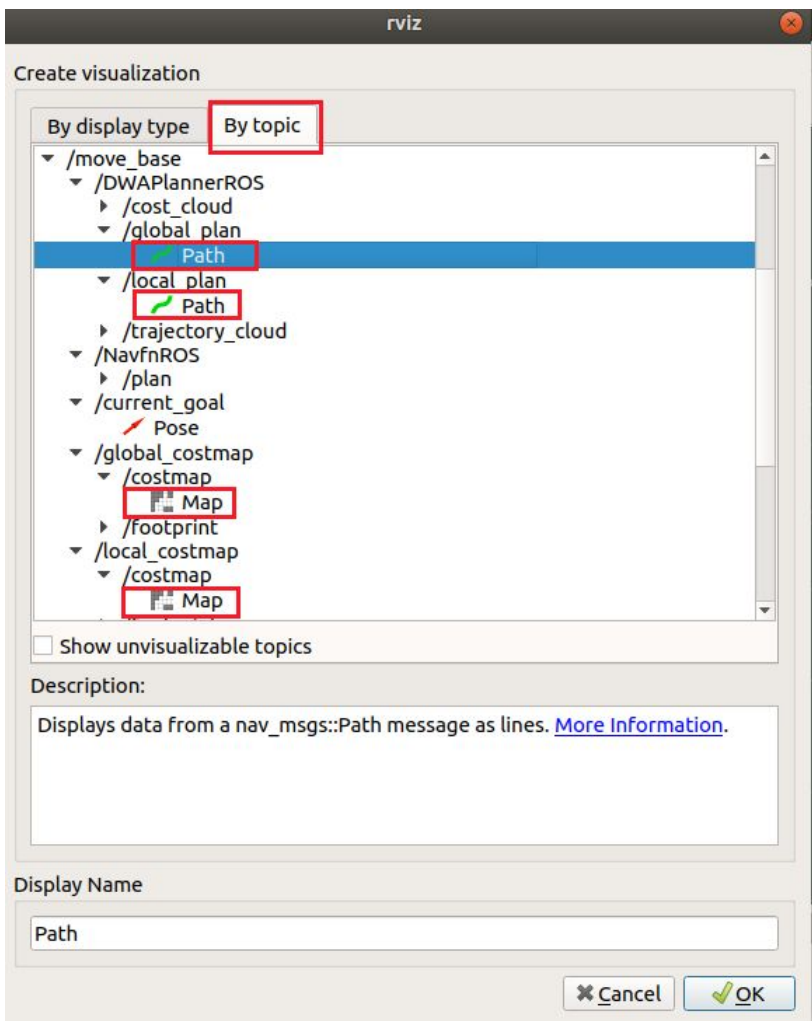
```
$ roslaunch skbot_navigation amcl_demo.launch  
map_file:=~/catkin_ws/maps/test_map.yaml
```

On terminal 3:

```
$ roslaunch skbot_description skbot_rviz_amcl.launch
```

- Use 2D Nav Goal to specify a goal pose for the robot. AMCL planner then finds a path for the robot.
- It would be necessary to tune in some config parameters files in folder /skbot\_navigation/config/ to get a correct behaviour from the robot. See the links [[here](#)] and [[here](#)]





Add following components into your display window:

- Map
- RobotModel
- LaserScan -> /skbot/laser/scan
- Global Path
- Local Path
- Global Costmap
- Local Costmap
- Cost Point cloud

It is easier to add by topic for some of the items. It is also possible to change the color for local / global costmaps, local and global trajectories generated by planner etc. Play around these variables.

ActivitiesRVZ rviz

20:49 Tue Jun 09

default.rviz\* - RViz

FilePanelsHelp

InteractMove CameraSelectFocus CameraMeasure2D Pose Estimate2D Nav GoalPublish Point

Displays

Topic/move\_base/DWAPla...

Unreliable

Line StyleLines

Color92; 53; 102

Alpha1

Buffer Length1

Offset0; 0; 0

Pose StyleNone

Map

Status: Ok

Topic/move\_base/local\_co...

Alpha0.7

Color Scheme**costmap**

Draw Behind

Resolution0.02

Width150

Height150

Position-3.18; -2.06; 0

Orientation0; 0; 1

Unreliable

Use Timestamp

Color Scheme

How to color the occupancy values.

AddDuplicateRemoveRename

Views

Type:Orbit (rviz)Zero

Current View

Near Clip ...0.01

Invert Z Axis

Target Fra...<Fixed Frame>

Distance10

Focal Shap...0.05

Focal Shap...✓

Yaw0.785398

Pitch0.785398

Focal Point0; 0; 0

SaveRemoveRename


Time

ROS Time:245.89ROS Elapsed:234.30Wall Time:1591732156.88Wall Elapsed:234.57

Reset

Experimental

31 fps



Provide 2D Goal pose for the robot using 2D Nav Goal tool.

Displays

Style

Flat Squares

Size (m)

0.05

Alpha

1

Decay Time

0

Position Transf...

XYZ

Color Transfor...

Intensity

Queue Size

10

Channel Name

intensity

Use rainbow

☒

Invert Rainbow

☐

Min Color

☒ 0; 0; 0

Max Color

☐ 255; 255; 255

Autocompute l...

☒

Min Intensity

999999

Max Intensity

-999999

☒ Map

☒ Path

☒ Path

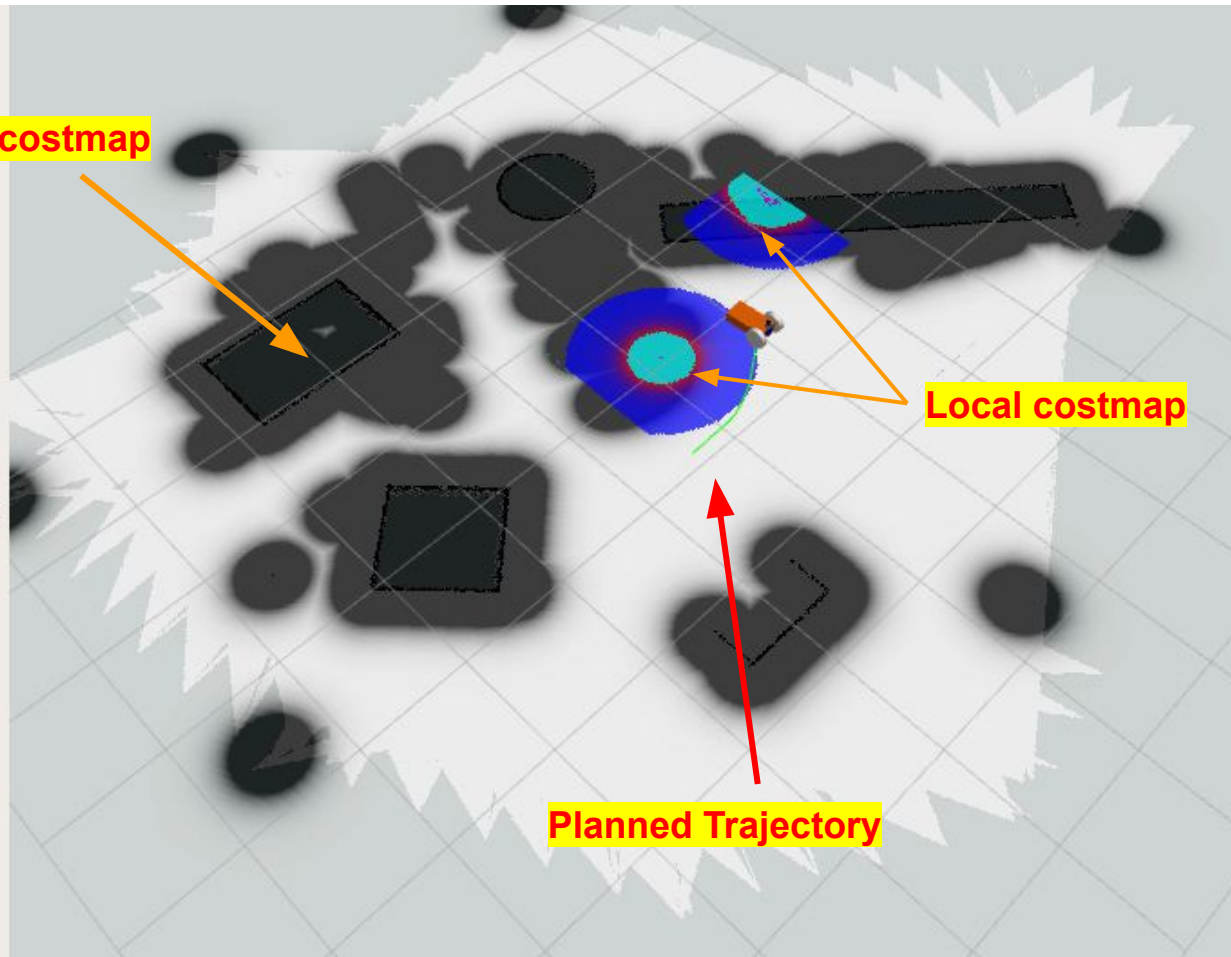
☒ Map

☐ PoseArray

☒ Map

se Timestamp

se map header timestamp when transforming

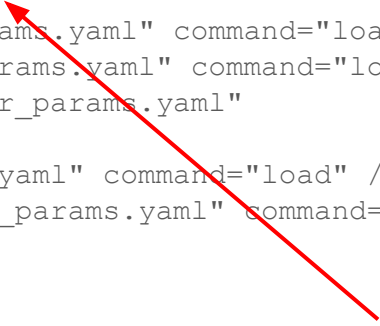


## File: /skbot\_navigation/launch/amcl\_demo.launch

```
<?xml version="1.0"?>
<launch>
  <master auto="start"/>
  <!-- Map server -->
  <arg name="map_file" default="$(find
skbot_navigation)/maps/test_map.yaml" />
  <node name="map_server" pkg="map_server" type="map_server" args="$(arg
map_file)" />
  <!-- Place map frame at odometry frame -->
  <node pkg="tf" type="static_transform_publisher"
name="map_odom_broadcaster"
  args="0 0 0 0 0 0 map odom 100"/>
  <!-- Localization -->
  <node pkg="amcl" type="amcl" name="amcl" output="screen">
    <remap from="scan" to="/skbot/laser/scan" />
    <param name="odom_frame_id" value="/skbot/odom"/>
    <param name="odom_model_type" value="diff-corrected"/>
    <param name="base_frame_id" value="chassis"/>
    <param name="update_min_d" value="0.5"/>
    <param name="update_min_a" value="1.0"/>
  </node>
```

.... contd.

```
<!-- Move base -->
<node pkg="move_base" type="move_base" respawn="false" name="move_base" output="screen">
  <param name="base_local_planner" value="dwa local planner/DWAPlannerROS"/>
  <rosparam file="$(find skbot_navigation) /config/costmap_common_params.yaml" command="load"
ns="global_costmap" />
  <rosparam file="$(find skbot_navigation) /config/costmap_common_params.yaml" command="load"
ns="local_costmap" />
  <rosparam file="$(find skbot_navigation)/config/local_costmap_params.yaml" command="load" />
  <rosparam file="$(find skbot_navigation)/config/global_costmap_params.yaml" command="load" />
  <rosparam file="$(find skbot_navigation)/config/base_local_planner_params.yaml"
command="load" />
  <rosparam file="$(find skbot_navigation)/config/move_base_params.yaml" command="load" />
  <rosparam file="$(find skbot_navigation)/config/dwa_local_planner_params.yaml" command="load"
/>
  <remap from="cmd_vel" to= "/skbot/cmd_vel"/>
  <remap from="odom" to= "/skbot/odom"/>
  <remap from="scan" to= "/skbot/laser/scan"/>
  <param name="move_base/DWAPlannerROS/yaw_goal_tolerance" value="1.0"/>
  <param name="move_base/DWAPlannerROS/xy_goal_tolerance" value="1.0"/>
</node>
</launch>
```



AMCL performance depends on these parameter files. These parameters should be tuned properly to get best performance.

## File: /skbot\_description/launch/skbot\_rviz\_amcl.launch


```
<?xml version="1.0"?>
<launch>
  <param name="robot_description" command="$(find xacro)/xacro.py '$(find
skbot_description)/urdf/skbot.xacro'"/>

  <!-- send fake joint values -->
  <node name="joint_state_publisher" pkg="joint_state_publisher"
type="joint_state_publisher">
    <param name="use_gui" value="False"/>
  </node>

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

  <!-- Show in Rviz -->
  <!--node name="rviz" pkg="rviz" type="rviz" args="-d $(find
skbot_description)/rviz/amcl.rviz"/-->

  <node name="rviz" pkg="rviz" type="rviz" />
</launch>
```



Custom Rviz file could be loaded here.

Following files are included in the `/skbot_navigation/config` folder:

```
base_local_planner_params.yaml  
global_costmap_params.yaml  
costmap_common_params.yaml  
local_costmap_params.yaml  
dwa_local_planner_params.yaml  
move_base_params.yaml
```

More discussion on how to tune these parameters are available at the following links:

- [Link1](#)
- [Link2](#)

# Other Resources

- [Richard Wang's Youtube Channel](#)
- [Devansh's Github page](#)