# Report – Homework 1

**Students:**

**Luca Marseglia**

**https://github.com/marseluca/arm_stack.git**

**Benito Vodola**

**https://github.com/BenitoVodola/arm_stack.git**

**Domenico Tuccillo**

**https://github.com/DomenicoTuccillo/arm_stack.git**

**Debora Ippolito**

**https://github.com/DeboraIppolito/arm_stack.git**

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

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

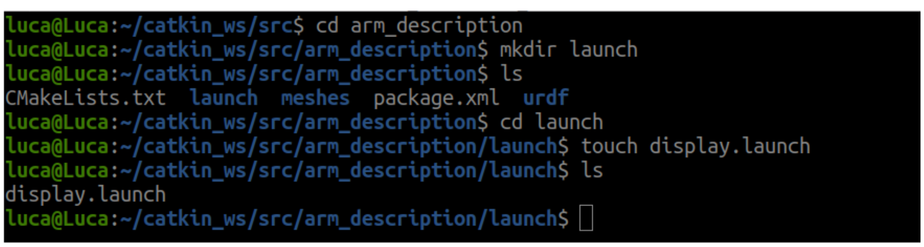
Launched the command:

```
$ git clone https://github.com/RoboticsLab2023/arm_description.git
```

To download the package into the “catkin_ws” workspace.

## (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.

First, a “display.launch” file was created inside a “launch” folder in the “arm_description” package:



```
luca@Luca:~/catkin_ws/src$ cd arm_description
luca@Luca:~/catkin_ws/src/arm_description$ mkdir launch
luca@Luca:~/catkin_ws/src/arm_description$ ls
CMakeLists.txt  launch  meshes  package.xml  urdf
luca@Luca:~/catkin_ws/src/arm_description$ cd launch
luca@Luca:~/catkin_ws/src/arm_description/launch$ touch display.launch
luca@Luca:~/catkin_ws/src/arm_description/launch$ ls
display.launch
luca@Luca:~/catkin_ws/src/arm_description/launch$
```

This file:

- Loads the URDF as a <param> named “robot_description”
- Starts the nodes: “robot_state_publisher”, “joint_state_publisher” and “rviz”

display.launch U

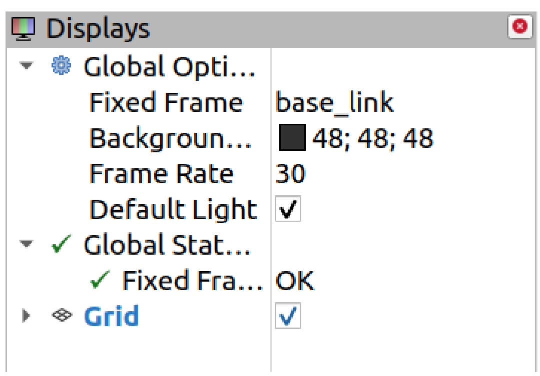
arm_description > launch > display.launch

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

  <!-- This lauch file just loads the URDF with the given hardware interface and robot name into the ROS Parameter Server -->
  <arg name="hardware_interface" default="PositionJointInterface"/>
  <arg name="robot_name" 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'"/>

  <node name="joint_state_publisher" pkg="joint_state_publisher" type="joint_state_publisher" />
  <node name="robot_state_publisher" pkg="robot_state_publisher" type="robot_state_publisher" />
  <node type="rviz" name="rviz" pkg="rviz" args="-d $(find rviz)/rviz/config_file.rviz" />

  <!-- <param name="robot_description" textfile="$(find arm_description)/urdf/arm.urdf" /> -->
  <param name="robot_description" command="$(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)"/>

</launch>
```

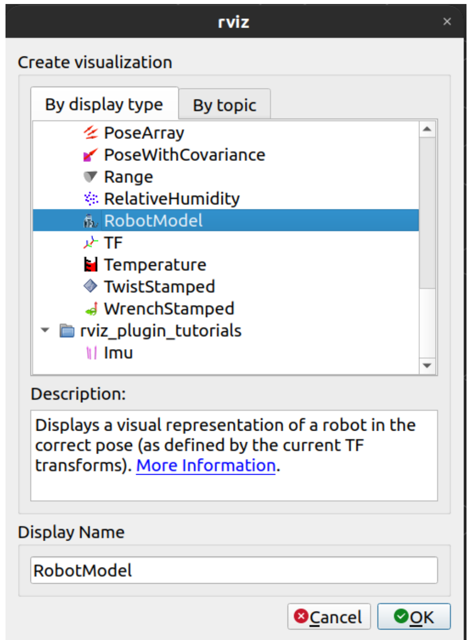
The file was launched using the “roslaunch” command:

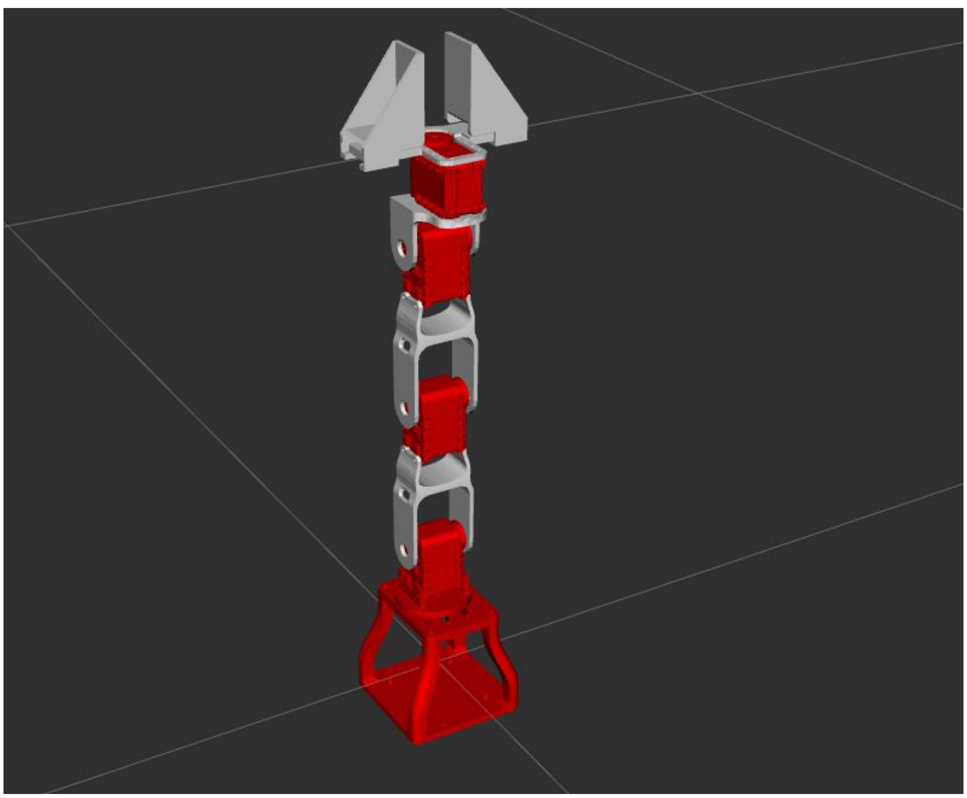
```
$ roslaunch arm_description display.launch
```

Then, the “Fixed frame” was changed to “base_link”:




And the “RobotModel” plugin was added to Rviz to visualize the robot:



Eventually, the configuration was saved in the "arm_config.rviz" file and then a line was added to the "display.launch" file, that automatically loads the configuration:

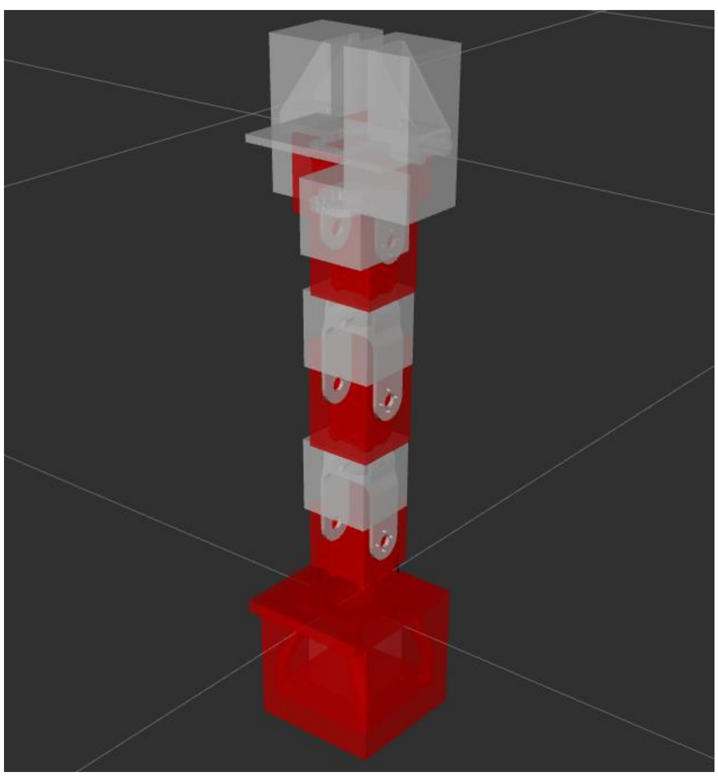
```
<node name="rviz" pkg="rviz" type="rviz" args="-d $(find arm_description)/arm_config.rviz"/>
```

**(c) Substitute the collision meshes of your URDF with primitive shapes. Use geometries of reasonable size approximating the links.**

The collision meshes for each link in the URDF file were replaced with <box> geometries approximating the links, after enabling collision visualization in Rviz:

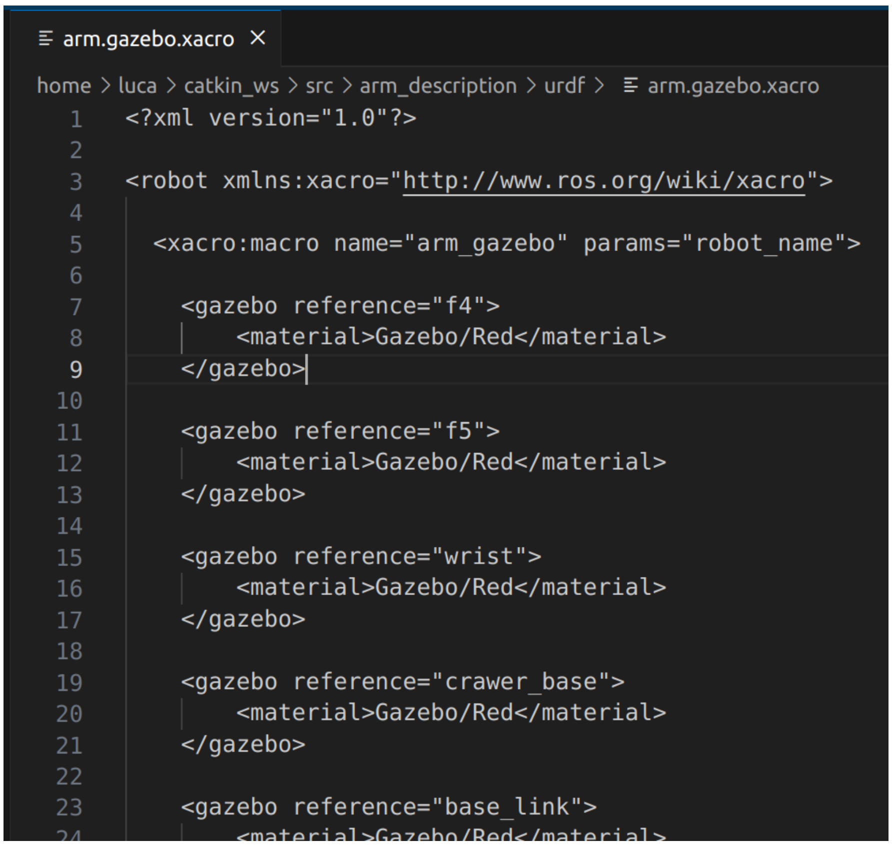
```
<robot name="arm">

  <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>
        <!-- <mesh filename="package://arm_description/meshes/base_link.stl" scale="0.001 0.001 0.001"/> -->
        <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>
```

The result is as follows:



**d) Create a file named arm.gazebo.xacro within your package, define a xacro:macro inside your file containing all the tags you find within your arm.urdf and import it in your URDF using xacro:include.**

The “arm.gazebo.xacro” file was created inside the “arm_description” package. Then a “xacro:macro” was defined inside this file containing all the gazebo tags in the URDF file of the robot:


arm.gazebo.xacro

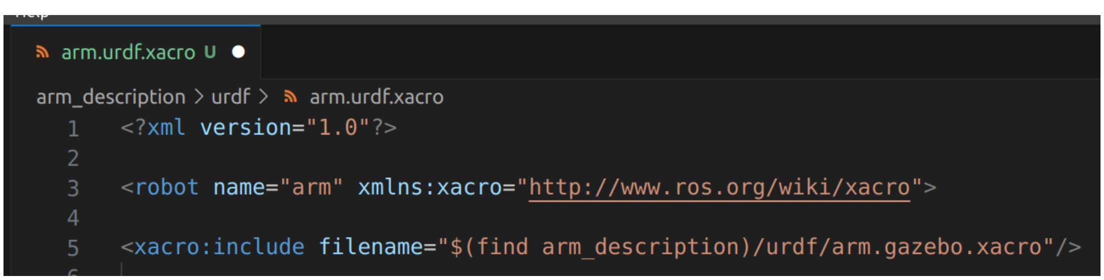
home > luca > catkin_ws > src > arm_description > urdf > arm.gazebo.xacro

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

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

  <xacro:macro name="arm_gazebo" params="robot_name">

    <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>
```

In the URDF file, the following lines were inserted inside the <robot> tag:

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

<xacro:arm_gazebo/>
```

To import “arm.gazebo.xacro” inside the URDF:


arm.urdf.xacro U

arm_description > urdf > arm.urdf.xacro

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

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

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

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

### (a) Create a package named arm_gazebo.

To create a package inside the src folder of our workspace, digit in a terminal the following code:

```
luca@Luca:~/catkin_ws/src$ catkin_create_pkg arm_gazebo
Created file arm_gazebo/package.xml
Created file arm_gazebo/CMakeLists.txt
Successfully created files in /home/luca/catkin_ws/src/arm_gazebo.
 the values in package.xml.
luca@Luca:~/catkin_ws/src$ ls
arm_description  arm_gazebo  iiwa_stack  my_package  ros_tutorials
```

### (b) Within this package create a launch folder containing an arm_world.launch file.

Inside the arm_gazebo package create a file a launch folder and inside it, a launch file named arm_world.lauch:

```
luca@Luca:~/catkin_ws/src$ cd arm_gazebo
luca@Luca:~/catkin_ws/src/arm_gazebo$ mkdir launch
luca@Luca:~/catkin_ws/src/arm_gazebo$ cd launch
luca@Luca:~/catkin_ws/src/arm_gazebo/launch$ touch arm_world.launch
luca@Luca:~/catkin_ws/src/arm_gazebo/launch$ ls
arm_world.launch
```

**(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.**

arm_world.launch

arm_gazebo > launch > arm_world.launch

```xml
<?xml version="1.0"?>
<launch>
  <!-- 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" />

    <!-- 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 -->
    <include file="$(find arm_description)/launch/arm_upload.launch">
        <arg name="hardware_interface" value="$(arg hardware_interface)"/>
        <arg name="robot_name" value="$(arg robot_name)" />
    </include>


  <!-- 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 arm -param robot_description"/>

    </launch>

```

Inside this launch file are defined different arguments with default values such as the robot's name or the hardware interface.

First, upload an empty world with the tag <include>. Then, with the same tag, include a launch file named “arm_upload.launch”, in which the URDF file with the information of our robot will be loaded. Moreover, assign the values of the arguments “hardware_interface” and “robot_name”.

In the end, run the spawn model node specifying the node name, the package, the type and the arguments.

Then, create the arm_upload.launch file:

```
luca@Luca:~/catkin_ws/src$ cd arm_description
luca@Luca:~/catkin_ws/src/arm_description$ cd launch
luca@Luca:~/catkin_ws/src/arm_description/launch$ touch arm_upload.launch
luca@Luca:~/catkin_ws/src/arm_description/launch$ ls
arm_upload.launch  display.launch
luca@Luca:~/catkin_ws/src/arm_description/launch$ 
```

This file loads robot’s information in the arm.urdf.xacro file by using the command:

$ (find xacro)/xacro

arm_description > launch > arm_upload.launch

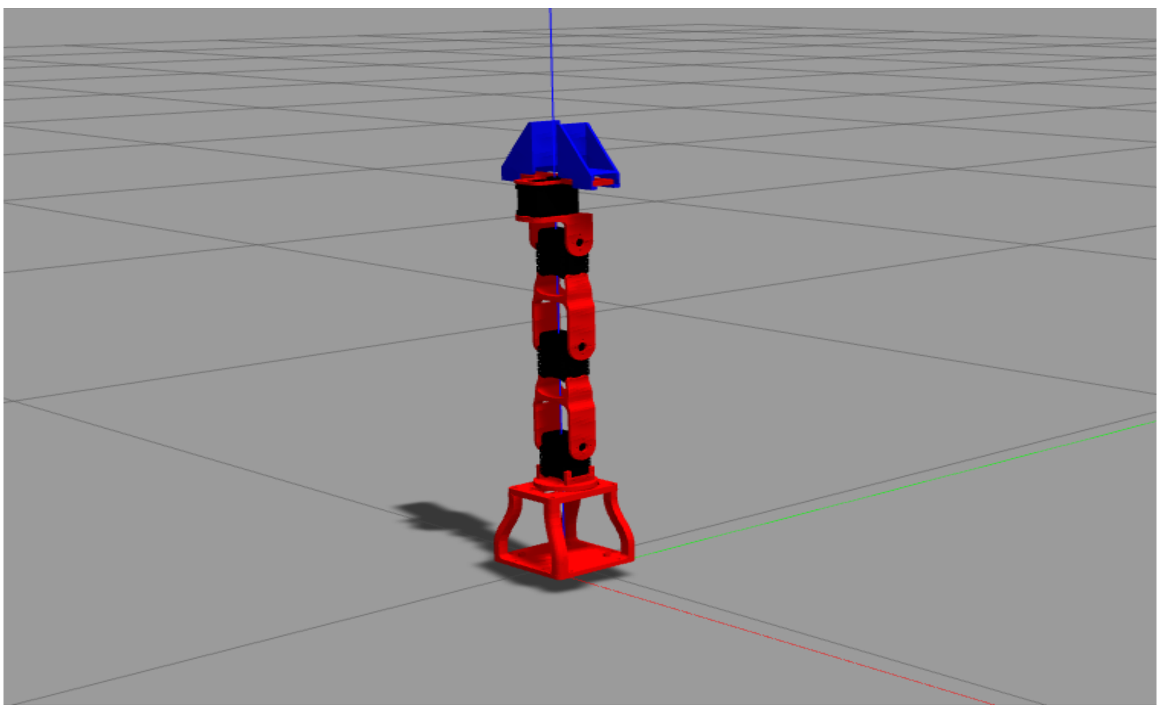
```
ml version="1.0"?>
unch>

!-- This lauch file just loads the URDF with the given hardware interface and robot name into the ROS Parameter Server -->
arg name="hardware_interface" default="PositionJointInterface"/>
arg name="robot_name" 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'"/>

!--param name="robot_description" textfile="$(find arm_description)/urdf/arm.urdf"/-->


aram name="robot_description" command="$(find xacro)/xacro --inorder '$(find arm_description)/urdf/arm.urdf.xacro' hardware_interface:=$(arg hardware_interface)"/>


aunch>

```

Launch the arm_world.launch file:

```
$ catkin build
```

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

```
$ rosrun arm_gazebo arm_world.launch
```



**(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.**

Create the arm.transmission.xacro in arm_description/urdf folder:

```
$ touch arm.trasmission.xacro
```

Inside it, define the xacro:macro to add the hardware interface:

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


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

Define a transmission and an actuator for each manipulator’s joint:

arm_description > urdf > 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:=PositionJointInterface robot_name:=arm">

   <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>


    <transmission name="${robot_name}_tran_1">
       <robotNamespace>/${robot_name}</robotNamespace>
      <type>transmission_interface/SimpleTransmission</type>
      <joint name="j1">
        <hardwareInterface>hardware_interface/${hardware_interface}</hardwareInterface>
      </joint>
      <actuator name="${robot_name}_motor_1">
        <hardwareInterface>hardware_interface/${hardware_interface}</hardwareInterface>
        <mechanicalReduction>1</mechanicalReduction>
      </actuator>
    </transmission>

     <transmission name="${robot_name}_tran_2">
       <robotNamespace>/${robot_name}</robotNamespace>
      <type>transmission_interface/SimpleTransmission</type>
      <joint name="j2">
        <hardwareInterface>hardware_interface/${hardware_interface}</hardwareInterface>
      </joint>
      <actuator name="${robot_name}_motor_2">
        <hardwareInterface>hardware_interface/${hardware_interface}</hardwareInterface>
        <mechanicalReduction>1</mechanicalReduction>
      </actuator>
    </transmission>
```

(Remember to close the xacro:macro at the end with </xacro:macro>.

Then, include arm.transmission.xacro in arm.urdf.xacro:

Help

arm.urdf.xacro U ● arm.transmission.xacro U

arm_description > urdf > arm.urdf.xacro

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

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

<xacro:include filename="$(find arm_description)/urdf/arm.gazebo.xacro"/>
<xacro:include filename="$(find arm_description)/urdf/arm.transmission.xacro"/>
```

(Remember to add the tag <xacro:arm_transmission> at the end of this file.

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

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:

```
luca@Luca:~/catkin_ws/src$ catkin_create_pkg arm_control
Created file arm_control/package.xml
Created file arm_control/CMakeLists.txt
Successfully created files in /home/luca/catkin_ws/src/arm_control. Please adjust the values in package.xml.
luca@Luca:~/catkin_ws/src$ ls
arm_control  arm_description  arm_gazebo  iiwa_stack  my_package  ros_tutorials
luca@Luca:~/catkin_ws/src$ cd arm_control
luca@Luca:~/catkin_ws/src/arm_control$ mkdir launch
luca@Luca:~/catkin_ws/src/arm_control$ ls
CMakeLists.txt  launch  package.xml
luca@Luca:~/catkin_ws/src/arm_control$ cd launch
luca@Luca:~/catkin_ws/src/arm_control/launch$ touch arm_control.launch
luca@Luca:~/catkin_ws/src/arm_control/launch$ ls
arm_control.launch
luca@Luca:~/catkin_ws/src/arm_control/launch$ cd ..
luca@Luca:~/catkin_ws/src/arm_control$ ls
CMakeLists.txt  launch  package.xml
luca@Luca:~/catkin_ws/src/arm_control$ mkdir
mkdir: operando mancante
Try 'mkdir --help' for more information.
luca@Luca:~/catkin_ws/src/arm_control$ mkdir config
luca@Luca:~/catkin_ws/src/arm_control$ ls
CMakeLists.txt  config  launch  package.xml
luca@Luca:~/catkin_ws/src/arm_control$ cd config
luca@Luca:~/catkin_ws/src/arm_control/config$ touch arm_control.yaml
luca@Luca:~/catkin_ws/src/arm_control/config$ ls
arm_control.yaml
luca@Luca:~/catkin_ws/src/arm_control/config$
```

**(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.**

arm_control > launch > arm_control.launch

```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 PositionJointInterface_J0_controller
    PositionJointInterface_J1_controller PositionJointInterface_J2_controller PositionJointInterface_J3_controller"/>
    <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>
```

The default values of the argument “controllers” are the name of needed controllers defined in the .yaml file. To use the parameters in arm_control.yaml it is necessary to use the tag “rosparam” and the command “load”. In the end, add the node “controller_spawner” to spawn controllers.

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

arm_control.launch | arm_control.yaml

arm_control > config > arm_control.yaml

```yaml
#iiwa:
  # 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
```

It is possible to specify the name of each controller, the type and the related joint. In this case it has been chosen a type of controller that allows to provide the motion by input. So, there will be assigned directly the joint variables by the controllers to the joints.

**(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.**

Create an arm_gazebo.launch file into the launch folder of the arm_gazebo package.

```
luca@Luca: ~/catkin_ws/src/arm_gazebo/launch 80x24
luca@Luca:~/catkin_ws$ cd src
luca@Luca:~/catkin_ws/src$ ls
arm_control  arm_description  arm_gazebo  iiwa_stack  my_package  ros_tutorials
luca@Luca:~/catkin_ws/src$ cd arm_gazebo
luca@Luca:~/catkin_ws/src/arm_gazebo$ cd launch
luca@Luca:~/catkin_ws/src/arm_gazebo/launch$ touch arm_gazebo.launch
luca@Luca:~/catkin_ws/src/arm_gazebo/launch$ ls
arm_gazebo.launch  arm_world.launch
luca@Luca:~/catkin_ws/src/arm_gazebo/launch$
```

Load Gazebo world with arm_world.launch and spawning the controllers within arm_control.launch.

```
arm_gazebo > launch > arm_gazebo.launch
<?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>
```

Go to the arm_description package and add the gazebo_ros_control plugin to your main URDF into the arm.gazebo.xacro file.

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

Launch the file:

```
$roslaunch arm_gazebo arm_gazebo.launch
```

```
[ INFO] [1698839228.753177597, 0.350000000]: Loaded gazebo_ros_control.
[INFO] [1698839228.768511, 0.363000]: Controller Spawner: Waiting for service controller_manager/switch_controller
[INFO] [1698839228.777579, 0.369000]: Controller Spawner: Waiting for service controller_manager/unload_controller
[INFO] [1698839228.786929, 0.375000]: Loading controller: joint_state_controller
[INFO] [1698839228.808206, 0.397000]: Loading controller: PositionJointInterface_J0_controller
[INFO] [1698839228.836843, 0.421000]: Loading controller: PositionJointInterface_J1_controller
[INFO] [1698839228.849586, 0.434000]: Loading controller: PositionJointInterface_J2_controller
[INFO] [1698839228.866721, 0.450000]: Loading controller: PositionJointInterface_J3_controller
[INFO] [1698839228.886691, 0.466000]: Controller Spawner: Loaded controllers: joint_state_controller, PositionJointInterface_J0_controller, PositionJointInterface_J1_controller, PositionJointInterface_J2_control
ler, PositionJointInterface_J3_controller
[INFO] [1698839228.896937, 0.475000]: Started controllers: joint_state_controller, PositionJointInterface_J0_controller, PositionJointInterface_J1_controller, PositionJointInterface_J2_controller, PositionJointI
nterface_J3_controller
[urdf_spawner-4] process has finished cleanly
log file: /home/benito/.ros/log/60b5f6c4-78ac-11ee-a9a3-81518a1d4bb7/urdf_spawner-4*.log
```

## 3. Add a camera sensor to your robot

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

In the arm.urdf.xacro file, there were created the requested joint and link. For the newborn camera_joint the parent link is, as requested, the base_link, while the child link is the newborn link_camera, whose geometry has been chosen like a box of suitable dimensions.

```
<joint name="camera_joint" type="fixed">
    <parent link="base_link"/>
    <child link="camera_link"/>
    <origin xyz="0.03 0 -0.013" rpy="3.1415 3.14 0"/>
  </joint>

<link name="camera_link">
    <visual>
      <geometry>
       <box size="0.03 0.03 0.03"/>
      </geometry>
      <origin rpy="0 0 0" xyz="0 0 0"/>
      <material name="green">
        <color rgba="0 1 0 1"/>
      </material>
    </visual>
    <collision>
      <geometry>
          <box size="0.03 0.03 0.03"/>
        </geometry>
      <origin rpy="0 0 0" xyz="0 0 0"/>
    </collision>

    <inertial>
            <mass value="0.0001" />
            <origin xyz="0 0 0" rpy="0 0 ${pi}"/>
            <inertia ixx="0.0000001" ixy="0" ixz="0" iyy="0.0000001" iyz="0" izz="0.0000001" />
        </inertial>
  </link>
```

**(b) In the arm.gazebo.xacro add the gazebo sensor reference tags and the libgazebo_ros_camera plugin to your xacro.**

In order to use a camera, the plugin below must be written in the arm.gazebo.xacro file.

```
<plugin name="camera_controller" filename=" libgazebo_ros_camera.so">
<alwaysOn>true</alwaysOn>
<updateRate>0.0</updateRate>
<cameraName>camera</cameraName>
<imageTopicName>image_raw</imageTopicName>
<cameraInfoTopicName>camera_info</cameraInfoTopicName>
<frameName>camera_link_optical</frameName>
<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>

```

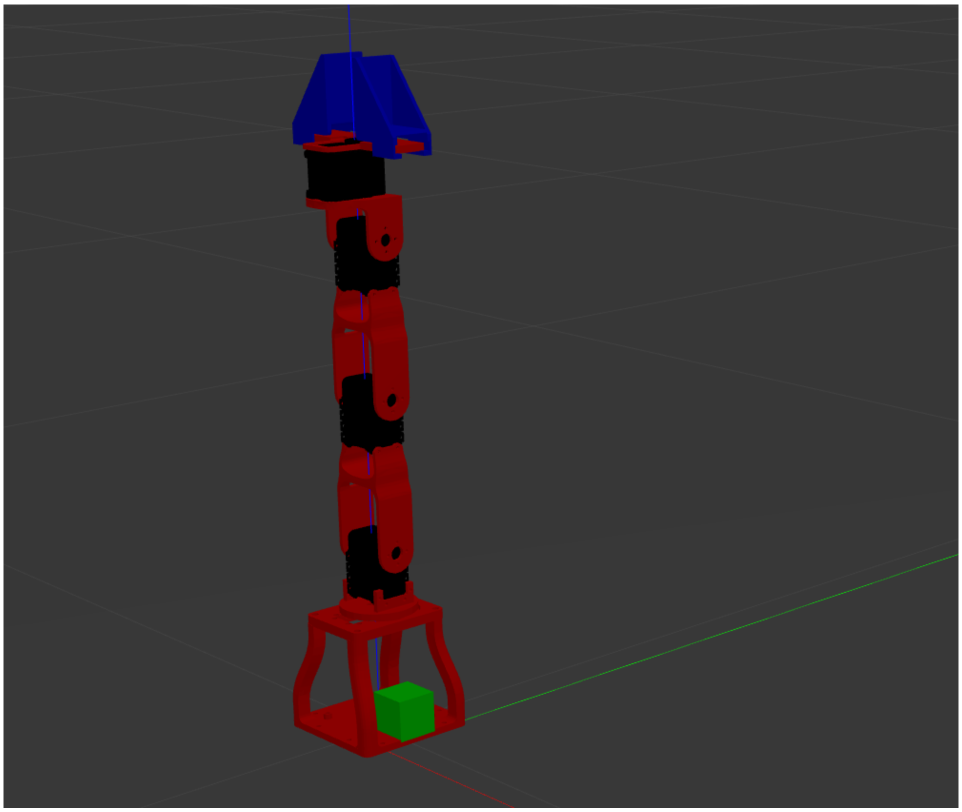
Then, in order to associate the camera_link to a camera rather than an actual link, one must add the following sensor tag and associate it with the gazebo tag of camera_link.

```

<gazebo reference="camera_link">
 <sensor type="camera" name="camera1">
   <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> <mean>0.0</mean> <stddev>0.007
      </stddev>
     </noise>
   </camera>
   <plugin name="camera_controller" filename="libgazebo_ros_camera.so"> ... </plugin>
 </sensor>
</gazebo>
```

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

On the terminal one must launch the command:

```
$ roslaunch arm_gazebo arm_gazebo.launch
```

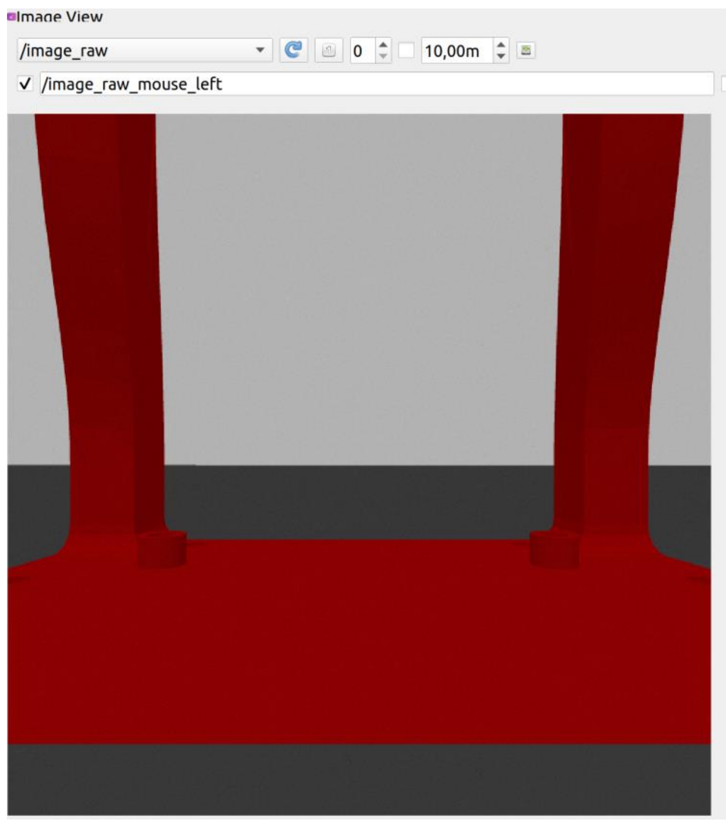


Here is possible to appreciate the camera link, which has been made green to distinguish it from the other links. Launching the command:

```
$ rosrun rqt_image_view rqt_image_view
```

it's possible to see what the camera sees in real-time; this command allows us to see what's inside the /image_raw topic, on which publishes gazebo.

```
12:08:53 domenico@domenico catkin_ws →
rostopic info /image_raw
Type: sensor_msgs/Image

Publishers:
 * /gazebo (http://domenico:39165/)

Subscribers:
 * /rqt_gui_cpp_node_8467 (http://domenico:37831/)
```



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

In the urdf folder of arm_description package it was imported, via terminal, the suggested camera.urdf.xacro file using the command:

```
$ git clone
https://github.com/CentroEPiaggio/irobotcreate2ros/blob/master/model/camera.urdf.xacro.
```

arm_description > urdf > camera.urdf.xacro

```
1  <?xml version="1.0"?>
2  <robot xmlns:xacro="http://www.ros.org/wiki/xacro" name="camera">
3
4      <xacro:property name="pi" value="3.1415926535897931"/>
5
6
7      <xacro:macro name="camera_sensor" params="xyz:='0 -0.03 0' rpy:='1.6 -1.6 0' parent:=crawer_base">
8          <joint name="camera_sensor_joint" type="fixed">
9              <axis xyz="0 1 0" />
10             <origin xyz="${xyz}" rpy="${rpy}"/>
11             <parent link="${parent}"/>
12             <child link="camera_link2"/>
13         </joint>
14
15         <link name="camera_link2">
16             <collision>
17                 <origin xyz="0 0 0" rpy="0 0 0"/>
18                 <geometry>
19                     <box size="0.02 0.08 0.05"/>
20                 </geometry>
21             </collision>
22             <visual>
23                 <origin xyz="0 0 0" rpy="0 0 0"/>
24                 <geometry>
25                     <box size="0.02 0.08 0.05"/>
26                 </geometry>
27                 <material name="iRobot/Green"/>
28             </visual>
29             <inertial>
30                 <mass value="0.0001" />
31                 <origin xyz="0 0 0" rpy="0 0 ${pi}"/>
32                 <inertia ixx="0.0000001" ixy="0" ixz="0" iyy="0.0000001" iyz="0" izz="0.0000001" />
33             </inertial>
```

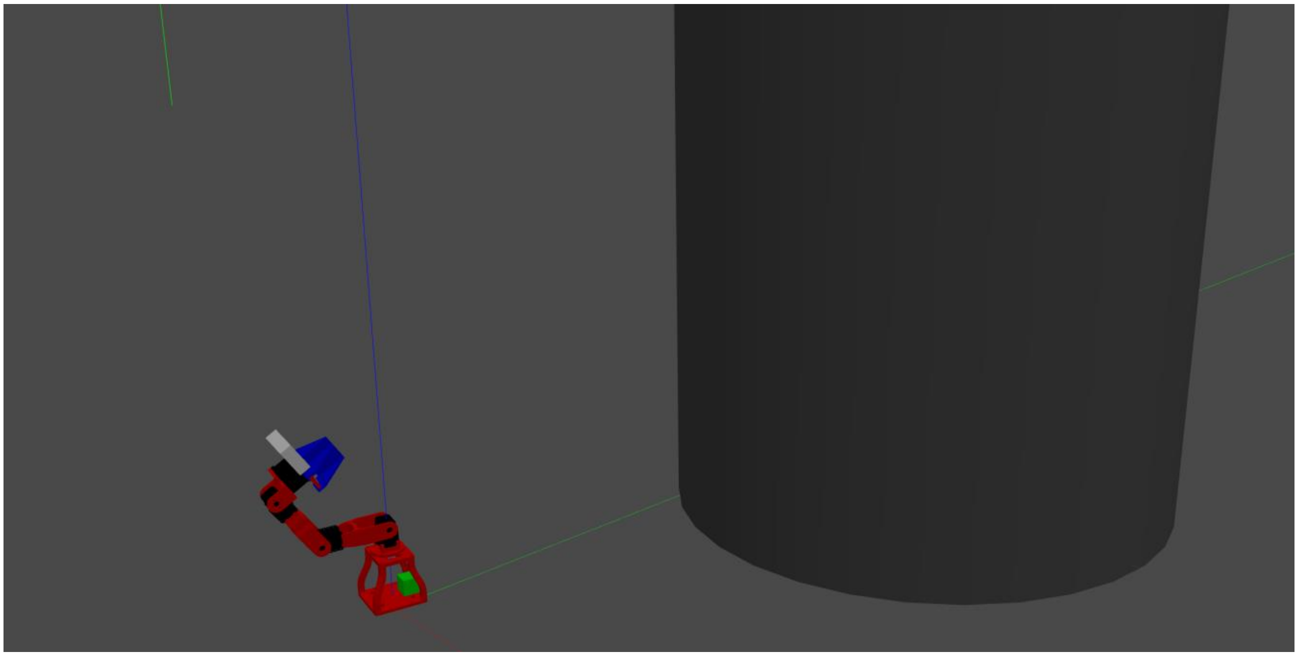
Where both the joint/link definition, the plugin and the reference tags are included, like it was done in the previous section, but in the proper way.

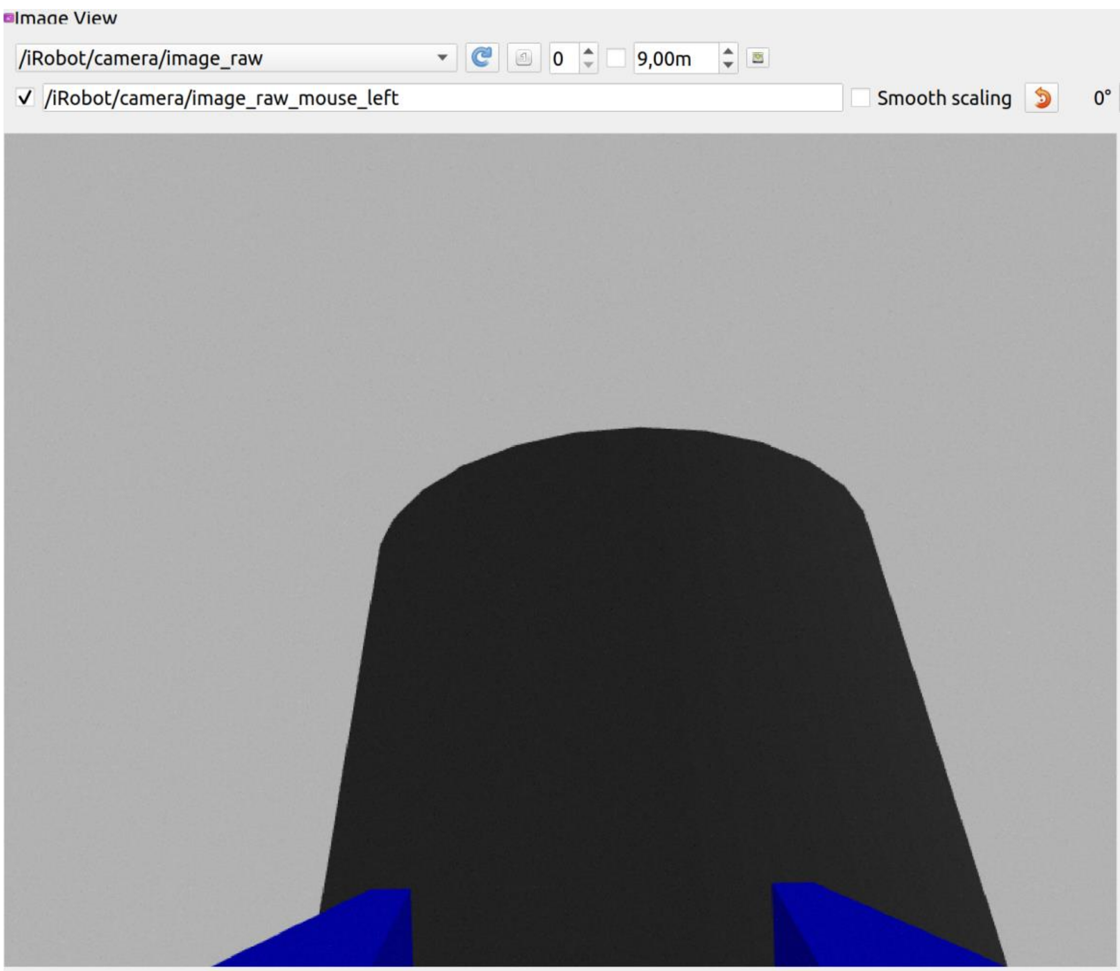
The camera_link has been chosen on the crawer base and observing the point of view of the pinch (see line 7 of previous image). At this point the file must be included in the main arm.urdf.xacro by writing there:

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

And:

<xacro:camera_sensor/>





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

### (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.

The package was created inside the src folder of the workspace using the catkin_create_pkg command:

```
$ catkin_create_pkg pkg_name dependencies
```

Inside this package the arm_controller_node.cpp file was created using the "touch" command.

```
luca@Luca:~/catkin_ws/src$ catkin_create_pkg arm_controller roscpp sensor_msgs s
td_msgs
Created file arm_controller/package.xml
Created file arm_controller/CMakeLists.txt
Created folder arm_controller/include/arm_controller
Created folder arm_controller/src
Successfully created files in /home/luca/catkin_ws/src/arm_controller. Please ad
just the values in package.xml.
luca@Luca:~/catkin_ws/src$ ls
arm_control     arm_description  iiwa_stack  ros_tutorials
arm_controller  arm_gazebo       my_package
luca@Luca:~/catkin_ws/src$ cd arm_controller
luca@Luca:~/catkin_ws/src/arm_controller$ ls
CMakeLists.txt  include  package.xml  src
luca@Luca:~/catkin_ws/src/arm_controller$ cd src
luca@Luca:~/catkin_ws/src/arm_controller/src$ touch arm_controller_node.cpp
luca@Luca:~/catkin_ws/src/arm_controller/src$ ls
arm_controller_node.cpp
luca@Luca:~/catkin_ws/src/arm_controller/src$
```

Everytime a new node is created the CMakeLists file must be modified appropriately:

Add the new .cpp file as an executable

Include the libraries used by the node

```

## Declare a C++ executable
## With catkin_make all packages are built within a single CMake context
## The recommended prefix ensures that target names across packages don't collide
  add_executable(${PROJECT_NAME}_node src/arm_controller_node.cpp)

```

```
## Specify libraries to link a library or executable target against
 target_link_libraries(${PROJECT_NAME}_node
  ${catkin_LIBRARIES}
)
```

## (b) Create a subscriber to the topic joint_states and a callback function that prints the current joint positions.

The topic joint_states is opened by gazebo when the arm_gazebo.launch file is launched. The type of the messages expected by the topic, the publishers and the subscribers can be seen using the command:

```
$ rostopic info arm/joint_states
```

```
debora@debora-Vivobook-ASUSLaptop-X1502ZA-F1502ZA:~/catkin_ws$ rostopic info arm/joint_states
Type: sensor_msgs/JointState

Publishers:
 * /gazebo (http://debora-Vivobook-ASUSLaptop-X1502ZA-F1502ZA:33531/)

Subscribers:
 * /arm/robot_state_publisher (http://debora-Vivobook-ASUSLaptop-X1502ZA-F1502ZA:33131/)
```

The sensor_msgs/JointState message contains several variables, as it's shown by the command:

```
$ rosmsg show sensor_msgs/JointState
```

```
debora@debora-Vivobook-ASUSLaptop-X1502ZA-F1502ZA:~/catkin_ws$ rosmsg show sensor_msgs/JointState
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
string[] name
float64[] position
float64[] velocity
float64[] effort
```

The subscriber node needs a callback function that prints the positions of the joints which is a vector of float64.

arm_controller > src > arm_controller_node.cpp

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

void jointStateCallback(const sensor_msgs::JointState::ConstPtr& msg) {

    ROS_INFO("Joint Positions:");
    for (size_t i = 0; i < msg->position.size(); i++) {
        ROS_INFO("%f", msg->position[i]);
    }
}

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

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


    // Subscriber
    ros::Subscriber sub = n.subscribe("arm/joint_states", 1000, jointStateCallback);
    ros::Rate loop_rate(10);


    while (ros::ok()) {
        ros::spinOnce();
        loop_rate.sleep();
    }

    return 0;
}
```

Let’s run the node; now the arm/joint_states topic has a new subscriber: the arm_controller_node.

```
debora@debora-Vivobook-ASUSLaptop-X1502ZA-F1502ZA:~/catkin_ws$ source devel/setup.bash
debora@debora-Vivobook-ASUSLaptop-X1502ZA-F1502ZA:~/catkin_ws$ rostopic info arm/joint_states
Type: sensor_msgs/JointState

Publishers:
 * /gazebo (http://debora-Vivobook-ASUSLaptop-X1502ZA-F1502ZA:40835/)

Subscribers:
 * /arm/robot_state_publisher (http://debora-Vivobook-ASUSLaptop-X1502ZA-F1502ZA:44201/)
 * /arm_controller_node (http://debora-Vivobook-ASUSLaptop-X1502ZA-F1502ZA:40843/)
```

## (c) Create publishers that write commands onto the controllers' /command topics.

Gazebo also creates some topics devoted to the controllers:

```
debora@debora-Vivobook-ASUSLaptop-X1502ZA-F1502ZA:~/catkin_ws$ rostopic list
/arm/PositionJointInterface_J0_controller/command
/arm/PositionJointInterface_J1_controller/command
/arm/PositionJointInterface_J2_controller/command
/arm/PositionJointInterface_J3_controller/command
/arm/joint_states
```

This topic expects messages of type std_msgs/Float64 containing the reference values for the controller. Let's create 4 publishers (one for each controller) that publish on these topics, in the arm_controller_node.

```cpp
}

int main(int argc, char **argv) {
    ros::init(argc, argv, "arm_controller_node");
    ros::NodeHandle n;


    // Subscribers
    ros::Subscriber sub = n.subscribe("arm/joint_states", 1000, jointStateCallback);

    // Publisher
    ros::Publisher arm_J0_pub = n.advertise<std_msgs::Float64>("/arm/PositionJointInterface_J0_controller/command", 1000);
    ros::Publisher arm_J1_pub = n.advertise<std_msgs::Float64>("/arm/PositionJointInterface_J1_controller/command", 1000);
    ros::Publisher arm_J2_pub = n.advertise<std_msgs::Float64>("/arm/PositionJointInterface_J2_controller/command", 1000);
    ros::Publisher arm_J3_pub = n.advertise<std_msgs::Float64>("/arm/PositionJointInterface_J3_controller/command", 1000);
    ros::Rate loop_rate(10);


    while (ros::ok()) {

 // Creazione di un vettore di 4 elementi di tipo std_msgs::Float64
    std::vector<std_msgs::Float64> msg_vector(4);

    // Assegnazione di valori ai singoli elementi del vettore
    msg_vector[0].data = 1.6;
    msg_vector[1].data = -1;
    msg_vector[2].data = 1.3;
    msg_vector[3].data = -1;

    // Publish to all joint controllers
    arm_J0_pub.publish(msg_vector[0]);
    arm_J1_pub.publish(msg_vector[1]);
    arm_J2_pub.publish(msg_vector[2]);
    arm_J3_pub.publish(msg_vector[3]);

    ros::spinOnce();
    loop_rate.sleep();
   /*In questo modo, ros::spinOnce() all'interno del ciclo while consente di elaborare i messaggi in arrivo sul subscriber
   mentre il nodo continua a pubblicare messaggi con il publisher.
   Questo assicura che il nodo funzioni correttamente sia come subscriber che come publisher.*/

    }

    return 0;
}
```

These publishers are added to the controllers' /command topics after running the code:

```
debora@debora-Vivobook-ASUSLaptop-X1502ZA-F1502ZA: ~/catkin_ws 156x16
debora@debora-Vivobook-ASUSLaptop-X1502ZA-F1502ZA:~/catkin_ws$ rostopic info /arm/PositionJointInterface_J0_controller/command
Type: std_msgs/Float64

Publishers:
 * /arm_controller_node (http://debora-Vivobook-ASUSLaptop-X1502ZA-F1502ZA:40843/)

Subscribers:
 * /gazebo (http://debora-Vivobook-ASUSLaptop-X1502ZA-F1502ZA:40835/)
```

The robot will now get into the posture specified by the controllers, as can be verified reading the joints' Positions printed by the subscriber node:

```
[ INFO] [1698691869.775230079, 2417.221000000]: -1.000000
[ INFO] [1698691869.775260333, 2417.221000000]: Joint Positions:
[ INFO] [1698691869.775280615, 2417.221000000]: 1.600000
[ INFO] [1698691869.775299501, 2417.221000000]: -1.000000
[ INFO] [1698691869.775319361, 2417.221000000]: 1.300000
[ INFO] [1698691869.775347820, 2417.221000000]: -1.000000
debora@debora-Vivobook-ASUSLaptop-X1502ZA-F1502ZA:~/catkin_ws$
```

The posture of the robot also changed in the gazebo simulation: