Automation and Robotics Engineering

ROBOTICS LAB

HOMEWORK 1

Building your robot manipulator

Instructor:
Mario Selvaggio

Student:
Francesco Grasso
P38000046

Accademic Year 2023/2024

1. Create the description of your robot and visualize it in Rviz

1.a) Download the arm_description package from the repo https://github.com/RoboticsLab2023/arm_description.git into your catkin_ws using git commands

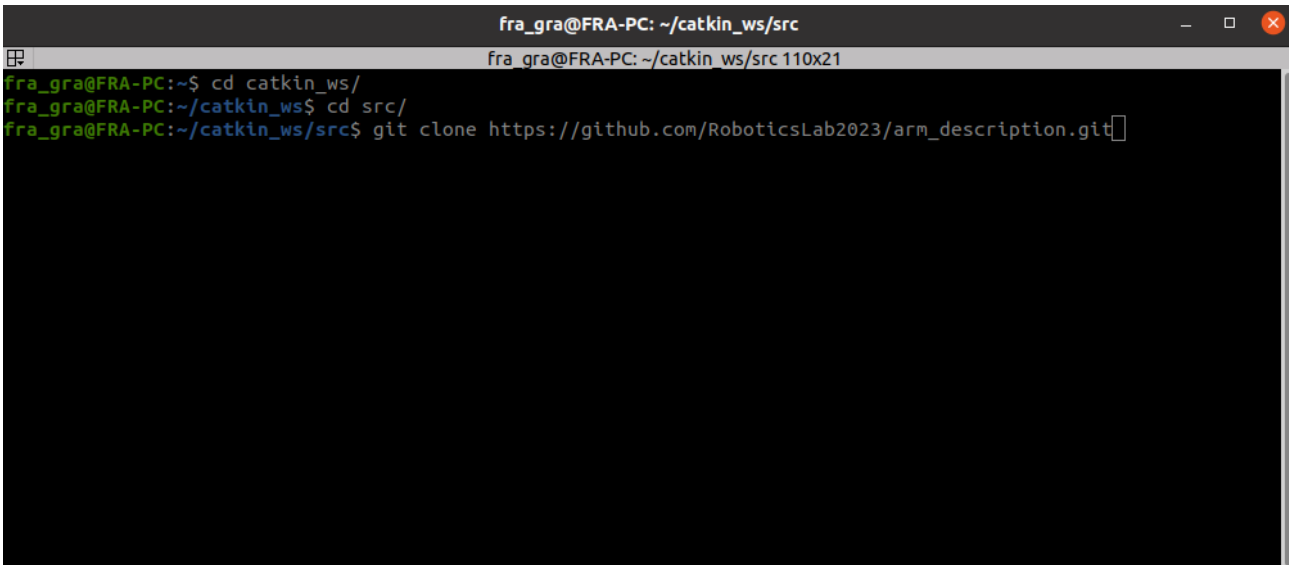


Figura 1

1.b) Within the package create a launch folder containing a launch file named display.launch that loads the URDF as a robot_description ROS param and starts the robot_state_publisher node, the joint_state_publisher node, and the rviz node. Launch the file using roslaunch. Note: To visualize your robot in rviz you have to change the Fixed Frame in the lateral bar and add the RobotModel plugin interface. Optional: save a .rviz configuration file, thad automatically loads the RobotModel plugin by default, and give it as an argument to your node in the display.launch file

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

  <arg name="rvizconfig" default="$(find arm_description)/rviz/arm.rviz" />


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

  <node name="robot_state_publisher" pkg="robot_state_publisher" type="robot_state_publisher"/>
  <node pkg="joint_state_publisher" type="joint_state_publisher" name="joint_state_publisher"/>

  <!-- Show in Rviz   -->
  <node name="rviz" pkg="rviz" type="rviz" args="-d $(arg rvizconfig)"/>
</launch>
```

Figura 2: display.launch file

This file load the urdf.xacro file as a robot_description ROS param and starts the robot_state_publisher node with the joint_state_publisher node and the rviz node.

This last node is processed with the previously saved configuration.

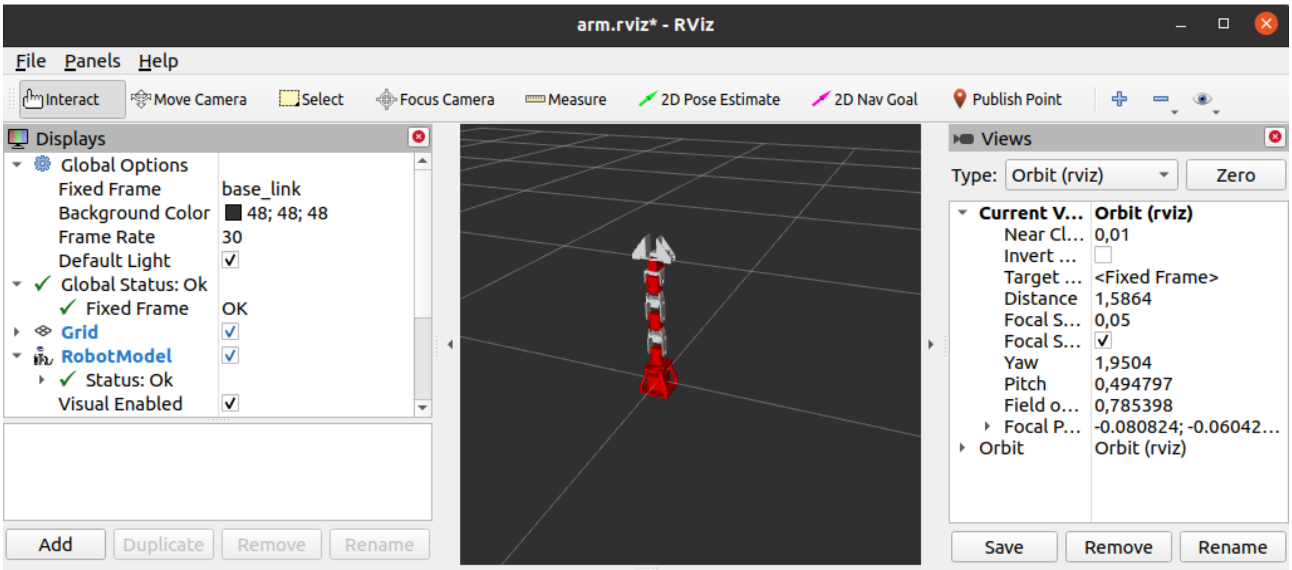


Figura 3: Rviz node

1.c) **Substitute the collision meshes of your URDF with primitive shapes. Use <box> geometries of reasonabe size approximating the links. Hint: Enable collision visualization in rviz (go to the lateral bar > Robot model > Collision Enabled) to adjust the collision meshes size**

In the URDF file the collision meshes have been modified by replacing them with <box> tag and the right shape sizes were chosen for each link.

```
  <link name="base_link">
    <visual>
      <geometry>
        <mesh filename="package://arm_description/meshes/base_link.stl" scale="0.001 0.001 0.001"/>
      </geometry>
      <origin rpy="0 0 0" xyz="0 0 0"/>
    </visual>
    <collision>
      <geometry>
        <box size="0.1 0.1 0.1"/>
      </geometry>
      <origin rpy="0 0 0" xyz="0 0 0"/>
    </collision>
    <inertial>
      <mass value="0.1"/>
      <inertia ixx="1.06682889e+08" ixy="0.0" ixz="0.0" iyy="9.92165844e+07" iyz="0.0" izz="1.26939175e+08"/>
    </inertial>
  </link>
```

Figura 4: base_link example with collision meshes modified

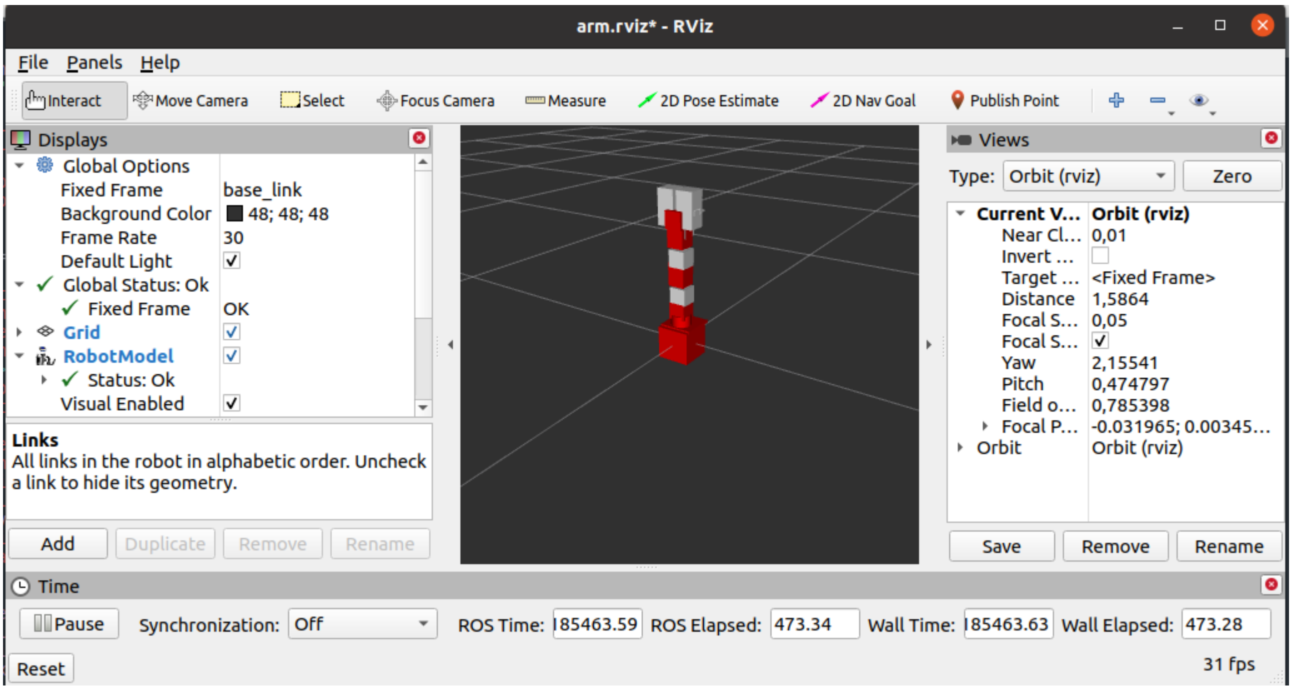


Figura 5: rviz with Collision Enabled

1.d) **Create a file named arm.gazebo.xacro within your package, define a xacro:macro inside your file containing all the <gazebo> tags you find within your arm.urdf and import it in your URDF using xacro:include.**
Remember to rename your URDF file to arm.urdf.xacro, add the string
xmlns:xacro="http://www.ros.org/wiki/xacro" within the <robot> tag, and load the URDF in your launch file using the xacro routine

```
<gazebo reference="f4">
<material>Gazebo/Red</material>
</gazebo>

<gazebo reference="f5">
<material>Gazebo/Red</material>
</gazebo>

<gazebo reference="wrist">
<material>Gazebo/Red</material>
</gazebo>

<gazebo reference="crawer_base">
<material>Gazebo/Red</material>
</gazebo>

<gazebo reference="base_link">
<material>Gazebo/Red</material>
</gazebo>

<gazebo reference="base_turn">
<material>Gazebo/Red</material>
</gazebo>

<gazebo reference="base_turn_rot">
<material>Gazebo/Red</material>
</gazebo>
```

Figura 6: examples of Gazebo tags inside arm.gazebo.xacro

The file **arm.gazebo.xacro** has been included in **arm.urdf.xacro** with the command: <**xacro:include filename=**
"$(find arm_description)/urdf/arm.gazebo.xacro"/>

The URDF has been loaded in the launch file as shown in **Fig. 2**.

2. **Add transmission and controllers to your robot and spawn it in Gazebo**

2.a) Create a package named arm_gazebo

The package named **arm_gazebo** has been created with the following steps:

1) **cd catkin_ws/src/**

2) **catkin_create_pkg arm_gazebo**

2.b) Within this package create a launch folder containing a arm_world.launch file

The file has been created in the following steps:

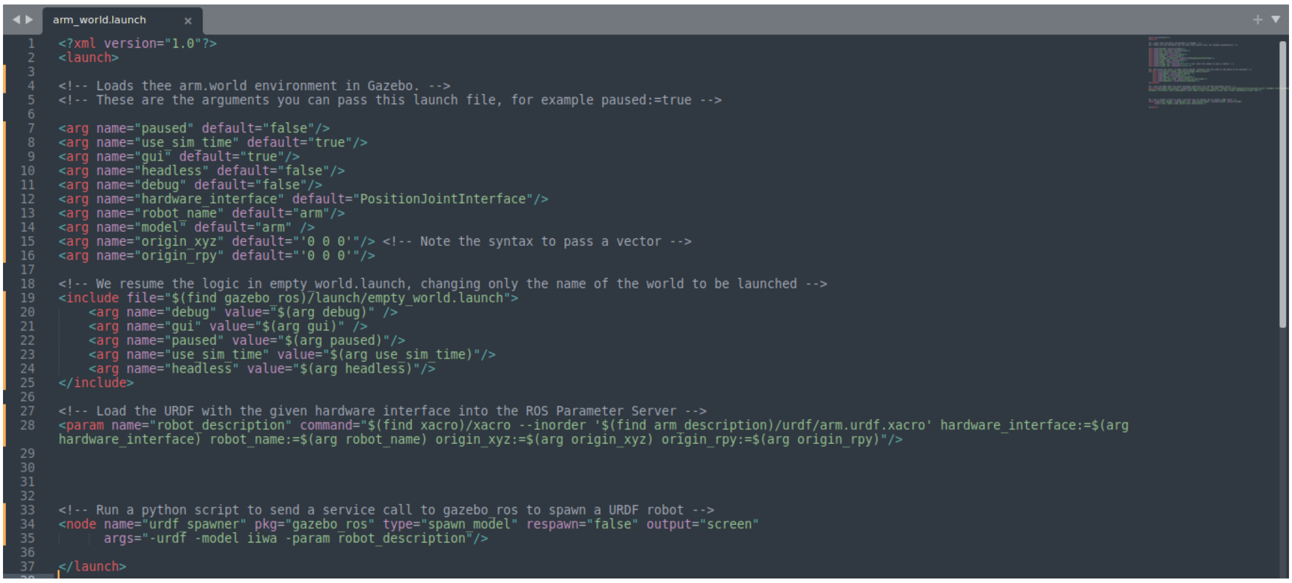
1) **cd catkin_ws/src/arm_gazebo**

2) **mkdir launch**

3) **cd launch**

4) **touch arm_world.launch**

2.c) Fill this launch file with commands that load the URDF into the ROS Parameter Server and spawn your robot using the spawn_model node.
Hint: follow the iiwa_world.launch example from the package iiwa_stack: https://github.com/IFL-CAMP/iiwa_stack/tree/master.
Launch the arm_world.launch file to visualize the robot in Gazebo



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

<!-- Loads thee arm.world environment in Gazebo. -->
<!-- These are the arguments you can pass this launch file, for example paused:=true -->

<arg name="paused" default="false"/>
<arg name="use_sim_time" default="true"/>
<arg name="gui" default="true"/>
<arg name="headless" default="false"/>
<arg name="debug" default="false"/>
<arg name="hardware_interface" default="PositionJointInterface"/>
<arg name="robot_name" default="arm"/>
<arg name="model" default="arm" />
<arg name="origin_xyz" default="'0 0 0'"/> <!-- Note the syntax to pass a vector -->
<arg name="origin_rpy" default="'0 0 0'"/>

<!-- We resume the logic in empty_world.launch, changing only the name of the world to be launched -->
<include file="$(find gazebo_ros)/launch/empty_world.launch">
    <arg name="debug" value="$(arg debug)" />
    <arg name="gui" value="$(arg gui)" />
    <arg name="paused" value="$(arg paused)"/>
    <arg name="use_sim_time" value="$(arg use_sim_time)"/>
    <arg name="headless" value="$(arg headless)"/>
</include>

<!-- Load the URDF with the given hardware interface into the ROS Parameter Server -->
<param name="robot_description" command="$(find xacro)/xacro --inorder '$(find arm_description)/urdf/arm.urdf.xacro' hardware_interface:=$(arg hardware_interface) robot_name:=$(arg robot_name) origin_xyz:=$(arg origin_xyz) origin_rpy:=$(arg origin_rpy)"/>




<!-- Run a python script to send a service call to gazebo_ros to spawn a URDF robot -->
<node name="urdf_spawner" pkg="gazebo_ros" type="spawn_model" respawn="false" output="screen"
      args="-urdf -model iiwa -param robot_description"/>

</launch>
```

Figura 7: arm_world.launch

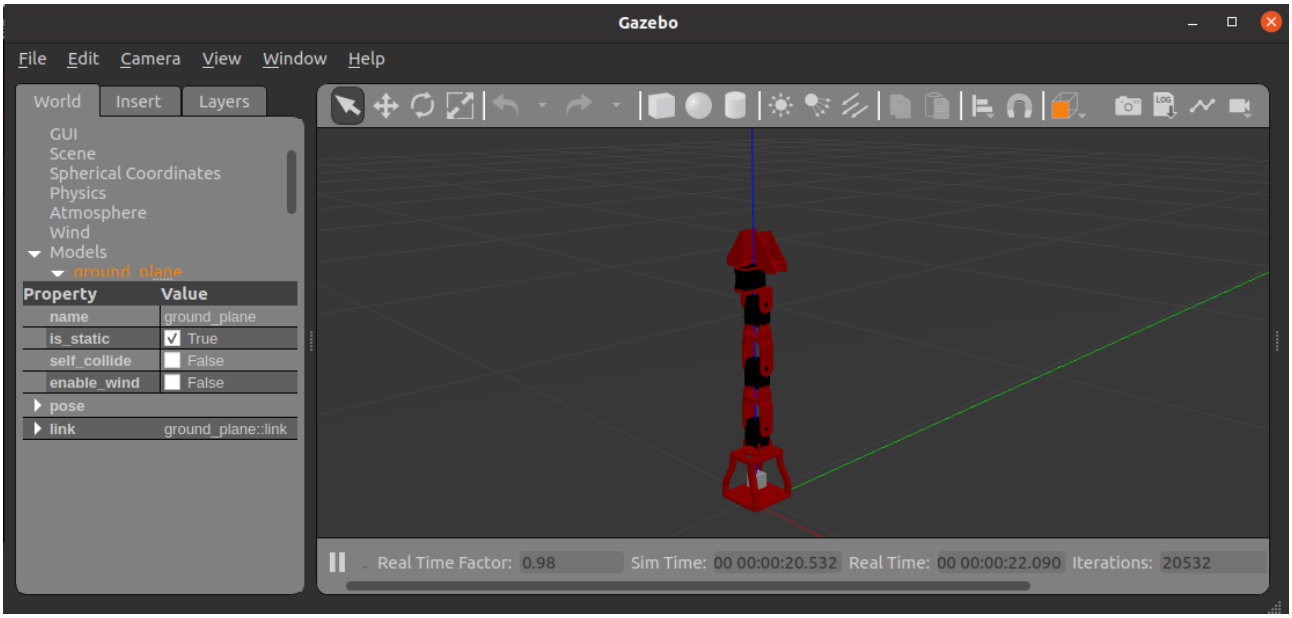


Figura 8: Simulation in Gazebo

2.d) **Now add a PositionJointInterface as hardware interface to your robot: create a arm.transmission.xacro file into your arm_description/urdf folder containing a xacro:macro with the hardware interface and load it into your arm.urdf.xacro file using xacro:include. Launch the file**

arm.transmission.xacro

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

  <xacro:macro name="arm_transmission" params="hardware_interface robot_name ">


    <transmission name="${robot_name}_tran_0">
      <robotNamespace>/${robot_name}</robotNamespace>
      <type>transmission_interface/SimpleTransmission</type>
      <joint name="j0">
        <hardwareInterface>hardware_interface/${hardware_interface}</hardwareInterface>
      </joint>
      <actuator name="${robot_name}_motor_0">
        <hardwareInterface>hardware_interface/${hardware_interface}</hardwareInterface>
        <mechanicalReduction>1</mechanicalReduction>
      </actuator>
    </transmission>

```

Figura 9: First transmission in arm.transmission.xacro

The file arm.transmission.xacro has been included in arm.urdf.xacro with the command: **<xacro:include filename=**

"$(find arm_description)/urdf/arm.transmission.xacro"/>

2.e) **Add joint position controllers to your robot: create a arm_control package with a arm_control.launch file inside its launch folder and a arm_control.yaml file within its config folder**

The following commands were typed:

1) **cd catkin_ws/src/**

2) **catkin_create_pkg arm_control**

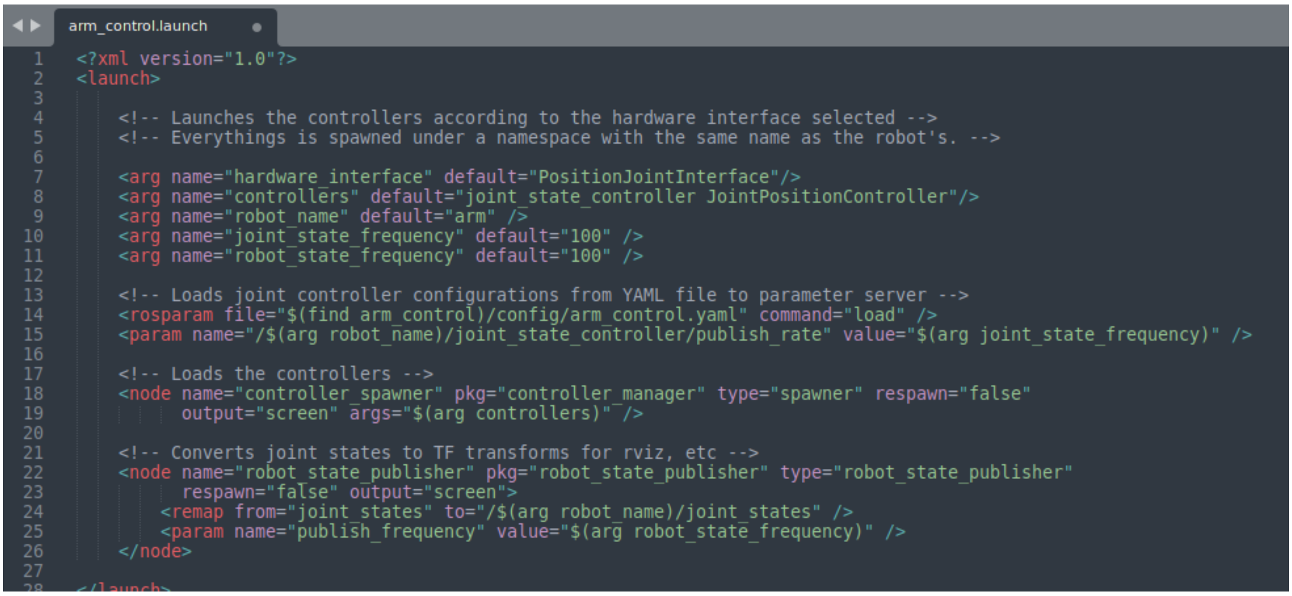
3) **mdkir launch**

4) mkdir config

5) touch launch/arm_control.launch

6) touch config/arm_control.yaml

2.f) **Fill the arm_control.launch file with commands that load the joint controller configurations from the .yaml file to the parameter server and spawn the controllers using the controller_manager package. Hint: follow the iiwa_control.launch example from corresponding package**




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

    <!-- Launches the controllers according to the hardware interface selected -->
    <!-- Everythings is spawned under a namespace with the same name as the robot's. -->

    <arg name="hardware_interface" default="PositionJointInterface"/>
    <arg name="controllers" default="joint_state_controller JointPositionController"/>
    <arg name="robot_name" default="arm" />
    <arg name="joint_state_frequency" default="100" />
    <arg name="robot_state_frequency" default="100" />

    <!-- Loads joint controller configurations from YAML file to parameter server -->
    <rosparam file="$(find arm_control)/config/arm_control.yaml" command="load" />
    <param name="/$(arg robot_name)/joint_state_controller/publish_rate" value="$(arg joint_state_frequency)" />

    <!-- Loads the controllers -->
    <node name="controller_spawner" pkg="controller_manager" type="spawner" respawn="false"
          output="screen" args="$(arg controllers)" />

    <!-- Converts 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">
        <remap from="joint_states" to="/$(arg robot_name)/joint_states" />
        <param name="publish_frequency" value="$(arg robot_state_frequency)" />
    </node>

</launch>
```

Figura 10: arm_control.launch

2.g) Fill the arm arm_control.yaml adding a joint_state_controller and a JointPositionController to all the joints

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


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


  PositionJointInterface_J0_controller:
    type: position_controllers/JointPositionController
    joint: j0

  PositionJointInterface_J1_controller:
    type: position_controllers/JointPositionController
    joint: j1

  PositionJointInterface_J2_controller:
    type: position_controllers/JointPositionController
    joint: j2

  PositionJointInterface_J3_controller:
    type: position_controllers/JointPositionController
    joint: j3

```

Figura 11: arm_control.yaml

2.h) Create an arm_gazebo.launch file into the launch folder of the arm_gazebo package loading the Gazebo world with arm_world.launch and spawning the controllers within arm_control.launch. Go to the arm_description package and add the gazebo_ros_control plugin to your main URDF into the arm.gazebo.xacro file. Launch the simulation and check if your controllers are correctly loaded arm_gazebo.launch

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

    <arg name="hardware_interface" default="PositionJointInterface" />
    <arg name="robot_name" default="arm" />

    <!-- Loads the Gazebo world. -->
    <include file="$(find arm_gazebo)/launch/arm_world.launch">
        <arg name="hardware_interface" value="$(arg hardware_interface)" />
        <arg name="robot_name" value="$(arg robot_name)" />
    </include>

    <group ns="$(arg robot_name)">
    <!-- Spawn controllers - it uses a position Controller for each joint -->

        <include file="$(find arm_control)/launch/arm_control.launch">
            <arg name="hardware_interface" value="$(arg hardware_interface)" />
            <arg name="controllers" value="joint_state_controller
                 $(arg hardware_interface)_J0_controller
                 $(arg hardware_interface)_J1_controller
                 $(arg hardware_interface)_J2_controller
                 $(arg hardware_interface)_J3_controller"/>
            <arg name="robot_name" value="$(arg robot_name)" />
        </include>
        </group>
</launch>

```

Figura 12: arm_gazebo.launch

```xml
<gazebo reference="crawer_right">
<material>Gazebo/Red</material>
</gazebo>

<!-- add the gazebo ROS plugin -->
<!-- Load Gazebo lib and set the robot namespace -->
  <gazebo>
      <plugin name="gazebo_ros_control" filename="libgazebo_ros_control.so">
        <robotNamespace>/${"arm"}</robotNamespace>
      </plugin>
    </gazebo>
```

Figura 13: gazebo_ros_control plugin in arm.gazebo.xacro

3. **Add a camera sensor to your robot**

3.a) Go into your arm.urdf.xacro file and add a camera_link and a fixed camera_joint with base_link as a parent link. Size and position the camera link opportunely

```xml
<joint name="camera_joint" type="fixed">
   <parent link="base_link"/>
   <child link="camera_link"/>
   <origin xyz="0 0 0" rpy="0 0 0"/>
 </joint>

   <link name="camera_link">
     <collision>
       <origin xyz="0 0 0" rpy="0 0 0"/>
       <geometry>
         <box size="${camera_link} ${camera_link} ${camera_link}"/>
       </geometry>
     </collision>

     <visual>
       <origin xyz="0 0 0" rpy="0 0 0"/>
       <geometry>
         <box size="${camera_link} ${camera_link} ${camera_link}"/>
       </geometry>
       <material name="Red"/>
     </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>
```

Figura 14: camera_link and camera_joint in arm.urdf.xacro

3.b) In the arm.gazebo.xacro add the gazebo sensor reference tags and the libgazebo_ros_camera plugin to your xacro (slide 74-75)

```xml
<!-- camera reference -->

  <!-- camera -->
  <gazebo reference="camera_link">
    <sensor name="camera1" type="camera">
      <update_rate>30.0</update_rate>
      <camera name="head">
        <horizontal_fov>1.3962634</horizontal_fov>
        <image>
          <width>800</width>
          <height>800</height>
          <format>R8G8B8</format>
        </image>
        <clip>
          <near>0.02</near>
          <far>300</far>
        </clip>
        <noise>
          <type>gaussian</type>
          <!-- Noise is sampled independently per pixel on each frame.
                That pixel's noise value is added to each of its color
                channels, which at that point lie in the range [0,1]. -->
          <mean>0.0</mean>
          <stddev>0.007</stddev>
        </noise>
      </camera>
```

Figura 15: gazebo sensor reference tags

```xml
  <!-- camera plugin -->

      <plugin filename="libgazebo_ros_camera.so" name="camera_controller">
        <alwaysOn>true</alwaysOn>
        <updateRate>0.0</updateRate>
        <cameraName>arm/camera1</cameraName>
        <imageTopicName>image_raw</imageTopicName>
        <cameraInfoTopicName>camera_info</cameraInfoTopicName>
        <frameName>camera_link_optical</frameName>
        <!-- setting hackBaseline to anything but 0.0 will cause a misalignment
              between the gazebo sensor image and the frame it is supposed to
              be attached to -->
        <hackBaseline>0.0</hackBaseline>
        <distortionK1>0.0</distortionK1>
        <distortionK2>0.0</distortionK2>
        <distortionK3>0.0</distortionK3>
        <distortionT1>0.0</distortionT1>
        <distortionT2>0.0</distortionT2>
        <CxPrime>0</CxPrime>
        <Cx>0.0</Cx>
        <Cy>0.0</Cy>
        <focalLength>0.0</focalLength>
      </plugin>
    </sensor>
  </gazebo>

  </xacro:macro>
</robot>
```

Figura 16: libgazebo_ros_camera.plugin

3.c) Launch the Gazebo simulation with using arm_gazebo.launch and check if the image topic is correctly published using rqt_image_view

A robot was placed in front of the camera to test the code:

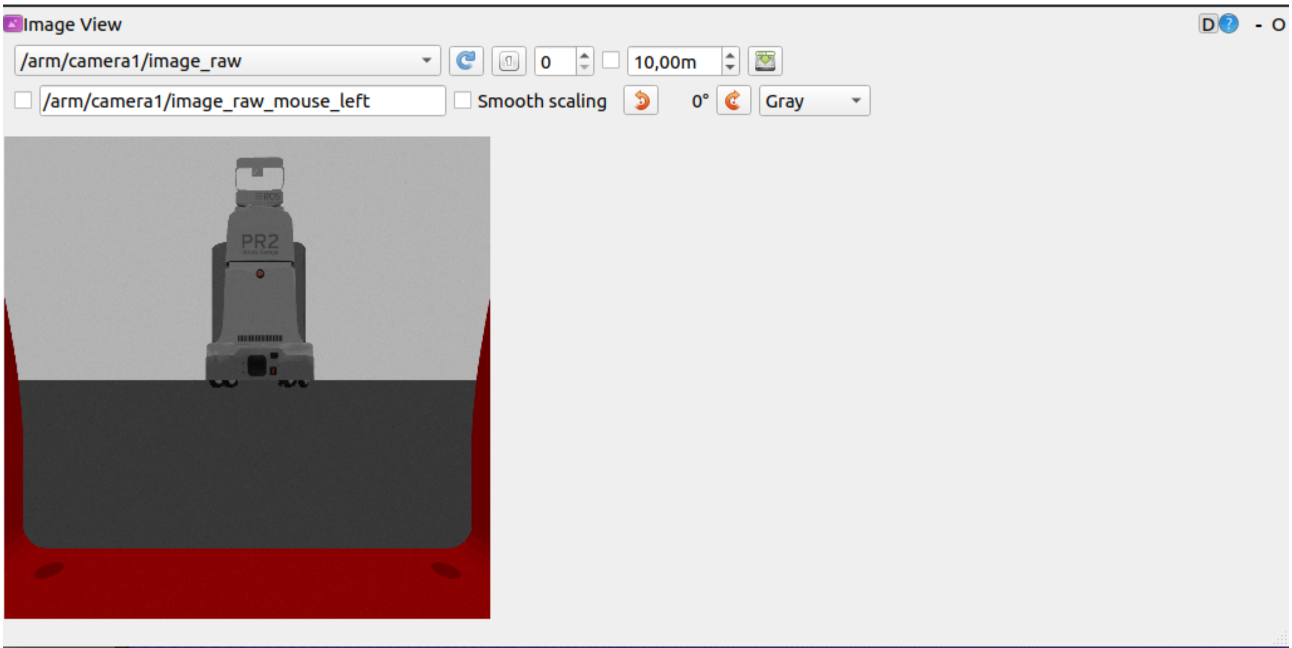


Figura 17: image_raw correctly published

3.d) Optionally: You can create a camera.xacro file (or download one from
https://github.com/ CentroEPiaggio/irobotcreate2ros/blob/master/mode
and add it to your robot URDF using <xacro:include>

1) The **camera.urdf.xacro** was downloaded from the link above and placed in the **arm_description/urdf** folder.

2) It was included in **arm.urdf.xacro** in the same way as **arm.transmission.xacro** and **arm.gazebo.xacro**.

3) The code about **camera_link**, **camera_joint** , **gazebo sensor reference** and **libgazebo_ros_camera plugin** has been commented.

4) An stl file of the **Canon 5D MkII camera** was downloaded from the web and placed in **arm_description/meshes**.

5) The **camera.urdf.xacro** has been modified as in the figure below:

```
            <child link="camera_link"/>
        </joint>

        <link name="camera_link">
            <collision>
                <origin xyz="0 0 0" rpy="0 0 0"/>
                <geometry>
                    <box size="0.02 0.08 0.05"/>
                </geometry>
            </collision>
            <visual>
                <origin xyz="0 0 0" rpy="0 0 1.570795"/>
                <geometry>
                    <mesh filename="package://arm_description/meshes/Canon_5D_Mk2.stl" scale=" 0.00025
                    0.00025 0.00025"/>
                </geometry>
            </visual>
            <inertial>
                <mass value="0.0001" />
                <origin xyz="0 0 0" rpy="0 0 3.14"/>
                <inertia ixx="0.0000001" ixy="0" ixz="0" iyy="0.0000001" iyz="0" izz="0.0000001" />
            </inertial>
        </link>
```

Figura 18: camera.urdf.xacro

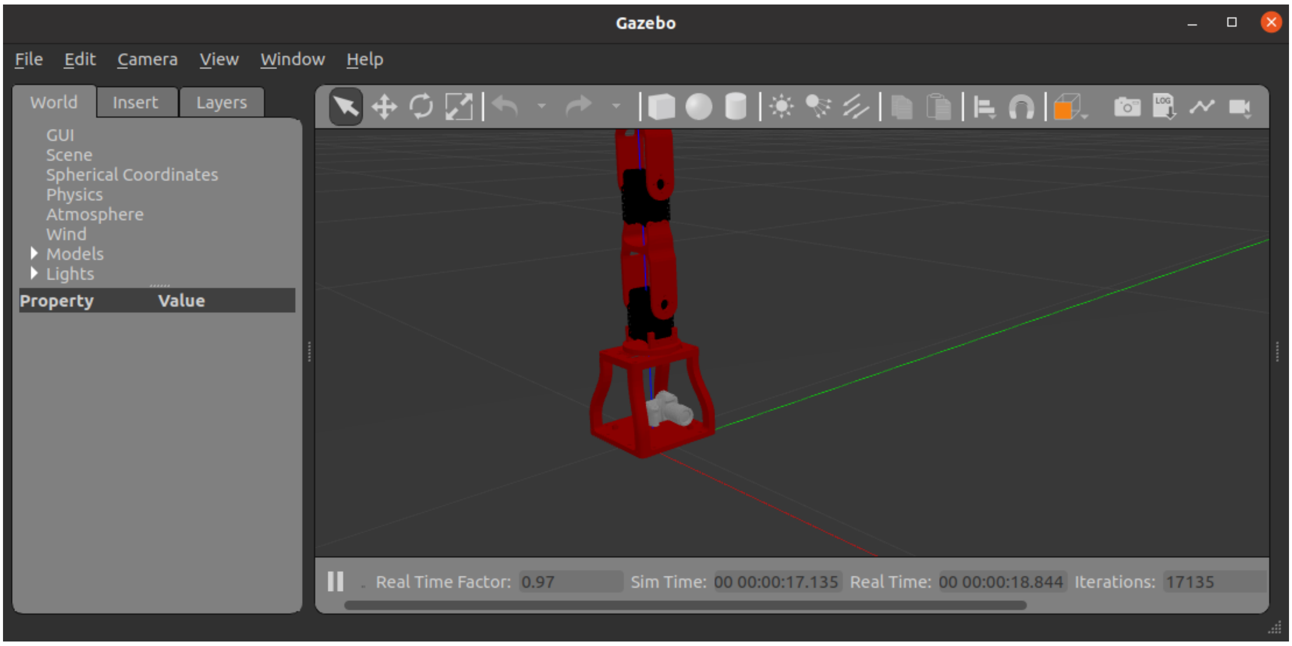


Figura 19: arm_gazebo.launch with STL camera

4. **Create a ROS publisher node that reads the joint state and sends joint position commands to your robot**

4.a) Create an arm_controller package with a ROS C++ node named arm_controller_node. The dependencies are roscpp, sensor_msgs and std_msgs. Modify opportunely the CMakeLists.txt file to compile your node. Hint: uncomment add_executable and target_link_libraries lines

The following commands were typed:

1) **cd catkin_ws/src/**

2) **catkin_create_pkg arm_controller roscpp sensor_msg std_msgs**

3) **cd arm_controller/src**

4) **touch arm_controller_node.cpp**

The following lines of code have been uncommented in the CMakeLists.txt file:

1) add_executable($PROJECT_NAME_node src/arm_controller_node.cpp)

2)target_link_libraries($PROJECT_NAME_node $catkin_LIBRARIES)

4.b) Create a subscriber to the topic joint_states and a callback function that prints the current joint positions (see Slide 45). Note: the topic contains a sensor_msgs/JointState

```cpp
#include "ros/ros.h"
#include "std_msgs/Float64.h"
#include <sensor_msgs/JointState.h>
#include "std_msgs/String.h"
#include <sstream>

void Call_Fun(const sensor_msgs::JointState::ConstPtr& msg)
{
  for (size_t i = 0; i < msg->position.size(); i++)
    {
        ROS_INFO("Joint Name: %s, Position: %f", msg->name[i].c_str(), msg->position[i]);
    }
}



int main(int argc, char **argv)
{


    ros::init(argc, argv, "arm_controller_node");
    ros::NodeHandle n;

// ********************** SUBSCRIBER **************************
   ros::Subscriber joint_states_sub = n.subscribe("/arm/joint_states", 10, Call_Fun);

```

Figura 20: Subscriber to the topic joint_states with Call_Fun as callback function.

4.c) Create publishers that write commands onto the controllers' /command topics (see Slide 46). Note: the command is a std_msgs/Float64

Four publishers have been added (one for each joint).

```cpp
// ********************** PUBLISHER **************************

    ros::Publisher j0_controller_pub = n.advertise<std_msgs::Float64>("/arm/PositionJointInterface_J0_controller/command", 10);
    ros::Publisher j1_controller_pub = n.advertise<std_msgs::Float64>("/arm/PositionJointInterface_J1_controller/command", 10);
    ros::Publisher j2_controller_pub = n.advertise<std_msgs::Float64>("/arm/PositionJointInterface_J2_controller/command", 10);
    ros::Publisher j3_controller_pub = n.advertise<std_msgs::Float64>("/arm/PositionJointInterface_J3_controller/command", 10);

    ros::Rate loop_rate(10);


    std_msgs::Float64 J2_command;

    while (ros::ok())
    {
        std_msgs::Float64 J0_command;
        std_msgs::Float64 J1_command;
        std_msgs::Float64 J2_command;
        std_msgs::Float64 J3_command;

         J0_command.data = 0.5;
         j0_controller_pub.publish(J0_command);

         J1_command.data = 1;
         j1_controller_pub.publish(J1_command);

         J2_command.data = -0.5;
         j2_controller_pub.publish(J2_command);

         J3_command.data = 0.8;
         j3_controller_pub.publish(J3_command);


        ros::spinOnce();
        loop_rate.sleep();
    }

    return 0;
}
```

Figura 21: Publishers that write commands onto /command topics

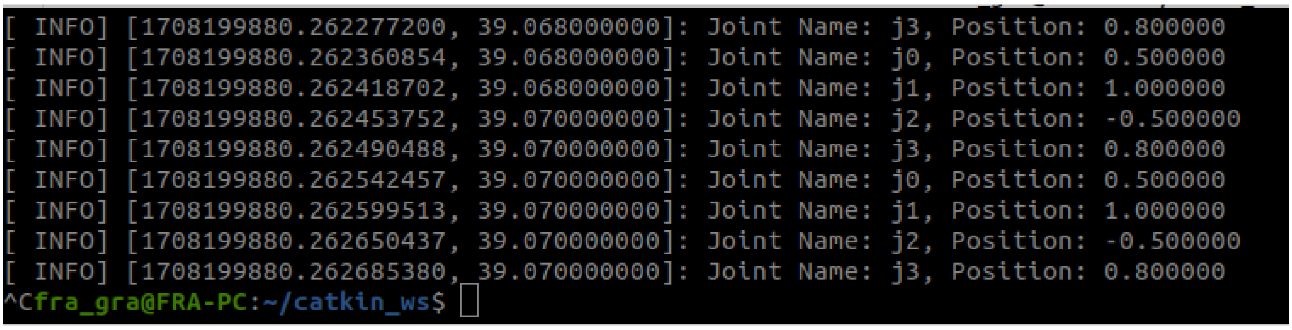


Figura 22: Subscriber output

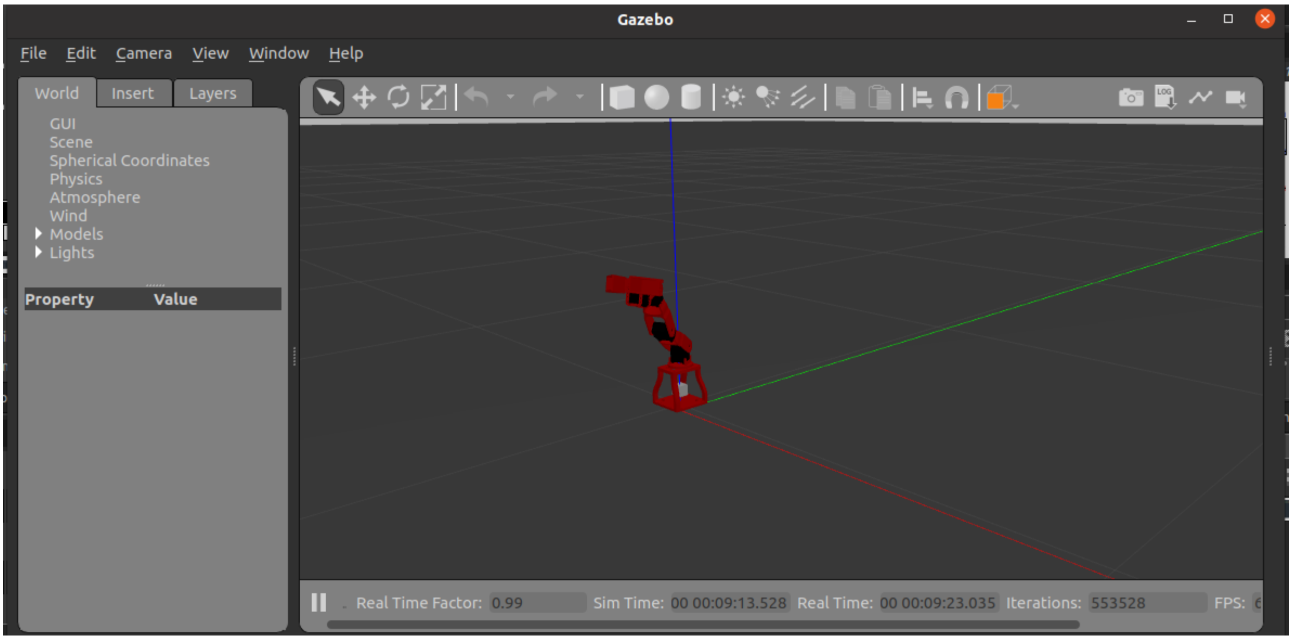


Figura 23: Final Pose of the Robot