Gazebot Tutorial: https://www.generationrobots.com/blog/en/robotic-simulation-scenarios-with-gazebo-and-ros/

Space Robot simulation with Gazebo and ROS 'by Christidi Olga '

This tutorial is intended for members of CSL Space Group that want to have realistic simulations of their scenarios for space robots moving in plane, for example Cepheus. Although, with some changes in codes, everyone who wants to simulate his own robot can find it very helpful. Gazebo is a 3D simulator, while ROS serves as the interface for the robot. Combining both, results in a powerful robot simulator.

With Gazebo you are able to create a 3D scenario on your computer with robots, obstacles and many other objects. Gazebo also uses a physical engine for illumination, gravity, inertia, etc. You can evaluate and test your robot in difficult or dangerous scenarios without any harm to your robot. Most of the time it is faster to run a simulator instead of starting the whole scenario on your real robot.

Originally, Gazebo was designed to evaluate algorithms for robots. For many applications, it is essential to test your robot application, like error handling, battery life, localization, navigation and grasping. As there was a need for a multi-robot simulator, Gazebo was developed and improved.

This Tutorial was tested with an Ubuntu 16.04 with ROS Kinetic and Gazebo 7.

Installation of Gazebo

If you have installed ROS Kinetic, you have Gazebo 7 already installed.

First Steps with Gazebo and ROS

If you have installed ROS Kinetic, you are ready to discover the fascinating world of simulation. In this tutorial we are going to:

- setup a ROS workspace
- create projects for your simulated robot
- create a Gazebo world
- create your own robot model
- connect your robot model to ROS

Setup a new workspace

I’ll assume that you start from scratch and need to create a new workspace for your project. Let’s first source our ROS Kinetic environment:

```
source /opt/ros/kinetic/setup.bash
```

Now let’s create the folder that will contain our workspace and the ‘src’ subfolder.

```
mkdir ~/catkin_ws/src
```

Go into the source and initialize the workspace:

```
cd ~/catkin_ws/src

catkin_init_workspace
```

Lets do a first build of your (empty) workspace just to generate the proper setup files:

```
cd ..

catkin_make
```

From now on, each time we’ll have to start ROS commands that imply using our packages, we’ll have to source the workspace environment in each terminal:

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

Create projects for your simulated space robot

The ROS community has established some conventions for packages that define robots. We’ll try to follow these conventions for this simple space robot. Let’s assume our space robot will be called cepheus. We are going to create 3 packages:

cepheus_gazebo: provides launch files and worlds for easy starting of simulation

cepheus_description: provides the 3D model of the robot and the description of joints

cepheus_control: configures the ROS interface to our robot’s joints

Ok, let’s create these. First make sure you’ve sourced your workspace environment and go into the ‘src’ subfolder:

```
cd ~/catkin_ws/src
```

Let’s create the three packages:

```
catkin_create_pkg cepheus_gazebo gazebo_ros

catkin_create_pkg cepheus_description

catkin_create_pkg cepheus_control
```

Creating your own World

Let’s start with the gazebo package, go in there and create the following subfolders:

```
cd cepheus_gazebo

mkdir launch worlds
```

At first we want to create a world for our gazebo server. Therefore we switch to our worlds directory and create a new world file.

```
cd worlds

gedit cepheus.world
```

A basic world file defines at least a name:

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

<sdf version="1.4">

<world name="myworld">

</world>

</sdf>
```

Press Ctr+S to save files. Here you could directly add models and objects with their position. Also the laws of physics may be defined in a world. This is an important step to understand, because in this file you could also attach a specific plugin to an object. The plugin itself contains ROS and Gazebo specific code for more complex behaviors. At first we just want to add some basic objects, like a ground and a basic illumination source inside the world tag:

```
<include>

<uri>model://sun</uri>
```

```
</include>

<include>

<uri>model://ground_plane</uri>

</include>
```

As the ground plane and the sun are basic models that are also on the gazebo server they will be downloaded on startup if they cannot be found locally. If you want to know which objects are available on the gazebo server, take a look at Gazebo model database. To start the gazebo server there are several methods. As it is a good practice to use a launch file, we will create one now. This could later also be used for multiple nodes. Change to the launch directory of your project:

```
cd ~/catkin_ws/src/cepheus_gazebo/launch
```

Create a new file:

```
gedit cepheus_world.launch
```

and insert:

```
<launch>

<include file="$(find gazebo_ros)/launch/empty_world.launch">

<arg name="world_name" value="$(find cepheus_gazebo)/worlds/cepheus.world"/>

<arg name="gui" value="true"/>

</include>

</launch>
```

This launch file will just execute a default launch file provided by Gazebo, and tell it to load our world file and show the Gazebo client. You can launch it by doing:

```
roslaunch cepheus_gazebo cepheus_world.launch
```

Now you should see the gazebo server and the gui starting with a world that contains a ground plane and a sun (which is not obviously visible without objects). If not, it can be that there are some connections problems with the server. If that happens, start gazebo without the launch file, go to the models, you should find the one you want in the server. Click on them, they will be put in the cache. Now, if you close gazebo and start it again with the launch file, it should work. If you would press “Save world as” in “File” in your gazebo client and save the world in a text file, you could investigate the full world description.

Creating your own Model

The more accurate you want to model your space robot, the more time you need to spend on the design. We could put our model as a SDF file in the ~/.gazebo/models directory, this is the standard way when you work only with Gazebo. However with ROS we’ll prefer to use a URDF file generated by Xacro and put it the description package.

The Universal Robotic Description Format (URDF) is an XML file format used in ROS as the native format to describe all elements of a robot. Xacro (XML Macros) is an XML macro language. It is very useful to make shorter and clearer robot descriptions.

Ok. First, press Ctr+C to stop gazebo. So, now we need to go into our description package and create the urdf subfolder and the description file:

```
roscd cepheus_description

mkdir urdf

cd urdf

gedit cepheus.xacro
```

Xacro Concepts

• xacro:include: Import the content from other file. We can divide the content in different xacros and merge them using xacro:include.
• property: Useful to define constant values using ${property_name}
• xacro:macro: Macro with variable values. Later, we can use this macro from another xacro file, and we specify the required value for the variables. To use a macro, you have to include the file where the macro is, and call it using the macro’s name and filling the required values.

This file will be the main description of our robot. Let’s put some basic structure:

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

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



</robot>
```

The structure is basic for a urdf file. The complete description (links, joints, transmission…) have to be within the robot tag. The xmlns:xacro="http://www.ros.org/wiki/xacro" specifies that this file will use xacro. If you want to use xacro you have to put this. With xacro, you can define parameters. Once again, this make the file clearer. They are usually put at the beginning of the file (within the robot tag, of course). We will also include two files:

```
 <xacro:include filename="$(find cepheus_description)/urdf/cepheus.gazebo" />

 <xacro:include filename="$(find cepheus_description)/urdf/materials.xacro" />
```

These three correspond respectively to:

- all the gazebo-specific aspects of our robot
- definition of the materials used (mostly colors)
- definitions of some macros for easier description of the robot

Every file has to contain the robot tag and everything we put in them should be in this tag.

Now, we need to go into urdf folder in cepheus_description package and create the meshes subfolder:

```
roscd cepheus_description/urdf

mkdir meshes
```

Download the stl files provided in meshes folder here (https://github.com/ntua-cslep/cepheus_space_robot) and paste them in your meshes folder.

Now we want to add Cepheus base. Insert this within the robot tag of cepheus.xacro:

```
<link

    name="cepheus">

    <inertial>

      <origin

        xyz="0.0024 0.02937 0.15711"

        rpy="0 0 0" />

      <mass

        value="7.11891369052411" />

      <inertia

        ixx="0.0937165866660239"

        ixy="-0.004838152027527"
        ixz="-0.00397891305009112"

        iyy="0.0905199286322999"

        iyz="-0.00257712868335304"

        izz="0.0755866683160163" />

    </inertial>

    <visual>

      <origin

        xyz="0 0 0"

        rpy="0 0 0" />

      <geometry>

        <mesh

          filename="package://cepheus_description/urdf/meshes/cepheus.STL" />

      </geometry>

      <material name="LightGrey"/>

    </visual>

    <collision>

      <origin

        xyz="0 0 0.215"

        rpy="0 0 0" />

      <geometry>

         <cylinder length="0.43" radius="0.2"/>

      </geometry>

    </collision>

  </link>
```

As you can see, we have three tags for this one link, where one is used to the collision detection engine, one to the visual rendering engine and the last to the physic engine. Most of the time they are the same, except when you have complicated and beautiful visual meshes. As they don’t need to be that complicated for collision detection, you could use a simple model for the collision. The material element in the visual tag refer to a color that must be defined in the materials.xacro and referred to in the cepheus.gazebo file, at least, in the structure we adopted. Create in urdf folder this file:

```
gedit cepheus.gazebo
```

Add this in “cepheus.gazebo”:

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

<gazebo reference="cepheus">
  <material>Gazebo/Grey</material>
</gazebo>

</robot>
```

Create in urdf folder this file:

```
gedit materials.xacro
```

And this in the “materials.xacro”:

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

<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="brown">

  <color rgba="${222/255} ${207/255} ${195/255} 1.0"/>

</material>


<material name="red">

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

</material>


<material name="white">

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

</material>


</robot>
```

As you can see, we add more than just the color we wanted, this is for convenience. Now, we can leave this file alone and use any color we want. We have two small things to do before testing our model with gazebo.

The physic engine does not accept a base_link with inertia. It is then useful to add a simple link without inertia and make a joint between it and cepheus link. Add this, before the cepheus link in the cepheus.xacro file :

```
<link name=“world” />


  <joint name="world_joint" type="continuous">

    <axis xyz="0 0 1"/>
    <parent link="world"/>

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

    <child link="cepheus"/>

    <dynamics damping="0" friction="0"/>

  </joint>
```

Type of joint continuous means a continuous hinge joint that rotates around the axis and has no upper and lower limits. This is not the best choice. The best would be to have a planar joint, since Cepheus can move in plane. In urdf files, exist the joint type planar. Although, in gazebo, is not valid and I haven’t found a better solution yet. In order to start gazebo with our model, we have to modify the previously created launch file cepheus_world.launch by adding the following two tags in the launch tag:

```
<param name="robot_description" command="$(find xacro)/xacro '$(find
cepheus_description)/urdf/cepheus.xacro'" />
```

```
<node name=“cepheus_spawn" pkg="gazebo_ros" type="spawn_model" output="screen"

 args="-urdf -param robot_description -model cepheus” />
```

The first tag will first call the xacro script to convert of xacro description into an actual URDF. This URDF is then inserted into a ROS parameter called “robot_description” (this is a standard name used by many ROS tools). The second tag launches a program from the gazebo_ros package that will load the URDF from the parameter “robot_description” and spawn the model into our Gazebo simulator. If you launch your project with this launch file, the gazebo client opens and the base of Cepheus should be there. As a next step we add the reaction wheel to the Cepheus. Add this after the cepheus link in the main urdf file :

```
<joint name="reaction_wheel_joint" type="continuous">

    <parent link="cepheus"/>

    <child link="reaction_wheel"/>

    <origin xyz="0.065 0 0.03815" rpy="0 0 0"/>

    <axis xyz="0 0 1" />

</joint>



  <link name="reaction_wheel">

    <visual>

      <material name="silver"> <color rgba="0.5 0.5 0.5 1"/> </material>
```

```
<geometry>

        <mesh

filename="package://cepheus_description/urdf/meshes/reaction_wheel.STL" />

      </geometry>

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

    </visual>

  <inertial>

    <origin xyz="0 0 0.025" rpy="0 0 0" />

     <mass value=“1”/>

            <inertia

        ixx=“0”

        ixy=“0”

        ixz=“0”

        iyy=“0”

        iyz=“0”

        izz="0.00197” />

    </inertial>

  </link>


  <transmission name="tran1">

    <type>transmission_interface/SimpleTransmission</type>

    <joint name="reaction_wheel_joint">

      <hardwareInterface>EffortJointInterface</hardwareInterface>

    </joint>

    <actuator name="motor1">

      <hardwareInterface>EffortJointInterface</hardwareInterface>

      <mechanicalReduction>1</mechanicalReduction>

    </actuator>

  </transmission>
```

We attach the reaction wheel to the cepheus link with a continuous joint, in order to rotate Cepheus if we rotate reaction wheel. What’s new is the transmission element. To use ros_control with your robot, you need to add some additional elements to your URDF. The element is used to link actuators to joints, see the spec for exact XML format. We’ll use them in a minute. Also add a gazebo tag in the cepheus.gazebo file for this link :

```
<gazebo reference="reaction_wheel">

  <material>Gazebo/White</material>

</gazebo>
```

As usual, we specify the color used in material. Last but not least, we want to add the arm of Cepheus. The arm is consisted of the shoulder, the elbow and the gripper.

```
  <joint

    name="left_shoulder"

    type="revolute">

    <origin

      xyz="0.17271 0.091404 0.05875"

      rpy="0 0 0" />

    <parent

      link="cepheus" />

    <child

      link="left_arm" />

    <axis

      xyz="0 0 1" />

    <limit

      lower="-0.86"

      upper="2.54"

      effort="20"

      velocity="20" />

    <dynamics

      damping="1"

      friction="1" />

  </joint>
```

```
<link
  name="left_arm">
  <inertial>
    <origin
      xyz="0.10516 0.00268 0.00996"
      rpy="0 0 0" />
    <mass
      value="0.07974" />
    <inertia
      ixx="0.0971318625270812"
      ixy="0.00469370636895255"
      ixz="-0.000908920687249813"
      iyy="0.0755866683160163"
      iyz="0.000665178045940108"
      izz="0.0871046527712426" />
  </inertial>
  <visual>
    <origin
      xyz="0 -0.0085 0"
      rpy="0 0 0" />
    <geometry>
      <mesh
        filename="package://mybot_description/urdf/meshes/left_arm.STL" />
    </geometry>
    <material name="LightGrey"/>
  </visual>
  <collision>
    <origin
      xyz="0.08 0.0 0.01"
      rpy="0 0 0" />
    <geometry>
      <box size="0.13 0.02 0.02"/>
    </geometry>
  </collision>
</link>

<transmission name="tran2">
  <type>transmission_interface/SimpleTransmission</type>
  <joint name="left_shoulder">
    <hardwareInterface>EffortJointInterface</hardwareInterface>
  </joint>
  <actuator name="motor2">
    <hardwareInterface>EffortJointInterface</hardwareInterface>
    <mechanicalReduction>1</mechanicalReduction>
  </actuator>
</transmission>

<joint
  name="left_elbow"
  type="revolute">
  <origin
    xyz="0.18120 -0.01176 -0.002"
    rpy="0 0 0" />
  <parent
    link="left_arm" />
  <child
    link="left_forearm" />
  <axis
    xyz="0 0 1" />
  <limit
    lower="-3.14"
    upper="1.5"
    effort="2"
    velocity="20" />
  <dynamics
    damping="1"
    friction="1" />
</joint>

<link
  name="left_forearm">
  <inertial>
    <origin
      xyz="0.06831 0.00878 0.01558"
      rpy="0 0 0" />
    <mass
      value="0.06392" />
    <inertia
      ixx="1"
      ixy="1"
      ixz="1"
      iyy="1"
      iyz="1"
      izz="1" />
  </inertial>
  <visual>
    <origin
      xyz="0 0 0"
      rpy="0 0 0" />
    <geometry>
      <mesh
        filename="package://mybot_description/urdf/meshes/left_forearm.STL" />
    </geometry>
    <material name="LightGrey"/>
  </visual>
  <collision>
    <origin
      xyz="0.105 0.0115 0.0125"
      rpy="0 0 0" />
    <geometry>
      <box size="0.13 0.025 0.025"/>
    </geometry>
  </collision>
</link>

<transmission name="tran3">
  <type>transmission_interface/SimpleTransmission</type>
  <joint name="left_elbow">
    <hardwareInterface>EffortJointInterface</hardwareInterface>
  </joint>
  <actuator name="motor3">
    <hardwareInterface>EffortJointInterface</hardwareInterface>
    <mechanicalReduction>1</mechanicalReduction>
  </actuator>
</transmission>

<joint
  name="left_wrist"
  type="fixed">
  <origin
    xyz="0.15965 0.01176 -0.00838"
    rpy="0 0 0 " />
  <parent
    link="left_forearm" />
  <child
    link="left_grip" />
  <axis
    xyz="0 0 1" />
    <limit
      lower="-1.570796327"
      upper="1.570796327"
      effort="1"
      velocity="10" />
    <dynamics
      damping="1"
      friction="1" />
  </joint>

  <link
    name="left_grip">
    <inertial>
      <origin
        xyz="0.01947 -0.00226 -0.00228"
        rpy="0 0 0" />
      <mass
        value="0.03547" />
      <inertia
        ixx="0.0926325136613974"
        ixy="-0.00333421740190982"
        ixz="0.00529966557344093"
        iyy="0.0752633235560557"
        iyz="0.00191940596202769"
        izz="0.0919273463968869" />
    </inertial>
    <visual>
      <origin
        xyz="0 0 0"
        rpy="0 0 0" />
      <geometry>
        <mesh
```

```
          filename="package://mybot_description/urdf/meshes/left_grip.STL" />
      </geometry>
      <material name="LightGrey"/>
    </visual>
    <collision>
      <origin
        xyz="0.0375 0 0.022"
        rpy="0 0 0" />
      <geometry>
        <box size="0.085 0.07 0.05"/>
      </geometry>
    </collision>
  </link>
```

Also add gazebo tags in the cepheus.gazebo file for these links :

```
<gazebo reference="left_arm">
  <material>Gazebo/Blue</material>
</gazebo>

<gazebo reference="left_forearm">
  <material>Gazebo/Yellow</material>
</gazebo>

<gazebo reference="left_grip">
  <material>Gazebo/Grey</material>
</gazebo>
```

Now you can launch your simulation and Cepheus should appear!

Connect your robot to ROS

Alright, our space robot is so nice, but we can’t do anything with it yet, as it has no connection with ROS. In order to add this connection we need to add gazebo plugins to our model. There are different kinds of plugins:

• World: Dynamic changes to the world, e.g. Physics, like illumination or gravity, inserting models
• Model: Manipulation of models (robots), e.g. move the robots
• Sensor: Feedback from virtual sensor, like camera, laser scanner
• System: Plugins that are loaded by the GUI, like saving images

In this tutorial we’ll use a plugin to provide access to the joints of the space robot (reaction wheel joint and arm joints). The transmission tags in our URDF will be used by this plugin to define how to link the joints to controllers. To activate the plugin, add the following to cepheus.gazebo:

```
<gazebo>

  <plugin name="gazebo_ros_control" filename="libgazebo_ros_control.so">

    <robotNamespace>/cepheus</robotNamespace>

  </plugin>

</gazebo>
```

With this plugin, we will be able to control the joints, however we need to provide some extra configuration and explicitely start controllers for the joints. In order to do so, we’ll use the package cepheus_control that we have defined before. Let’s first create the configuration file:

```
roscd cepheus_control

mkdir config

cd config

gedit cepheus_control.yaml
```

This file will define four controllers: one velocity controller for reaction wheel, two position controllers for each arm joint and one for publishing the joint states. It also defines the PID gains for these controllers. Add the following in this file:

```
cepheus:

  # Publish all joint states -----------------------------------

  joint_state_controller:

    type: joint_state_controller/JointStateController

    publish_rate: 50


  # Position Controllers ---------------------------------------
  left_shoulder_position_controller:

    type: effort_controllers/JointPositionController

    joint: left_shoulder

    pid: {p: 1.0, i: 0.1, d: 1.0}


  # Position Controllers ---------------------------------------

  left_elbow_position_controller:

    type: effort_controllers/JointPositionController

    joint: left_elbow

    pid: {p: 1.0, i: 0.1, d: 1.0}


  # Velocity Controllers ---------------------------------------

  reaction_wheel_velocity_controller:

    type: effort_controllers/JointVelocityController

    joint: reaction_wheel_joint

    pid: {p: 1.0, i: 0.1, d: 0.001}
```

Now we need to create a launch file to start the controllers. For this, let’s do:

```
roscd cepheus_control

mkdir launch

cd launch

gedit cepheus_control.launch
```

In this file we’ll put two things. First we’ll load the configuration and the controllers, and we’ll also start a node that will provide 3D transforms (tf) of our robot. This is not mandatory but that makes the simulation more complete:

```
<launch>


  <rosparam file="$(find cepheus_control)/config/cepheus_control.yaml"
command="load"/>


  <node name="controller_spawner"
```

```
    pkg="controller_manager"

    type="spawner" respawn="false"

    output="screen" ns="/cepheus"

    args="joint_state_controller left_shoulder_position_controller
left_elbow_position_controller reaction_wheel_velocity_controller"/>



  <node name="robot_state_publisher" pkg="robot_state_publisher"
type="robot_state_publisher" respawn="false" output="screen">

    <param name="robot_description" command="$(find xacro)/xacro '$(find
mybot_description)/urdf/cepheus.xacro'" />

    <remap from="/joint_states" to="/cepheus/joint_states" />

  </node>



</launch>
```

In args above, we can select which controllers we want to start. We could launch our model on gazebo and then launch the controller, but to save some time (and terminals), we’ll start the controllers automatically by adding a line to the “cepheus_world.launch” in the cepheus_gazebo package :

```
<include file="$(find cepheus_control)/launch/cepheus_control.launch" />
```

Now launch your simulations. In a separate terminal, if you do a “rostopic list” you should see the topics corresponding to your controllers. You can send commands manually to your robot. For example you can send to reaction wheel joint the velocity command 50 rad/s:

```
rostopic pub   /cepheus/reaction_wheel_velocity_controller/command
std_msgs/Float64 "data: 50"
```

The reaction wheel should start rotating in one direction and Cepheus in the opposite. Or you can send to elbow joint the position command 0.5 rad:

```
 rostopic pub -1  /cepheus/left_elbow_position_controller/command
std_msgs/Float64 "data: 0.5"
```

The elbow of the arm should start rotating and go to 0.5 rad. Congratulations, you can now control your joints through ROS ! You can also monitor the joint states by doing :

```
  rostopic echo /cepheus/joint_states
```

Finally, you can send commands from Simulink blocks dedicated to ROS (ROS Publisher, ROS Subscriber). Read this tutorial to get you started.

I hope you enjoyed!