UNIVERSITA' DEGLI STUDI DI NAPOLI FEDERICO II

Scuola Politecnica e delle Scienze di Base
Corso di Laurea Magistrale in Ingegneria dell'Automazione e Robotica

Report

HOMEWORK 1

Academic Year 2024/2025

Professor
Mario Selvaggio

Students
Pasquale Farese matr. P38000285
Nello Di Chiaro matr. P38000286
Gianmarco Corrado matr. P38000287
Andrea Colapinto matr. P38000309

Abstract

In this report we want to show how we managed to complete the Homework 1.
The homework is based on four points and each of them represents a different task. Moreover, each point is formed by other sub-points that have guided us through the fulfilling of the homework.

The first task was to download the *arm_description* from *Github* and to make a launch file that loads the model and shows it on *Rviz*.
The second one required to add sensors and controllers to the robot and to spawn it on *Gazebo*.
The third task, instead, asked to mount a camera sensor on our robot and to show the image it publishes.
At the end, the fourth task required to create a node that works as a publisher for commands and as a subscriber for the state of each joint.

Repositories

- Pasquale Farese: `https://github.com/PasFar/Homework-1.git`
- Nello Di Chiaro: `https://github.com/Nellodic34/Homework-1.git`
- Gianmarco Corrado: `https://github.com/giancorr/Homework-1.git`
- Andrea Colapinto: `https://github.com/colandrea02/Homework_1.git`

Contents

Chapter 1

Description of the robot and visualization in Rviz

For what concerns this first part of the homework we used the *git clone* command to download locally the arm_description package from the repository *https://github.com/RoboticsLab2024/ arm_ description.git* into our ROS2 workspace.

Then we created a launch folder containing a launch file named *display.launch.py*; its job is to load the URDF file, that we previously downloaded, as a robot_description ROS param and starts the

robot_state_publisher node, the *joint_state_publisher* node and the *rviz2* node. When we finished, we used the *ros2 launch* command to test it, launching each node. Moreover, the *.rviz* configuration file has been saved and loaded inside the rviz folder, under the config folder. The results are showed in the figure 1.1.

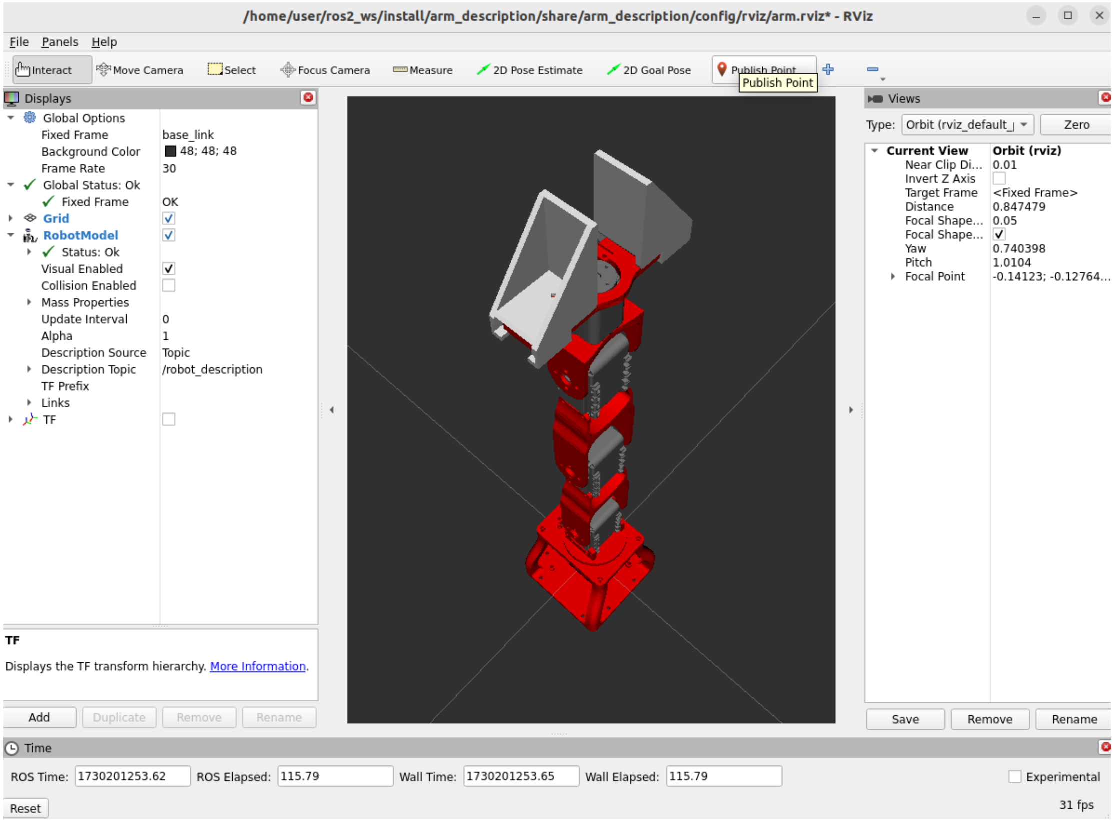

Figure 1.1: Arm showed in Rviz.

At the end, as requested, we substituted the collision meshes of our URDF with primitive shapes. In order to do that we used the collision visualization in rviz to adjust the collision boxes size trying to properly approximate the links shapes; furthermore, we loaded the model parts in *CATIA*, a software for 3d modelling, to misure each part of the arm. We finally added *<box>* geometries with the correct size into the URDF file under the *<collision>* tag, as showed in the figure 1.2.

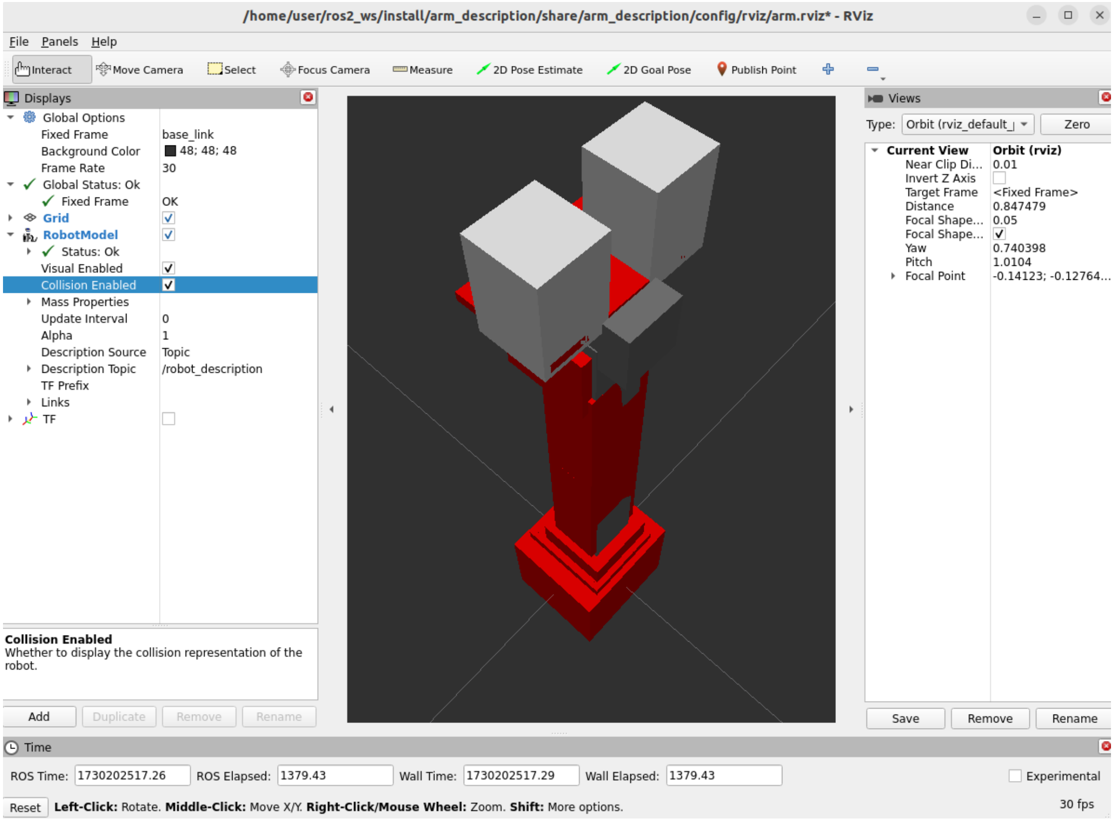

Figure 1.2: Collision boxes showed in Rviz.

Chapter 2

Sensors and controllers and spawning in Gazebo

At first we created a new package named *arm_gazebo*. Inside of this package, we created a launch folder containing a *arm_world.launch.py* file. This launch file has been created to load the URDF to the */robot_description* topic and to spawn the manipulator in Gazebo. In order to do that, this launch file launches the *robot_state_publisher* node, the *joint_state_publisher* node, Gazebo and the spawn entity in it, taking as arguments the robot description topic. The results are displayed in the 2.1 figure.

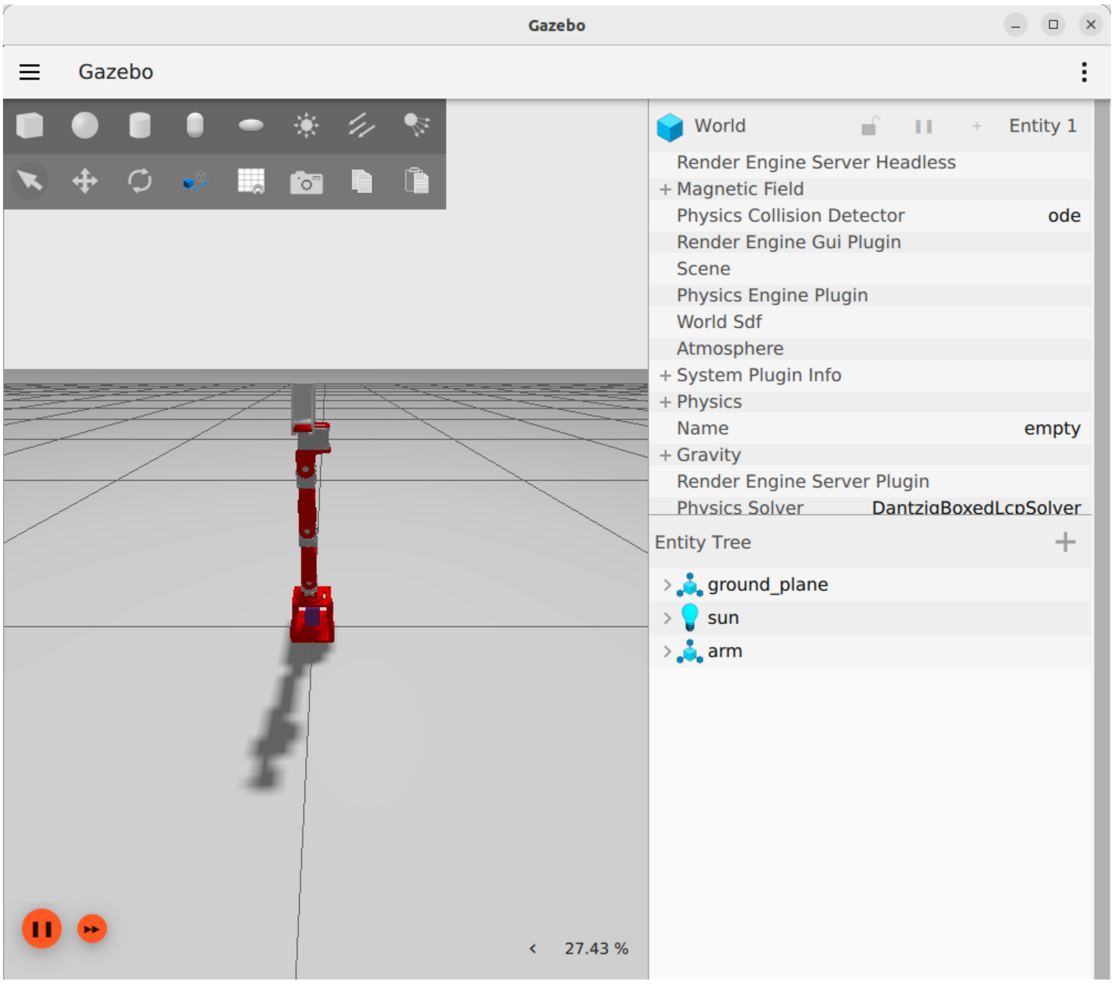

Figure 2.1: Robot spawned in Gazebo.

Then, we had to add a *PositionJointInterface* as a hardware interface using the *<ros2_control>* tag. To accomplish this point, we created an *arm_hardware_interface.xacro* file inside the *urdf/ros2_control* folder.

```xml
<?xml version="1.0" encoding="utf-8"?>
<robot xmlns:xacro="http://www.ros.org/wiki/xacro">


<xacro:macro name="joint_ros2_control" params="name initial_pos">

    <joint name="${name}">
        <command_interface name="position"/>
        <state_interface name="position">
            <param name="initial_value">${initial_pos}</param>
        </state_interface>
        <state_interface name="velocity">
            <param name="initial_value">0.0</param>
        </state_interface>
        <state_interface name="effort">
            <param name="initial_value">0.0</param>
        </state_interface>
    </joint>

</xacro:macro>


</robot>
```

Figure 2.2: Code snippet of *arm_ hardware_ interface.xacro*.

This contains a macro that defines the hardware interface for a generic joint. Then, in the *arm.urdf.xacro* file, we recalled this macro for each joint in order to connect the hardware interface to them.

```xml
<xacro:include filename="$(find arm_description)/urdf/ros2_control/arm_hardware_interface.xacro"/>

<xacro:arg name="j0_pos" default="0.0"/>
<xacro:arg name="j1_pos" default="0.0"/>
<xacro:arg name="j2_pos" default="0.0"/>
<xacro:arg name="j3_pos" default="0.0"/>

<ros2_control name="HardwareInterface_Ignition" type="system">

  <hardware>
    <plugin>ign_ros2_control/IgnitionSystem</plugin>
  </hardware>

  <xacro:joint_ros2_control name="j0" initial_pos="$(arg j0_pos)"/>
  <xacro:joint_ros2_control name="j1" initial_pos="$(arg j1_pos)"/>
  <xacro:joint_ros2_control name="j2" initial_pos="$(arg j2_pos)"/>
  <xacro:joint_ros2_control name="j3" initial_pos="$(arg j3_pos)"/>

</ros2_control>
```

Figure 2.3: Code snippet of *arm.urdf.xacro* - macro call.

Therefore, we added in the *arm.urdf.xacro* file the commands to load the joint controller configuration from a *.yaml* file and spawn the controllers using the *controller_manager* package. To do that we used the plugin tag inside a gazebo tag and specified the file where the controllers are defined.

```xml
<gazebo>
  <plugin filename="ign_ros2_control-system" name="ign_ros2_control::IgnitionROS2ControlPlugin">
  <parameters>$(find arm_control)/config/arm_control.yaml</parameters>
    <controller_manager_prefix_node_name>controller_manager</controller_manager_prefix_node_name>
  </plugin>
</gazebo>
```

Figure 2.4: Code snippet of *arm.urdf.xacro* - configuring the *arm_control.yaml* path.

Then we created an *arm_control* package with an *arm_control.launch.py* file inside its launch folder and then created a config folder containing the *arm_control.yaml* file. We filled the *arm_control.yaml* adding a *joint_state_broadcaster* and a *JointPositionController* to all the joints.

```yaml
controller_manager:
  ros__parameters:
    update_rate: 225  # Hz

    joint_state_broadcaster:
      type: joint_state_broadcaster/JointStateBroadcaster

    position_controller:
      type: position_controllers/JointGroupPositionController

position_controller:
  ros__parameters:
    joints:
      - j0
      - j1
      - j2
      - j3
```

Figure 2.5: Code snippet of *arm_control.yaml*

Finally we created an *arm_gazebo.launch.py* file into the launch folder of the *arm_gazebo* package. This launch file loads the Gazebo world by launching *arm _world.launch.py* and it also spawns the controllers within *arm_control.launch.py*. It is possible to see that the controllers are correctly loaded using the *ros2 control list_controllers* from command line, as showed in the next figure.

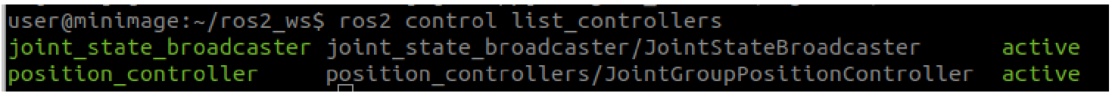

Figure 2.6: List of active controllers.

Chapter 3

Camera sensor

In order to add the camera sensor, we created a *camera.xacro* file containing the description of the camera. In particular, in this file we added the *camera_joint* as fixed, with *base_link* as parent, and a *camera_link*.

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

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

    <joint name="camera_joint" type="fixed">
      <parent link="base_link"/>
      <child link="camera_link"/>
      <origin xyz="0.0 0 0.00" rpy="0.0 0 -1.57"/>
    </joint>

    <link name="camera_link">
      <visual>
        <geometry>
          <box size="0.03 0.03 0.03"/>
        </geometry>
        <material name="purple"/>
        <origin xyz="-0.001 0 0" rpy="0.0 0 0"/>
      </visual>
    </link>

</robot>
```

Figure 3.1: Code snippet of *camera.xacro*

Then, we included this description in the *arm.urdf.xacro* using the *xacro:include* tag. The camera link is defined as a purple box created with the *<box>* tag and positioned on the base link of the robot, as we can see on figure 3.2.

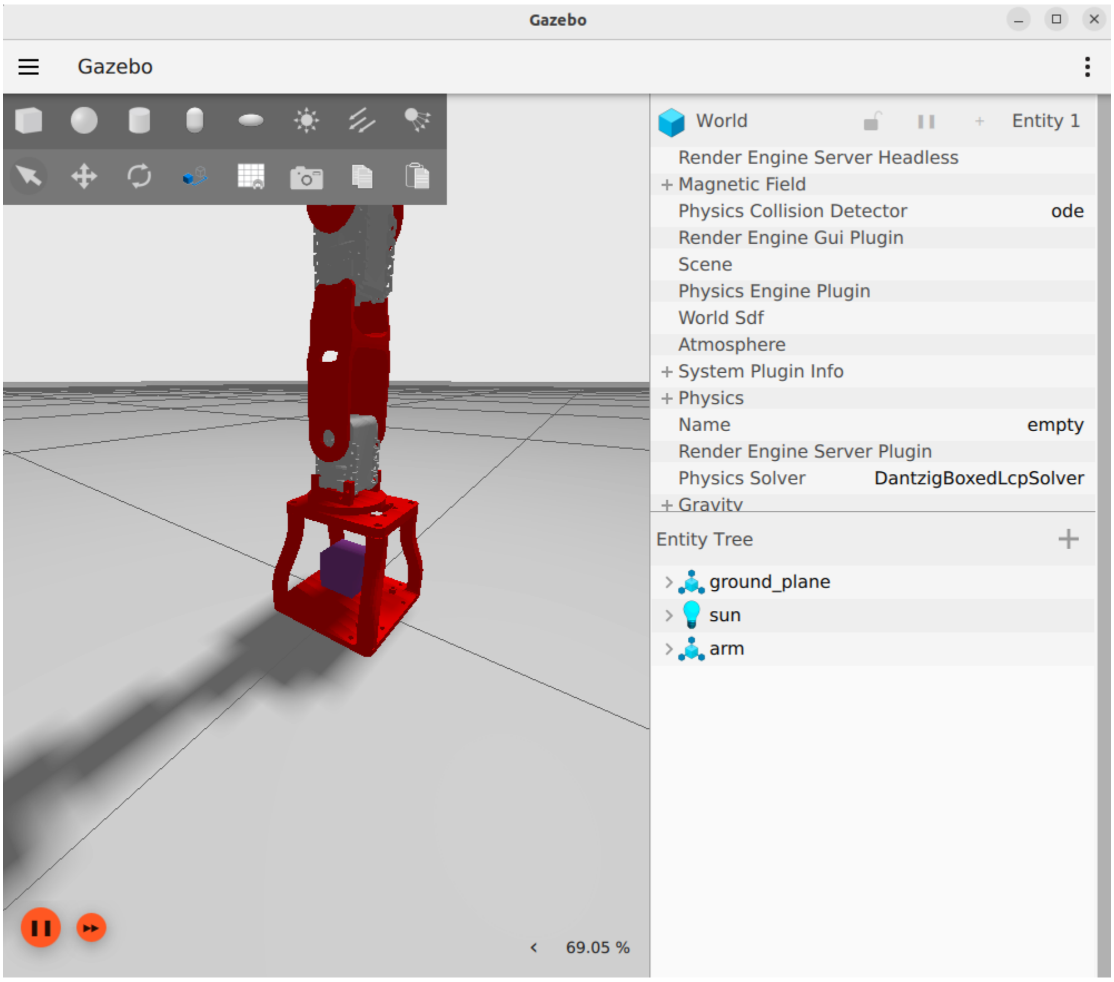

Figure 3.2: Camera link.

Moreover, we added the gazebo sensor reference using the *gz-sim-sensors-system* plugin and in order to make the camera work both on *Gazebo* and on *ROS2*, we added the *ros_ign_bridge*.

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

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

    <joint name="camera_joint" type="fixed">
      <parent link="base_link"/>
      <child link="camera_link"/>
      <origin xyz="0.0 0 0.00" rpy="0.0 0 -1.57"/>
    </joint>

    <link name="camera_link">
      <visual>
        <geometry>
          <box size="0.03 0.03 0.03"/>
        </geometry>
        <material name="purple"/>
        <origin xyz="-0.001 0 0" rpy="0.0 0 0"/>
      </visual>
    </link>

</robot>
```

Figure 3.3: Code snippet of *arm_world.launch.py* - ros_ign_bridge.

Therefore, we added the *image* plugin on *Rviz*. In order to see if the *image* plugin was displaying correctly what the camera is viewing, we added a box in *Gazebo*. The results of this operation is shown in the figure 3.4.

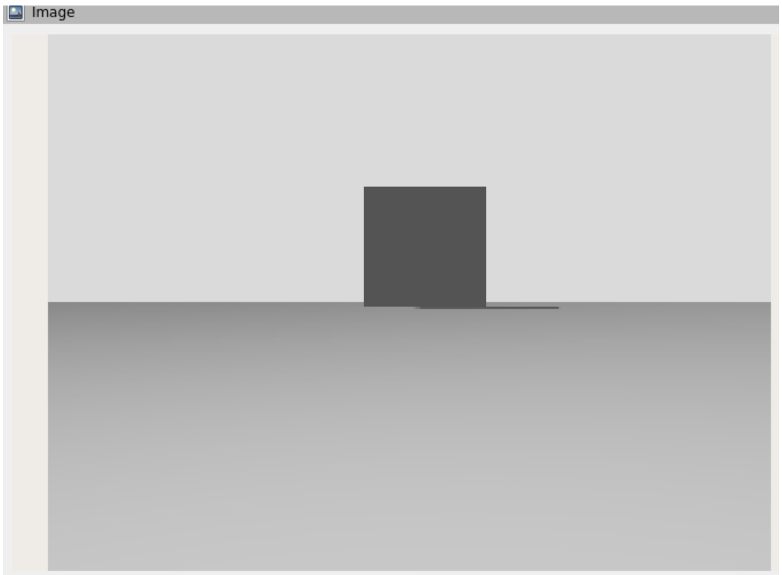

Figure 3.4: Camera view.

Chapter 4

ROS publisher and subscriber

In the last part of the homework we created a *ROS* package called *arm_ controller*, which has inside a *ROS node* called *arm_ controller_ node* which has two principal roles:

- Subscribe to the */joint_ states* topic in order to retrieve the state of each joint.
- Publish to the */position_ controller/commands* in order to publish a position command to each joint, using the position controller previously loaded.

To accomplish this task, we implemented the node with a class which has both the publisher and the subscriber with their related callback functions and a main function. The role of the latter is to

create the node and make it work until the node itself is not stopped. In particular, the subscriber subscribes to the */joint_states* topic and prints the values of position, velocity and effort on screen, specifying the joint it refers to.
The publisher instead publishes on the */position_controller/commands* topic the position command it gets from the *params.yaml*.

Figure 4.1: Log of what the node is printing in the shell.

Moreover, it is possible to change the position command at run-time, changing the parameter using the *ros2 set param* command, specifying the node and the position command. This is made possible by adding a parameter callback function that checks periodically if the parameter has been changed.

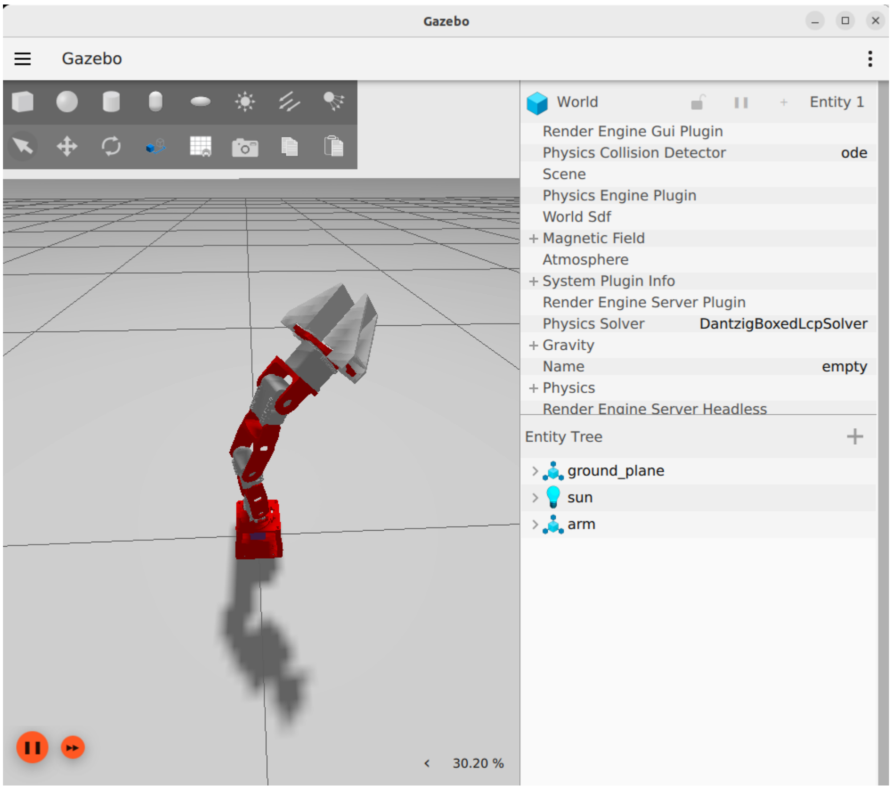

Figure 4.2: Arm pose after sending position command.

In figure 4.2 it is possible to see the arm configuration after publishing the position command with values $[0.4, -0.1, 0.5, 0.4]$.