DIPARTIMENTO DI INGEGNERIA ELETTRICA E TECNOLOGIE DELL'INFORMAZIONE

UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

Robotics Lab

Homework 1

Teacher

Esteemed Prof. Mario Selvaggio

Candidate

Salvatore Del Peschio

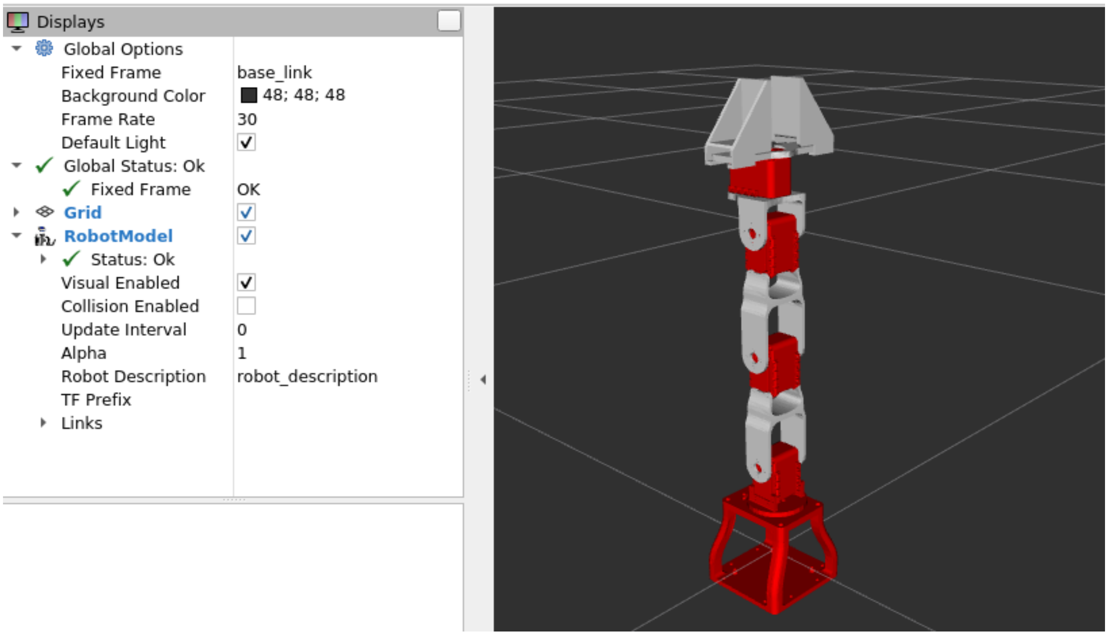
P38000190

1. *Create the description of your robot and visualize it in Rviz:*
   a) *Download the arm_description package from the repo https://github.com/RoboticsLab2023/arm_description.git into your catkin_ws using git commands.*
   $ cd /home/dev/catkin_ws/src/
   $ git clone https://github.com/RoboticsLab2023/arm_description.git
   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, that automatically loads the RobotModel plugin by default, and give it as an argument to your node in the display.launch file.*
   $ cd arm_description
   $ mkdir launch
   $ cd launch
   $ touch display.launch
   The **display.launch** file has been created to load parameters from the **arm.urdf** file. Subsequently, the **robot_state_publisher** and **joint_state_publisher** nodes are executed, and **Rviz** is launched.

display.launch

```
<?xml version="1.0"?>
<launch>
  <!-- Load the URDF as a robot_description parameter -->
  <param name="robot_description" textfile="$(find arm_description)/urdf/arm.urdf"/>

  <!-- Start robot_state_publisher -->
  <node name="robot_state_publisher" pkg="robot_state_publisher" type="robot_state_publisher" />

  <!-- Start joint_state_publisher -->
  <node name="joint_state_publisher" pkg="joint_state_publisher" type="joint_state_publisher" />

  <!-- Start RViz with your custom configuration file -->
  <node name="rviz" pkg="rviz" type="rviz"/>

</launch>
```

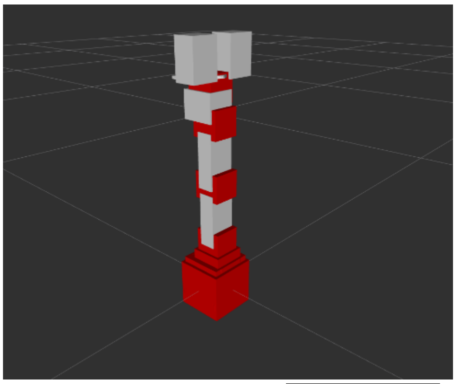
Once the Rviz environment is configured, the robot is displayed as shown in the following image:



A configuration file is saved and loaded by modifying the instruction for launching **Rviz**.

```
<!-- Start RViz with your custom configuration file -->
<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 <box> geometries of reasonable 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.*
By modifying the link's geometry within the collision field, **<box>** elements have been set to simplify the computational aspect of collisions detection. The exact measurements of these **<box>** elements were obtained using a slicer program.



arm.urdf

```
...
<link name="base_turn_rot">
  <visual>
    <geometry>
      <mesh filename="package://arm_description/meshes/base_turn_table_rot.stl" scale="0.001 0.001 0.001"/>
    </geometry>

  </visual>
  <collision>
    <geometry>
        <box size="0.06 0.06 0.021"/>
      </geometry>
    <origin rpy="0 0 0" xyz="0 0 0"/>
  </collision>
...
```

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.*
Within the **arm_description** package, the file **arm.gazebo.xacro** was created, containing all the **<gazebo>** tags. The **arm.urdf** file was renamed to **arm.urdf.xacro** and modified to include **arm.gazebo.xacro** and invoke the macro **arm_gazebo**.

arm.gazebo.xacro

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

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

  <xacro:macro name="arm_gazebo">

    <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>
...
```

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"/>
...
  <!-- Gazebo -->
  <xacro:arm_gazebo />

</robot>
```

Furthermore, the **display.launch** file was also modified to load the ROS parameter robot_description using xacro

display.launch

```
...
  <!-- Load the URDF as a robot_description parameter -->
  <param name="robot_description" command="$(find xacro)/xacro '$(find arm_description)/urdf/arm.urdf.xacro'"/>
...
```

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

   a) *Create a package named arm_gazebo.*

   $ cd /home/dev/catkin_ws/src/
   $ catkin_create_pkg arm_gazebo

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

   $ cd arm_gazebo
   $ mkdir launch
   $ cd launch
   $ touch 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. 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.*

arm_world.launch

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

    <!-- Loads the 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="robot_name" default="arm" />

    <!-- We resume the logic in empty_world.launch, changing only the name of the world to be launched -->
    <include file="$(find gazebo_ros)/launch/empty_world.launch">
        <arg name="world_name" value="$(find arm_gazebo)/worlds/arm.world"/>
        <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="$(find arm_description)/launch/$(arg robot_name)_upload.launch" />

    <!-- 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>
```

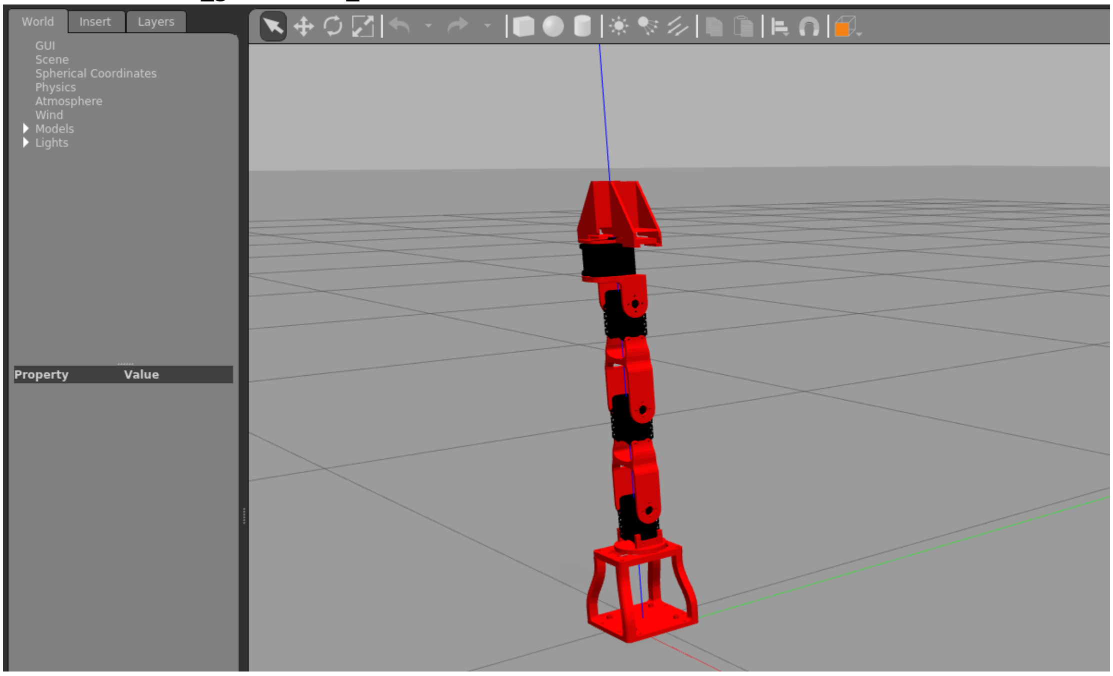
The **arm_world.launch** file was created to launch **Gazebo**, loading the **arm.world** world and the ROS parameter robot description using the **arm_upload.launch** launch file, following the structure of the **iiwa_stack** package.

arm_upload.launch

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

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

</launch>
```

By using the launch file with the following command, the robot is correctly loaded into the **Gazebo** physics engine:

```
$ roslaunch 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.*

arm.transmission.xacro

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

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

  <xacro:macro name="arm_transmission">

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

    <transmission name="$(arg robot_name)_tran_0">
      <robotNamespace>/$(arg robot_name)</robotNamespace>
      <type>transmission_interface/SimpleTransmission</type>
      <joint name="j0">
        <hardwareInterface>hardware_interface/$(arg hardware_interface)</hardwareInterface>
      </joint>
      <actuator name="$(arg robot_name)_motor_0">
        <hardwareInterface>hardware_interface/$(arg hardware_interface)</hardwareInterface>
        <mechanicalReduction>1</mechanicalReduction>
      </actuator>
    </transmission>
...
```

The **arm.urdf.xacro** file was modified to include **arm.transmission.xacro** and invoking the macro. The **arm_world.launch** launch file was executed again using the previous command, and the robot is correctly visualized.

arm.urdf.xacro

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

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

  <xacro:include filename="$(find arm_description)/urdf/arm.gazebo.xacro"/>
  <xacro:include filename="$(find arm_description)/urdf/arm.transmission.xacro"/>
...
  <!-- Gazebo -->
  <xacro:arm_gazebo />
  <!-- Transmissions -->
  <xacro:arm_transmission />


</robot>
```

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.*

```
$ cd /home/dev/catkin_ws/src/
$ catkin_create_pkg arm_control
$ mkdir launch
$ mkdir config
$ touch launch/arm_control.launch
$ touch config/arm_control.yaml
```

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.*

Following the example from the **iiwa_control.launch** file, the parameters from the **arm_control.yaml** file are loaded. Using the **controller_spawner** node, a controller for each joint is launched along with the joint state controller

arm_control.launch

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

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

    <!-- Loads the controllers -->
    <node name="controller_spawner" pkg="controller_manager" type="spawner" respawn="false"
          output="screen" ns="arm" args="joint0_position_controller joint1_position_controller joint2_position_controller joint3_po

    <!-- 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="/arm/joint_states" />
    </node>

</launch>
```

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

The image on the next page shows the file where, for each joint, a **JointPositionController** has been added, and a **joint_state controller** is loaded with a publish rate of 50.

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_control.yaml

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

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

  joint0_position_controller:
    type: position_controllers/JointPositionController
    joint: j0
    pid: {p: 100.0, i: 0.01, d: 10.0}

  joint1_position_controller:
    type: position_controllers/JointPositionController
    joint: j1
    pid: {p: 100.0, i: 0.01, d: 10.0}

  joint2_position_controller:
    type: position_controllers/JointPositionController
    joint: j2
    pid: {p: 100.0, i: 0.01, d: 10.0}

  joint3_position_controller:
    type: position_controllers/JointPositionController
    joint: j3
    pid: {p: 100.0, i: 0.01, d: 10.0}
```

arm_gazebo.launch

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

    <include file="$(find arm_gazebo)/launch/arm_world.launch" />
    <include file="$(find arm_control)/launch/arm_control.launch" />

</launch>
```

In the **arm_gazebo.launch** file, the **arm_world.launch** launch file has been included to load the robot into the **Gazebo** environment, and the **arm_control.launch** to load the controllers. The **gazebo_ros_control** plugin has been added inside the **arm.gazebo.xacro** file within the **arm_gazebo** macro, which is then invoked in the **arm.urdf.xacro**

arm.gazebo.xacro

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

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

  <xacro:macro name="arm_gazebo">

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

By launching the simulation, you can verify from the terminal that the controllers are loaded correctly.

```
[ INFO] [1698919148.246399925, 0.370000000]: Loaded gazebo_ros_control.
[INFO] [1698919148.322729, 0.450000]: Controller Spawner: Waiting for service controller_manager/swi
tch_controller
[INFO] [1698919148.325717, 0.450000]: Controller Spawner: Waiting for service controller_manager/unl
oad_controller
[INFO] [1698919148.329144, 0.460000]: Loading controller: joint0_position_controller
[INFO] [1698919148.358275, 0.490000]: Loading controller: joint1_position_controller
[INFO] [1698919148.368313, 0.500000]: Loading controller: joint2_position_controller
[INFO] [1698919148.378261, 0.510000]: Loading controller: joint3_position_controller
[urdf_spawner-3] process has finished cleanly
log file: /root/.ros/log/d3f0c1da-7958-11ee-b2b0-000c290dea24/urdf_spawner-3*.log
[INFO] [1698919148.388630, 0.520000]: Loading controller: joint_state_controller
[INFO] [1698919148.408691, 0.540000]: Controller Spawner: Loaded controllers: joint0_position_contro
ller, joint1_position_controller, joint2_position_controller, joint3_position_controller, joint_stat
e_controller
[INFO] [1698919148.418759, 0.550000]: Started controllers: joint0_position_controller, joint1_positi
on_controller, joint2_position_controller, joint3_position_controller, joint_state_controller
```

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 a parent link. Size and position the camera link opportunely.*

A camera link and a joint have been added to the **arm.urdf.xacro** file. The camera is displayed as a white box and has been positioned to frame the robot.

arm.urdf.xacro

```xml
<!-- Camera Joint and link -->
<joint name="camera_joint" type="fixed">
  <parent link="base_link"/>
  <child link="camera_link"/>
  <origin xyz="0.75 0 0.5" rpy="0 0.3 -3.14"/>
</joint>

<link name="camera_link">
  <visual>
      <geometry>
        <box size="0.05 0.05 0.05"/>
      </geometry>
  </visual>
</link>
<!--  -->
```

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

arm.gazebo.xacro

```xml
...
<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">
    <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>
  </sensor>
</gazebo>
...
```

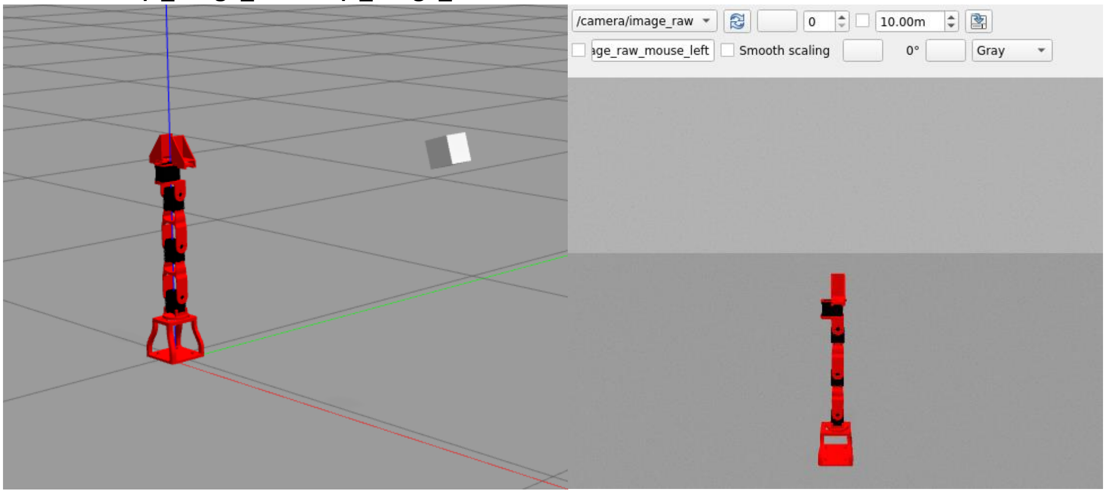
c) *Launch the Gazebo simulation with using arm_gazebo.launch and check if the image topic is correctly published using rqt_image_view.*

The simulation is launched using the following command:

$ roslaunch arm_gazebo arm_gazebo.launch

In another terminal, the **rqt_image_view** node is executed with the following command. After selecting the correct topic for camera visualization, you can verify the camera properly working while framing the robot:

$ rosrun rqt_image_view rqt_image_view



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>.*

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"/>
  <xacro:include filename="$(find arm_description)/urdf/camera.xacro"/>
...
  <!-- Gazebo -->
  <xacro:arm_gazebo />
  <!-- Transmissions -->
  <xacro:arm_transmission />
  <!-- Camera -->
  <xacro:arm_camera />

</robot>
```

arm.camera.xacro

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

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

  <xacro:macro name="arm_camera">

    <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">
        <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>
      </sensor>
    </gazebo>

  </xacro:macro>

</robot>
```

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. Hint: uncomment add_executable and target_link_libraries lines*
   `$ cd /home/dev/catkin_ws/src/`
   `$ catkin_create_pkg arm_controller roscpp sensor_msgs std_msgs`
   The previous commands were executed to create the **arm_controller** package, adding the dependencies. The **CMakeLists.txt** file was modified by uncommenting and modifying the following lines:
   `add_executable(${PROJECT_NAME}_node src/arm_controller_node.cpp)`
   `target_link_libraries(${PROJECT_NAME}_node ${catkin_LIBRARIES})`
   b) *Create a subscriber to the topic joint_states and a callback function that prints the current joint positions (see Slide 45). Note: the topic contains a sensor_msgs/JointState*

arm_controller_node.cpp

```cpp
#include "ros/ros.h"
#include "sensor_msgs/JointState.h"

void jointStateCallback(const sensor_msgs::JointState::ConstPtr& msg) {
    ROS_INFO("Received joint positions:");
    for (size_t i = 0; i < msg->position.size(); i++) {
        ROS_INFO("Joint %ld: %f", i, msg->position[i]);
    }
}

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

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

    // Subscribe to /arm/joint_states with a buffer of size 10
    ros::Subscriber joint_state_sub = nh.subscribe("/arm/joint_states", 10, jointStateCallback);

    ros::spin();

    return 0;
}
```

A subscriber node has been created. When it receives a message on the **/arm/joint_states** topic, it invokes the **jointStateCallback** callback function, which is responsible for printing the positions of all the joints to the terminal.

```
[ INFO] [1698921049.926648609, 1330.350000000]: Received joint positions:
[ INFO] [1698921049.926777746, 1330.350000000]: Joint 0: -0.000000
[ INFO] [1698921049.927022106, 1330.350000000]: Joint 1: -0.000000
[ INFO] [1698921049.927056592, 1330.350000000]: Joint 2: -0.000000
[ INFO] [1698921049.927108303, 1330.350000000]: Joint 3: 0.000000
[ INFO] [1698921049.946745977, 1330.370000000]: Received joint positions:
[ INFO] [1698921049.946865965, 1330.370000000]: Joint 0: -0.000000
[ INFO] [1698921049.947001034, 1330.370000000]: Joint 1: -0.000000
[ INFO] [1698921049.947073409, 1330.370000000]: Joint 2: -0.000000
[ INFO] [1698921049.947318544, 1330.370000000]: Joint 3: 0.000000
```

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

The following lines of code have been added to the **arm_controller_node.cpp** file to publish sinusoids on the ***/command** topics to verify that all the joints are actuated correctly.

```
// Publishers to */command topics
ros::Publisher joint0_pub = nh.advertise<std_msgs::Float64>("/arm/joint0_position_controller/command", 1);
ros::Publisher joint1_pub = nh.advertise<std_msgs::Float64>("/arm/joint1_position_controller/command", 1);
ros::Publisher joint2_pub = nh.advertise<std_msgs::Float64>("/arm/joint2_position_controller/command", 1);
ros::Publisher joint3_pub = nh.advertise<std_msgs::Float64>("/arm/joint3_position_controller/command", 1);
ros::Rate loop_rate(10);

double amplitude = 100; // Amplitude
double frequency = 0.5; // Frequency
double time = ros::Time::now().toSec();
double command = amplitude * sin(2 * M_PI * frequency * time);

while (ros::ok())
{
    time = ros::Time::now().toSec();
    command = amplitude * sin(2 * M_PI * frequency * time);

    std_msgs::Float64 joint0_command;
    joint0_command.data = sin(ros::Time::now().toSec());
    joint0_pub.publish(joint0_command);

    std_msgs::Float64 joint1_command;
    joint1_command.data = cos(ros::Time::now().toSec());
    joint1_pub.publish(joint1_command);

    std_msgs::Float64 joint2_command;
    joint2_command.data = sin(ros::Time::now().toSec());
    joint2_pub.publish(joint2_command);

    std_msgs::Float64 joint3_command;
    joint3_command.data = cos(ros::Time::now().toSec());
    joint3_pub.publish(joint3_command);

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

Once you run the command to launch the node:

`$ rosrun arm_controller arm_controller_node`

You can observe the robot oscillating due to the joint actuation.