



UNIVERSITA' DEGLI STUDI DI
NAPOLI FEDERICO II

Scuola Politecnica e delle Scienze di Base
Corso di Laurea Magistrale in Ingegneria Informatica

Robotics Lab Project

HOMEWORK 1

Academic Year 2024/2025

Relatore

Ch.mo prof. Mario Selvaggio

Candidato

Emmanuel Patellaro

P38000239

Repository

https://github.com/EmmanuelPat6/Homework_1.git

Abstract

This project focuses on developing and simulating a 4-degree-of-freedom robotic manipulator within the ROS and Gazebo environments. The primary objective is to create ROS packages that model and control the robot's arm in a virtual setup, enabling the manipulation of joint positions and integration of sensors. First, an Rviz visualization is shown. The assignment requires defining the robot's URDF structure, configuring joint controllers, and adding a camera sensor. The image provided by the camera is shown both in Rviz and through rqt-image-view. At the end, some ROS publishers node are implemented to facilitate real-time joint state commands transmission and a publisher node is implemented to monitor joint positions. This report details each step and methodologies employed in achieving an operation robot simulation, showing the various commands used during the simulation.

Contents

Abstract	1
Arm Description	1
Creation of the robot description and visualization in RViz	1
Collision	8
Gazebo and Controllers	10
GAZEBO	10
Control	13
Camera Sensor	25
Publisher and Subscriber	32

Building a Robot Manipulator

Arm Description

Creation of the robot description and visualization in RViz

To begin, it is necessary to start the Docker container using the ROS2 image that contains all the necessary installations for the operations of this project.

```
./docker_run_container.sh rl24img rl24cont rl24fold
```

Now, let's download from GitHub the **arm_description** package from the repo https://github.com/RoboticsLab2024/arm_description.git. It is necessary to download it in the **src** folder (short for "source") which is a directory within a workspace that contains the source code of the ROS packages you are developing or using.

```
cd src
git clone https://github.com/RoboticsLab2024/arm_description.git
```

Within the package, I created a **launch** folder containing a launch file named **display.launch.py** that loads the **URDF** as a **robot_description** ROS param and starts the **robot_state_publisher** node, the **joint_state_publisher** node, and the **rviz2** node. The instructions used are:

```
cd arm_description
mkdir launch
```

```
cd launch
touch display.launch.py
```

Then it is necessary to modify the **display.launch.py** in such a way to obtain the desired behavior. *Only the most important and significant parts of the written code will be reported, including for the subsequent examples.*

```
def generate_launch_description():
    declared_arguments = []

    arm_description_path = os.path.join(
        get_package_share_directory('arm_description'))

    xacro_arm = os.path.join(arm_description_path,
        "urdf", "arm.urdf.xacro")

    robot_description_arm_xacro =
    {"robot_description": Command(['xacro ', xacro_arm])}
    joint_state_publisher_node = Node(
        package="joint_state_publisher_gui",
        executable="joint_state_publisher_gui",
    )

    robot_state_publisher_node = Node(
        package="robot_state_publisher",
        executable="robot_state_publisher",
        output="both",
        parameters=[robot_description_arm_xacro,
            {"use_sim_time": True},
        ],
    )
```

```
rviz_node = Node(  
    package="rviz2",  
    executable="rviz2",  
    name="rviz2",  
    output="log",  
    arguments=["-d", LaunchConfiguration("rviz_config_file")],  
)  
  
nodes_to_start = [  
    robot_state_publisher_node,  
    joint_state_publisher_node,  
    rviz_node  
]
```

I also created a new folder inside the **config** folder called **rviz** to save the **arm.rviz** file, which will allow starting the visualization in **RViz2** without having to select the appropriate features each time. This file will be saved once the initial setup is completed.

I added in **display.launch.py** this code:

```
declared_arguments.append(  
    DeclareLaunchArgument(  
        "rviz_config_file",  
        default_value=PathJoinSubstitution(  
            [FindPackageShare("arm_description"),  
            "config", "rviz", "arm.rviz"]  
        ),  
        description="RViz config file (absolute path)  
        to use when launching rviz.",  
    )  
)
```

```
)
```

To execute all, it is necessary to give first some important commands. First of all

```
colcon build
```

from the main workspace directory to build all the packages in the **src** directory, and then

```
source install/local_setup.bash
```

to make ROS 2's packages available for me to use them in that terminal. In particular, if I want to build a specific package (useful when I have a lot of packages in the **src** directory), it is possible, in this case, to give the command

```
colcon build --packages-select arm_description
```

Before the building it is important to add to the **CMakeLists.txt** this lines

```
install(  
  DIRECTORY config launch meshes urdf  
  DESTINATION share/${PROJECT_NAME}  
)
```

At this point it is possible to launch the **display.launch.py**

```
install(  
  DIRECTORY config launch meshes urdf  
  DESTINATION share/${PROJECT_NAME}  
)
```

in such a way to allow to the builder to place all the files in the **share** folder, which is where ROS2 do all.

To launch the **display.launch.py** I did

```
ros2 launch arm_description display.launch.py
```


Now, some figures explaining the previously mentioned steps will be shown

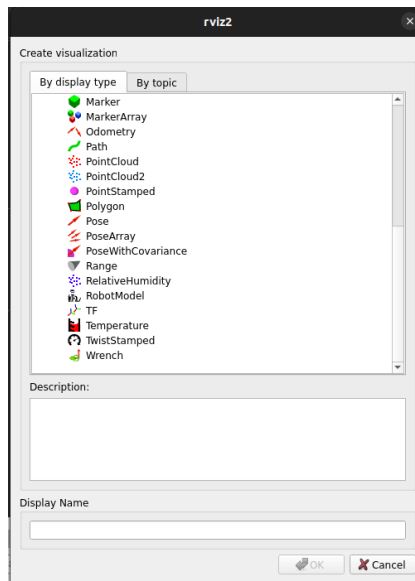


Figure 1: Rviz Options

From this window I added **Robot Model** and **TF**. The **Displays** window must have the following shape:

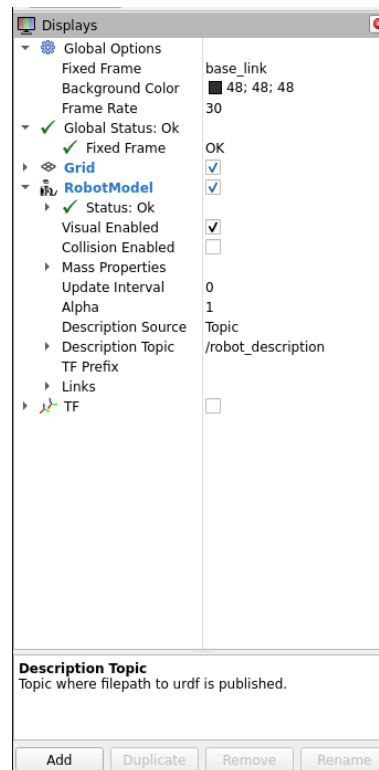


Figure 2: Displays

Thanks to the **Joint State Publisher** I have these sliders, which allow me to control the position of the joints within RViz2

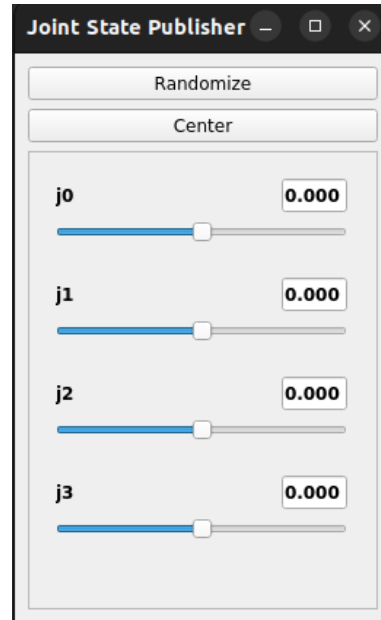


Figure 3: Joint State Publisher

Now, let's look to the robot visualization

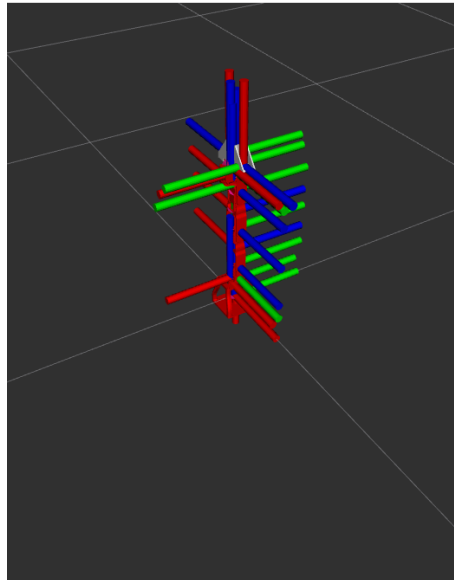


Figure 4: Arm in RViz with TF Active

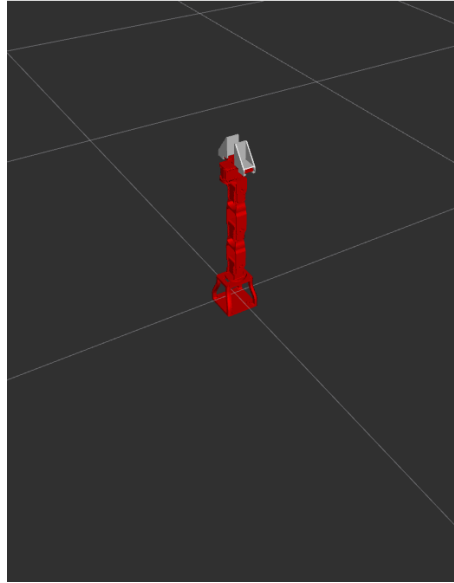


Figure 5: Arm in RViz without TF Active

All that remains is to try moving the sliders to make the robot assume a desired position.

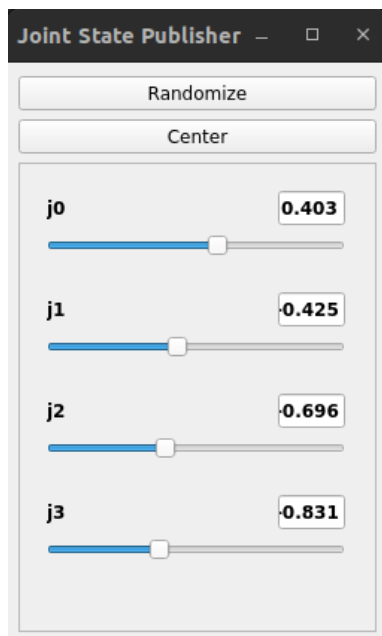


Figure 6: Joint State Publisher for Arm Pose of Figure 7

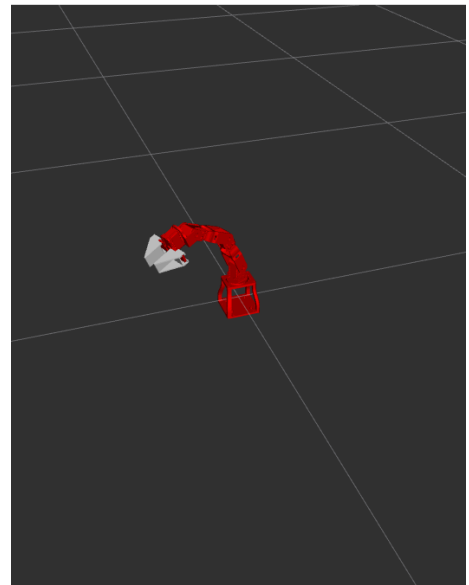


Figure 7: Arm in RViz in a given pose shown in Figure 6

As mentioned earlier, to save the configuration **arm.rviz** in RViz2, it is necessary to follow the steps shown in Figure 8 and choose the correct directory.

In fact, initially, it will direct you to the **install** directory, but you need to navigate to the relevant **rviz** folder within **src/config**.

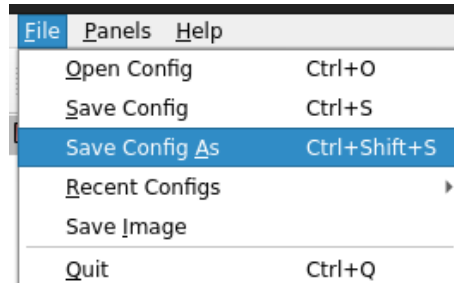


Figure 8: arm.rviz configuration

Collision

Now, let's substitute the collision meshes of the **URDF** file with primitive shapes. In order to do, this I used `<box>` geometries of reasonable size approximating the links. Then, I showed the results adding the **Collision Enabled** from **RobotModel** in the lateral bar. An example of what I did for each link was, for the `base_link`, changing

```
<collision>
  <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"/>
</collision>
```

into

```
<collision>
  <geometry>
    <box size="0.1 0.1 0.08"/>
  </geometry>
</collision>
```

```

</geometry>

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

```

I did this for each link of the robot, appropriately changing the dimensions of the boxes. Now, let's see the results

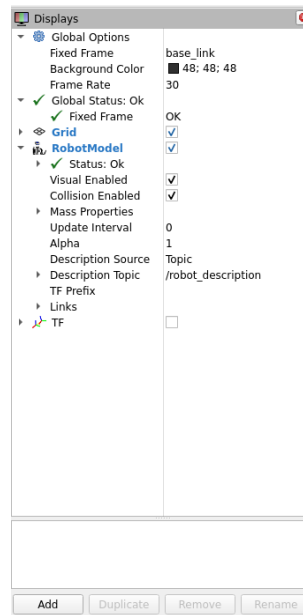


Figure 9: Collision Enabled

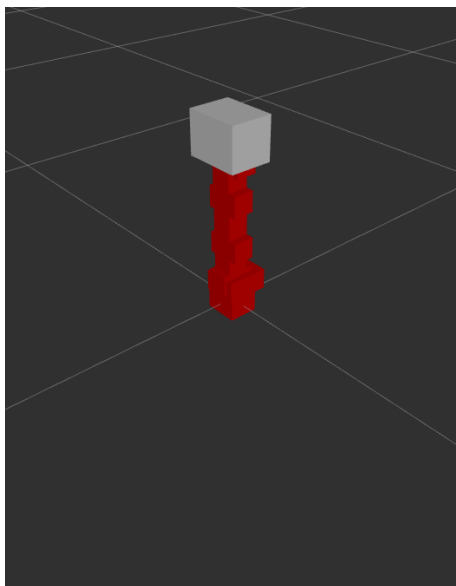


Figure 10: Box Meshes in RViz

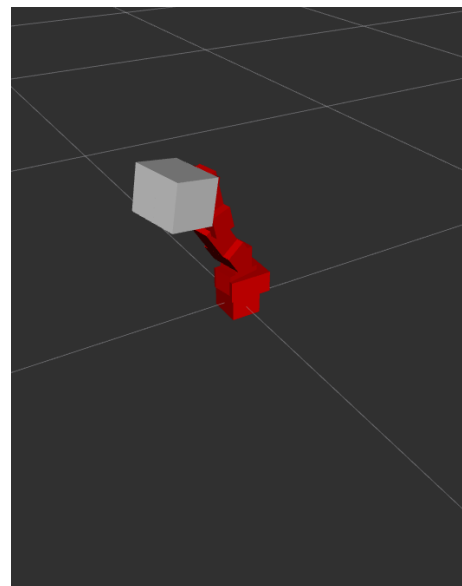


Figure 11: Box Meshes in RViz

Gazebo and Controllers

GAZEBO

Let's now focus on the simulation part of the project and start with the spawning of the robot in Gazebo. I created a package called **arm_gazebo** with these instructions:

```
cd src
ros2 pkg create --build-type ament_cmake
--license Apache-2.0 arm_gazebo
```

Following the same steps as before, I created a new launch file, called **arm_world.launch.py**

```
cd arm_gazebo
mkdir launch
cd launch
touch arm_world.launch.py
```

In the **urdf** folder of **arm_description**, I modified the file **arm.urdf** into **arm.urdf.xacro**, adding, within the **<robot>** tag, the following line

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

Then, I loaded the **URDF** in the launch file using the **xacro** routine. This because **URDF** generation can be scripted with **XACRO** macro. In ROS 2,

XACRO (XML Macros) is used to simplify and manage **URDF (Unified Robot Description Format)** files by enabling the use of macros. This allows you to create modular, reusable components within a robot's description, reducing redundancy and improving readability. By using **XACRO**, you can define parameters, reuse blocks of code, and easily adjust values (such as link sizes, joint properties, and other parameters) across the entire robot model. This makes robot descriptions more manageable, especially for complex robotic systems. To continue, I modified the **package.xml** in **arm_description** folder, adding

```
<export>
  <build_type>ament_cmake</build_type>
  <gazebo_ros gazebo_model_path="${prefix}/.." />
</export>
```

At this point, I need to do a build of both packages (from now on, each time the packages are built will be omitted, as it is necessary to do this every time a modification is made).

```
colcon build --packages-select arm_description arm_gazebo
```

I added, for the package **arm_gazebo**, in its **package.xml** file, the dependency to the building of the **arm_description** package, because the first package needs the file contained in the second one.

```
<build_depend>arm_description</build_depend>
```

Now, it is possible to launch the **arm_world.launch.py** file to spawn the robot in GAZEBO (the contents of this file will be shown in the next section).

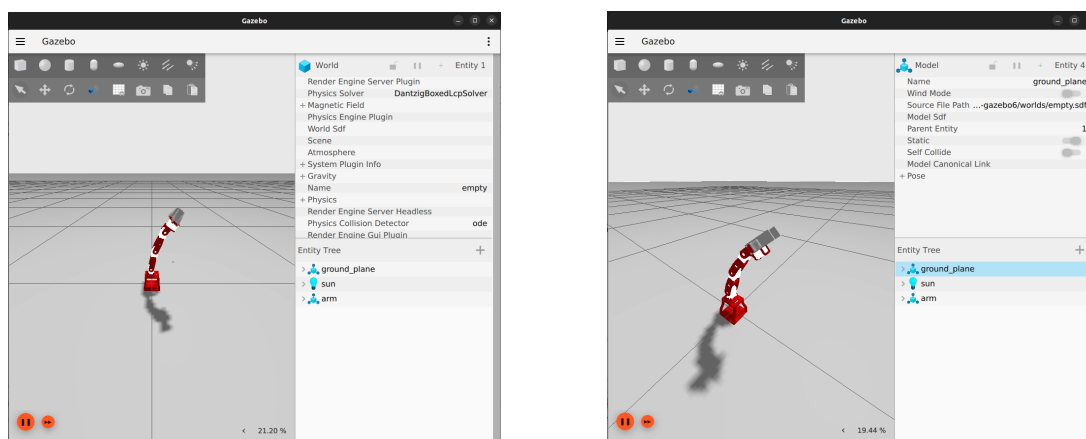


Figure 12: Arm in GAZEBO

Obviously, all drops down because there are no controllers for the moment. So, for this reason, the arm tends to fall to the ground.

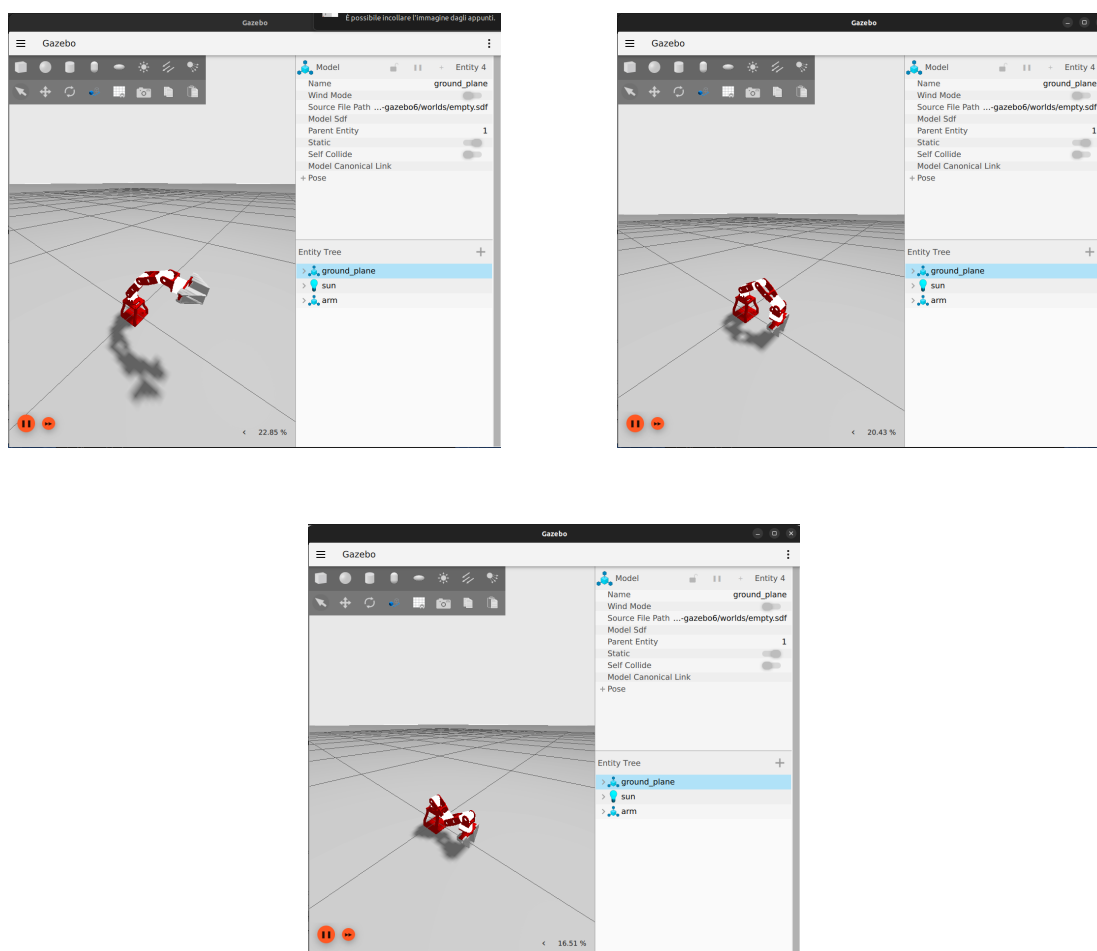


Figure 13: Arm in GAZEBO dropping down

Control

As can be noticed from the previous section, it is necessary to **control** the robot, in order to use it. This allows me to manipulate the robot as needed for the task I have to perform. First of all, I add a **PositionJointInterface** as a **Hardware Interface** to my robot, using **ros2_control**. The hardware components realize communication to physical hardware and represent its abstraction in the **ros2_control framework**. The hardware components are **systems**, **sensors** and **actuators**. So, for this purpose, I created an **arm_hardware_interface.xacro** file in the **arm_description/urdf/ros2_control** folder containing a macro that defines the **Hardware Interface** for the joint. So I did:

```
cd src/arm_description/urdf
mkdir ros2_control
cd ros2_control
touch arm_hardware_interface.xacro
```

This file contains:

```
<?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>
```

To include it in the **arm.urdf.xacro** file I added in this

```
<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>
```

In this way, I can specify in the **arm_world.launch.py** the initial pose of my arm (it is also possible to provide it from the command line). So in the launch file, to do this, it is necessary give this

```
robot_description_arm = {"robot_description":  
Command(['xacro ', xacro_arm, ' j0_pos:=0.0', ' j1_pos:=0.5',  
' j2_pos:=0.5', ' j3_pos:=1.0'])}
```

At this point, the launch file becomes (without reporting all the *import*):

```
def generate_launch_description():  
  
    declared_arguments = []  
  
    arm_description_path = os.path.join(  
        get_package_share_directory('arm_description'))  
  
    xacro_arm = os.path.join(arm_description_path,  
        "urdf", "arm.urdf.xacro")  
  
    robot_description_arm = {"robot_description":  
Command(['xacro ', xacro_arm, ' j0_pos:=0.0',  
' j1_pos:=0.5', ' j2_pos:=0.5', ' j3_pos:=1.0'])}  
  
    robot_state_publisher_node = Node(  
        package="robot_state_publisher",  
        executable="robot_state_publisher",  
        output="both",  
        parameters=[robot_description_arm,
```

```
        {"use_sim_time": True},
    ],
    remappings=[('/robot_description', '/robot_description')]
)

declared_arguments.append(DeclareLaunchArgument('gz_args',
default_value='-r -v 1 empty.sdf', description=
'Arguments for gz_sim'),)

gazebo_ignition = IncludeLaunchDescription(
    PythonLaunchDescriptionSource(
        [PathJoinSubstitution
         ([FindPackageShare('ros_gz_sim'),
          'launch',
          'gz_sim.launch.py'])]),
    launch_arguments={'gz_args':
        LaunchConfiguration('gz_args')}.items()
)

position = [0.0, 0.0, 0.05]

gz_spawn_entity = Node(
    package='ros_gz_sim',
    executable='create',
    output='screen',
    arguments=['-topic', 'robot_description',
               '-name', 'arm',
               '-allow_renaming', 'true',
               "-x", str(position[0]),
```

```
        "-y", str(position[1]),
        "-z", str(position[2]),],
    )

    ign = [gazebo_ignition, gz_spawn_entity]

    nodes_to_start = [
        robot_state_publisher_node,
        *ign,
    ]

    return LaunchDescription(declared_arguments + nodes_to_start)
```

At this point, I created a new package (the last one) called **arm_control** with an **arm_control.launch.py** file inside its **launch** folder and an **arm_control.yaml** file within its **config** folder.

```
cd src
ros2 pkg create --build-type ament_cmake
--license Apache-2.0 arm_control
cd arm_control
mkdir launch
touch arm_control.launch.py
cd ..
mkdir config
cd config
touch arm_control.yaml
```

The **arm_control.yaml** file is the file in which controllers are usually defined. This file contains a list of controllers that the controller manager will load (not necessary the controllers I will use during the simulation). In the

arm_controllers.yaml file I put:

```
controller_manager:
  ros__parameters:
    update_rate: 30  # Hz

    joint_trajectory_controller:
      type: joint_trajectory_controller/JointTrajectoryController

    joint_state_broadcaster:
      type: joint_state_broadcaster/JointStateBroadcaster

    position_controller:
      type: position_controllers/JointGroupPositionController

joint_trajectory_controller:
  ros__parameters:
    command_interfaces:
      - position
    state_interfaces:
      - position
      - velocity
    joints:
      - j0
      - j1
      - j2
      - j3

    state_publish_rate: 200.0
    action_monitor_rate: 20.0
```

```
allow_partial_joints_goal: true
open_loop_control: true
allow_integration_in_goal_trajectories: true

constraints:
  stopped_velocity_tolerance: 0.01
  goal_time: 0.0

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

As it is easy to notice from the code written above, I added in the **arm_control.yaml** file a **joint_state_broadcaster** and a **JointPositionController** to all the joints. Instead, in the **arm_control.launch.py** I inserted

```
def generate_launch_description():
    declared_arguments = []

    joint_state_broadcaster = Node(
        package="controller_manager",
        executable="spawner",
        arguments=["joint_state_broadcaster",
                  "--controller-manager", "/controller_manager"],
```

```
)

position_controller = Node(
    package="controller_manager",
    executable="spawner",
    arguments=["position_controller",
               "--controller-manager", "/controller_manager"],
)

nodes_to_start = [
    joint_state_broadcaster,
    position_controller
]

return LaunchDescription(declared_arguments
+ nodes_to_start)
```

Then, I added in the **arm.urdf.xacro** file

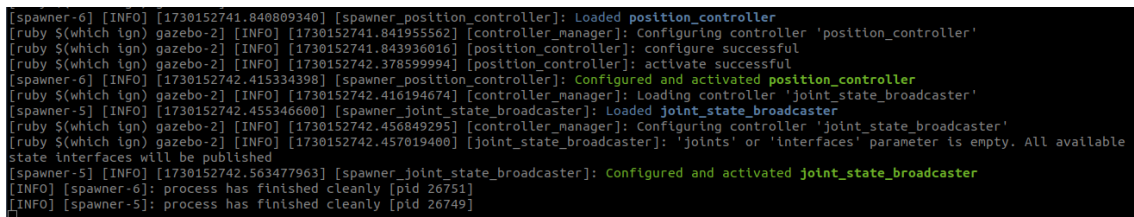
```
<gazebo>
  <plugin filename="ign_ros2_control-system"
    name="ign_ros2_control::IgnitionROS2ControlPlugin">
    <parameters>$(find arm_control)/config/
    arm_control.yaml</parameters>
    <controller_manager_prefix_node_name>controller_manager
    </controller_manager_prefix_node_name>
  </plugin>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

in such a way to load the joint controller configurations from the **arm_control.yaml** file. To conclude this part I created a 'global' launch file,

called `arm_gazebo.launch.py`, into the `launch` folder of the `arm_gazebo` package. This loads the **Gazebo world** with `arm_world.launch.py` and it spawns the **controllers** with `arm_control.launch.py`.

Remember: it is necessary that the controllers spawn after that the robot spawn in Gazebo. For this reason, first the `arm_gazebo.launch.py` must be launched and then, only after this, the `arm_control.launch.py`. All that remains is to launch this file and try to simulate possible movements of the robot using the appropriate commands, which will be shown below.

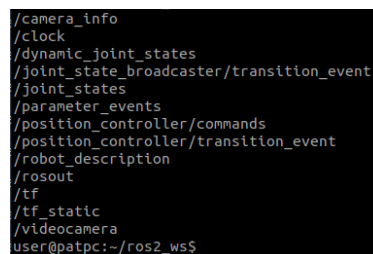
```
ros2 launch arm_gazebo arm_gazebo.launch.py
```



```
[spawnner-6] [INFO] [1730152741.840809340] [spawnner_position_controller]: Loaded position_controller
[ruby $(which ign) gazebo-2] [INFO] [1730152741.841955562] [controller_manager]: Configuring controller 'position_controller'
[ruby $(which ign) gazebo-2] [INFO] [1730152741.843936016] [position_controller]: configure successful
[ruby $(which ign) gazebo-2] [INFO] [1730152742.378599994] [position_controller]: activate successful
[spawnner-6] [INFO] [1730152742.415334398] [spawnner_position_controller]: Configured and activated position_controller
[ruby $(which ign) gazebo-2] [INFO] [1730152742.416194674] [controller_manager]: Loading controller 'joint_state_broadcaster'
[spawnner-5] [INFO] [1730152742.455346600] [spawnner_joint_state_broadcaster]: Loaded joint_state_broadcaster
[ruby $(which ign) gazebo-2] [INFO] [1730152742.456849295] [controller_manager]: Configuring controller 'joint_state_broadcaster'
[ruby $(which ign) gazebo-2] [INFO] [1730152742.457019400] [joint_state_broadcaster]: 'joints' or 'interfaces' parameter is empty. All available
state interfaces will be published
[spawnner-5] [INFO] [1730152742.503477963] [spawnner_joint_state_broadcaster]: Configured and activated joint_state_broadcaster
[INFO] [spawnner-6]: process has finished cleanly [pid 26751]
[INFO] [spawnner-5]: process has finished cleanly [pid 26749]
```

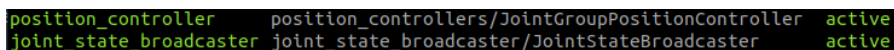
Figure 14: Controllers correctly configured and activated

```
ros2 topic list
```



```
/camera_info
/clock
/dynamic_joint_states
/joint_state_broadcaster/transition_event
/joint_states
/parameter_events
/position_controller/commands
/position_controller/transition_event
/robot_description
/rosout
/tf
/tf_static
/videocamera
user@patpc:~/ros2_ws$
```

```
ros2 control list_controllers
```



```
position_controller    position_controllers/JointGroupPositionController    active
joint_state_broadcaster    joint_state_broadcaster/JointStateBroadcaster    active
```

```
ros2 control list_hardware_interfaces
```

```

command interfaces
j0/position [available] [claimed]
j1/position [available] [claimed]
j2/position [available] [claimed]
j3/position [available] [claimed]
state interfaces
j0/effort
j0/position
j0/velocity
j1/effort
j1/position
j1/velocity
j2/effort
j2/position
j2/velocity
j3/effort
j3/position
j3/velocity

```

Now, to understand the type of commands that I have to give to the robot to do a movement, I can check it in this way

```
ros2 topic info /position_controller/commands
```

```

Type: std_msgs/msg/Float64MultiArray
Publisher count: 0
Subscription count: 1

```

```
ros2 interface show std_msgs/msg/Float64MultiArray
```

```

# This was originally provided as an example message.
# It is deprecated as of Foxy
# It is recommended to create your own semantically meaningful message.
# However if you would like to continue using this please use the equivalent in example_msgs.

# Please look at the MultiArrayLayout message definition for
# documentation on all multiarrays.

MultiArrayLayout layout # specification of data layout
#
#
#
#
MultiArrayDimension[] dim #
  string label #
  uint32 size #
  uint32 stride #
  uint32 data_offset #
float64[] data # array of data

```

So, the command I give to the robot is a

std_msgs/msg/Float64MultiArray. To do this from the terminal, I can **pub** some data (in this case the four joint variables) on the **/position_controller/commands** in this way

```
ros2 topic pub /position_controller/commands
```

```
std_msgs/msg/Float64MultiArray "data: [0.0, 0.0, 0.0, 0.0]"
```

```

publishing #46: std_msgs.msg.Float64MultiArray(layout=std_msgs.msg.MultiArrayLayout(dim=[], data_offset=0),
data=[0.0, 0.0, 0.0, 0.0])

publishing #47: std_msgs.msg.Float64MultiArray(layout=std_msgs.msg.MultiArrayLayout(dim=[], data_offset=0),
data=[0.0, 0.0, 0.0, 0.0])

publishing #48: std_msgs.msg.Float64MultiArray(layout=std_msgs.msg.MultiArrayLayout(dim=[], data_offset=0),
data=[0.0, 0.0, 0.0, 0.0])

publishing #49: std_msgs.msg.Float64MultiArray(layout=std_msgs.msg.MultiArrayLayout(dim=[], data_offset=0),
data=[0.0, 0.0, 0.0, 0.0])

publishing #50: std_msgs.msg.Float64MultiArray(layout=std_msgs.msg.MultiArrayLayout(dim=[], data_offset=0),
data=[0.0, 0.0, 0.0, 0.0])

publishing #51: std_msgs.msg.Float64MultiArray(layout=std_msgs.msg.MultiArrayLayout(dim=[], data_offset=0),
data=[0.0, 0.0, 0.0, 0.0])

publishing #52: std_msgs.msg.Float64MultiArray(layout=std_msgs.msg.MultiArrayLayout(dim=[], data_offset=0),
data=[0.0, 0.0, 0.0, 0.0])

publishing #53: std_msgs.msg.Float64MultiArray(layout=std_msgs.msg.MultiArrayLayout(dim=[], data_offset=0),
data=[0.0, 0.0, 0.0, 0.0])

```

In this specific case, I am instructing the robot to move to the pose $[0.0, 0.0, 0.0, 0.0]$, which is the singularity pose with the arm fully extended upwards. Let's show, with some images, the arm movement:

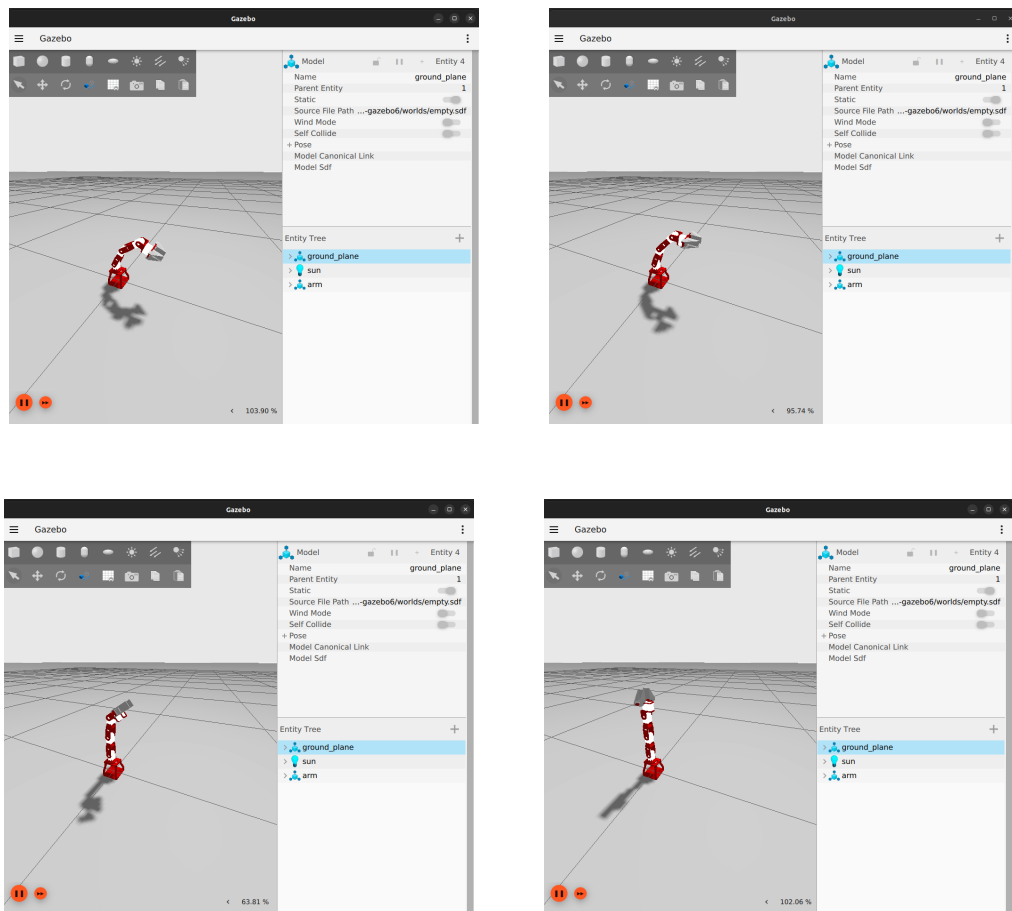


Figure 15: Arm Movement

With

```
ros2 topic echo /joint_states
```

it is possible to see in Real-Time the **joint states** like, for example, position and velocity.

```
stamp:
  sec: 59
  nanosec: 704000000
  frame_id: ''
name:
  j0
  j1
  j2
  j3
position:
  -3.0001532369738245e-10
  0.5000000001908978
  0.5000000003558196
  1.0000000001034208
velocity:
  -2.790583204203615e-19
  9.499410225201505e-14
  5.53383522644393e-14
  9.753420703751552e-14
effort:
  0.0
  0.0
  0.0
  0.0
```

```
stamp:
  sec: 109
  nanosec: 718000000
  frame_id: ''
name:
  j0
  j1
  j2
  j3
position:
  -3.081878006707664e-17
  4.430933510115708e-12
  1.3086993940621888e-11
  1.0100947471239837e-11
velocity:
  -3.4939002396174804e-19
  9.050430574456499e-17
  1.3500392094172337e-16
  3.905620100735581e-17
effort:
  0.0
  0.0
  0.0
  0.0
```

Remember that sometimes it may be useful and necessary to fix the robot to the ground by adding a world link and a world joint. Here, it is not necessary because the robot is well secured to the ground, and the fact that it lifts during movement (when, for example, the gripper pushes too hard against the ground) could be interpreted as a sign of the robot breaking.

Camera Sensor

Now, the goal is to add a camera to my robot so that it can visualize the surrounding environment. First of all, I created a **camera.xacro** file in which I added a **camera_link** and a fixed **camera_joint** with **base_link** as a parent link.

```
cd src/arm_description/urdf
touch camera.xacro
```

which contains

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

  <link name="camera_link">
    <visual>
      <geometry>
        <box size="0.0000010 0.00000003 0.00000003"/>
      </geometry>
      <material name="red"/>
    </visual>
  </link>

  <joint name="camera_joint" type="fixed">
    <parent link="base_link"/>
```

```
    <child link="camera_link"/>
    <origin xyz="0.0 -0.05 0.0" rpy="0.0 -0.2 -1.57"/>
  </joint>

</robot>
```

Then I added it into the **arm.urdf.xacro** file using **<xacro:include>** in this way.

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

Then I created an **arm_camera.xacro** file in the **arm_gazebo/urdf** folder in such a way to add in it the **Gazebo sensor reference tags** and the **gz-sim-sensors-system** plugin to my xacro. A **plugin** is an extension for Gazebo.

```
cd src/arm_gazebo
mkdir urdf
cd urdf
touch arm_camera.xacro
```

The **arm_camera.xacro** file will contain

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

  <gazebo>
    <plugin filename="gz-sim-sensors-system"
      name="gz::sim::systems::Sensors">
      <render_engine>ogre2</render_engine>
    </plugin>
```

```
</gazebo>

<gazebo reference="camera_link">
  <sensor name="camera" type="camera">
    <camera>
      <horizontal_fov>1.047</horizontal_fov>
      <image>
        <width>320</width>
        <height>240</height>
      </image>
      <clip>
        <near>0.1</near>
        <far>100</far>
      </clip>
    </camera>
    <always_on>1</always_on>
    <update_rate>30</update_rate>
    <visualize>true</visualize>
    <topic>camera</topic>
  </sensor>
</gazebo>

</robot>
```

I added it into **arm.urdf.xacro**

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

It is necessary to modify the **CMakeLists.txt** file in the **arm_gazebo** folder

```
install(
```

```
DIRECTORY launch urdf
DESTINATION share/${PROJECT_NAME}
)
```

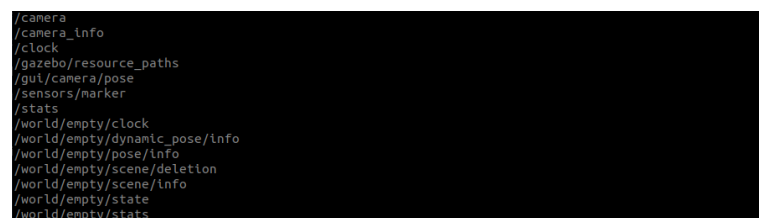
It is important to add in the **arm_world.launch.py** the node for the camera

```
bridge_camera = Node(
    package='ros_ign_bridge',
    executable='parameter_bridge',
    arguments=[
        '/camera@sensor_msgs/msg/Image@gz.msgs.Image',
        '/camera_info@sensor_msgs/msg/
CameraInfo@gz.msgs.CameraInfo',
        '--ros-args',
        '-r', '/camera:=/videocamera',
    ],
    output='screen'
)
```

Notice that I recalled it *videocamera*. With

```
ign topic -l
```

I can see the two topic **/camera** and **/camera_info**.

A terminal window with a black background and white text. It lists various ROS topics. The first two topics are **/camera** and **/camera_info**, which are highlighted in bold. Other topics include /clock, /gazebo/resource_paths, /gui/camera/pose, /sensors/marker, /stats, /world/empty/clock, /world/empty/dynamic_pose/info, /world/empty/pose/info, /world/empty/scene/deletion, /world/empty/scene/info, /world/empty/state, and /world/empty/stats.

```
/camera
/camera_info
/clock
/gazebo/resource_paths
/gui/camera/pose
/sensors/marker
/stats
/world/empty/clock
/world/empty/dynamic_pose/info
/world/empty/pose/info
/world/empty/scene/deletion
/world/empty/scene/info
/world/empty/state
/world/empty/stats
```

With the help of **RViz2**, I chose the orientation of the axis thanks to **TF**, after several attempts.

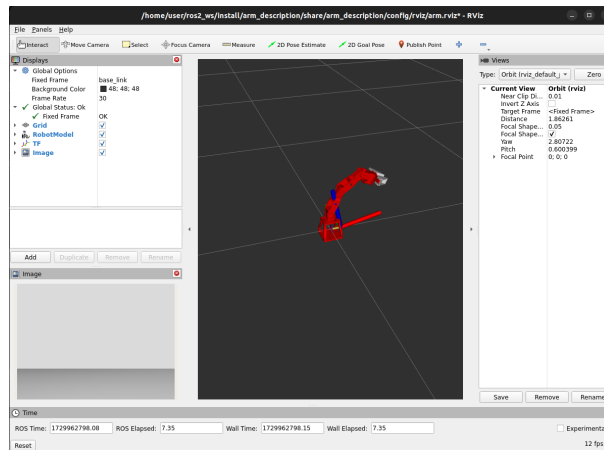


Figure 16: Camera Orientation

The camera will point in the direction of the red axis. It is possible to check the visualization both with **RViz** and **rqt_image_view**. For the first one I need to add **Image** in this way



Figure 17: Camera Options in RViz2

The image visualized by the robot will be shown at the bottom left. In rqt, instead, I need to select

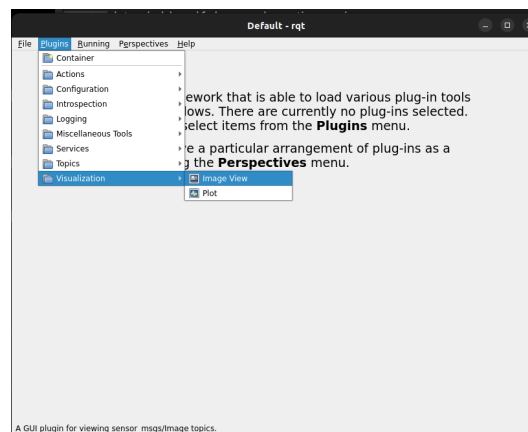


Figure 18: Camera Options in rqt

and then to choose **/videocamera** above **Image View**. After, I added some objects in Gazebo and checked if the camera showed them.

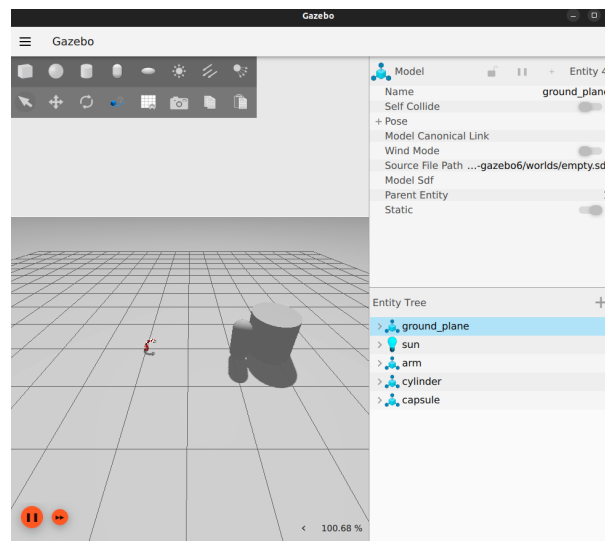


Figure 19: Objects in Gazebo

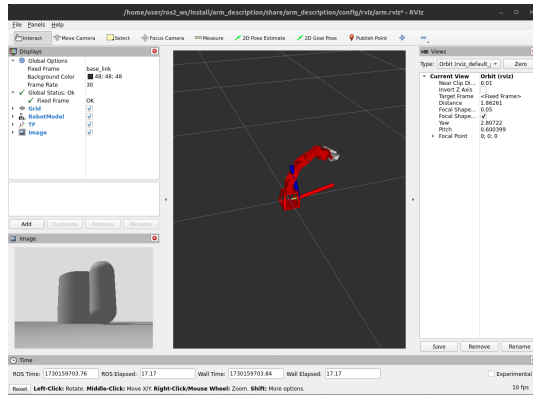


Figure 20: Image in RViz2

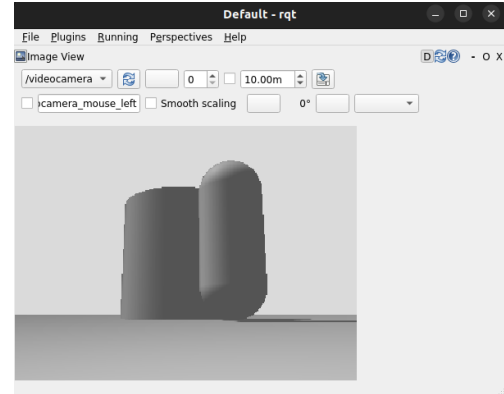


Figure 21: Image in rqt

If I lower the end effector, it's also possible to see it in the image

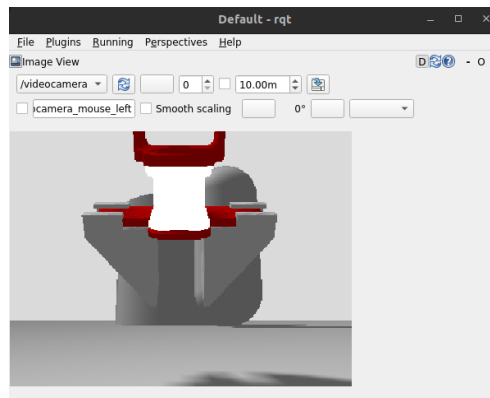


Figure 22: Image with End-Effector and Objects

Publisher and Subscriber

To conclude my project, I created a node which contains a Publisher and a Subscriber. In particular, the first one sends periodically a joint position command to my robot, and the other prints the current joint positions. So, to do this, I created, inside the **arm_controller** package, a **ROS C++ Node**, named **arm_controller_node** containing both.

```
cd src/arm_control
mkdir src (if not present)
cd src
touch arm_controller_node.cpp
```

First of all I modified opportunely the **CMakeLists.txt** file in such a way to be able to compile and execute the node. I added also some dependencies **rclcpp**, **sensor_msgs** and **std_msgs** because of the types of the two messages I have to publish and subscribe. In fact, as previously mentioned, for the **/position_controller/command** topic, the command is a **std_msgs/msg/Float64MultiArray**. Instead, for the **/joint_states** topic, the type is a **sensor_msgs/JointState**. Let's see the last part of the **CMakeLists.txt** file.

```
add_executable(arm_controller src/arm_controller_node.cpp)
ament_target_dependencies(arm_controller
    rclcpp
    sensor_msgs
```

```
std_msgs
)

install(
  TARGETS arm_controller
  DESTINATION lib/${PROJECT_NAME}
)

install(
  DIRECTORY config launch
  DESTINATION share/${PROJECT_NAME}
)

ament_package()
```

Now, all that remains is to appropriately modify the **arm_controller_node.cpp** file in this way:

```
#include <rclcpp/rclcpp.hpp>
#include <sensor_msgs/msg/joint_state.hpp>
#include "std_msgs/msg/float64_multi_array.hpp"
#include <chrono>
#include <functional>
#include <memory>
#include <string>
using namespace std::chrono_literals;

class ArmController : public rclcpp::Node {
public:
  ArmController() : Node("arm_controller") {
    // Subscriber for the topic /joint_states
```

```
subscriber_ = this->create_subscription
<sensor_msgs::msg::JointState>(
    "/joint_states", 10,
    std::bind(&ArmController::jointStateCallback,
    this, std::placeholders::_1));

// Publisher for the topic /position_controller/commands
publisher_ = this->create_publisher
<std_msgs::msg::Float64MultiArray>(
    "/position_controller/commands", 10);
timer_ = this->create_wall_timer(
    500ms, std::bind(&ArmController::jointCommand,
    this));

}

private:
void jointStateCallback(const sensor_msgs::msg::
JointState::SharedPtr msg) {
    // Joint State Values
    RCLCPP_INFO(this->get_logger(), "Joint States:");
    for (size_t i = 0; i < msg->name.size(); ++i) {
        RCLCPP_INFO(this->get_logger(),
            "%s Position: %f",
            msg->name[i].c_str(), msg->position[i]);
    }
}

void jointCommand() {
    auto command_msg = std_msgs::msg::Float64MultiArray();
```

```
        command_msg.data = {1.0, 0.5, 0.1, 0.5};

        // Publish the command
        publisher_->publish(command_msg);
        //RCLCPP_INFO(this->get_logger(),
        // "Publishing command: [%f, %f, %f, %f]",
        // command_msg.data[0],
        // command_msg.data[1], command_msg.data[2],
        // command_msg.data[3]);
    }

    rclcpp::TimerBase::SharedPtr timer_;
    rclcpp::Subscription<sensor_msgs::msg::JointState>::
    SharedPtr subscriber_;
    rclcpp::Publisher<std_msgs::msg::Float64MultiArray>
    ::SharedPtr publisher_;
};

int main(int argc, char * argv[]) {
    rclcpp::init(argc, argv);
    rclcpp::spin(std::make_shared<ArmController>());

    // auto position_publisher =
    //     std::make_shared<ArmController>();
    // position_publisher->jointCommand();

    rclcpp::shutdown();
    return 0;
}
```

As I can clearly see, the publisher prints the values of the joint variables in the

form "**j_n Position: v**", where n is the number of the relative joint (j0, j1, j2 or j3) and v is the joint variable value. Instead, the publisher sends the command on the corresponding topic so that the arm moved in the pose **[1.0, 0.5, 0.1, 0.5]** starting from the one defined in the **arm_world.launch.py** file ([0.0, 0.5, 0.5, 1.0] in the previous example). Now let's run the node in this way (with the launch of the previous chapter already running):

```
ros2 run arm_control arm_controller
```

In the following, I will show the results.

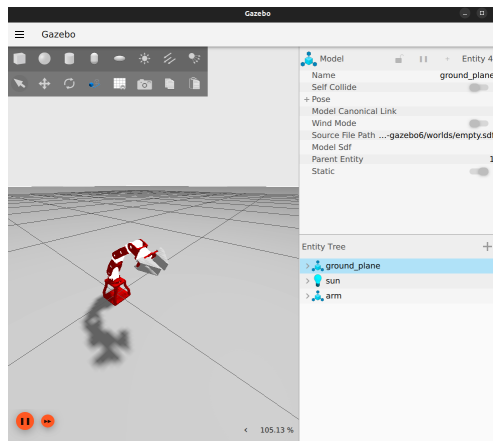


Figure 23: Arm Initial Pose

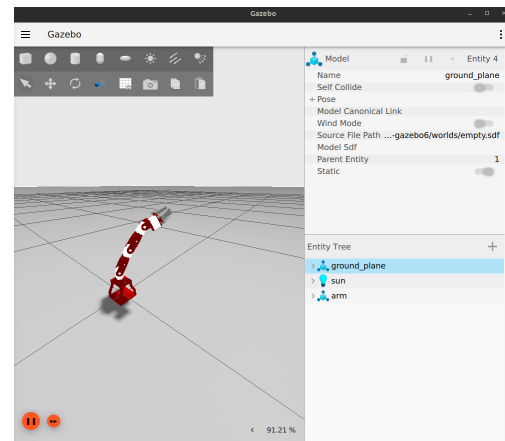


Figure 24: Arm Final Pose

In the terminal, as I said before, I will visualize this during the simulation

```
[INFO] [1730201951.629723391] [arm_controller]: j1 Position: 0.500000
[INFO] [1730201951.629756048] [arm_controller]: j2 Position: 0.330500
[INFO] [1730201951.629871328] [arm_controller]: j3 Position: 0.807338
[INFO] [1730201951.666765833] [arm_controller]: Joint States:
[INFO] [1730201951.666822115] [arm_controller]: j0 Position: 0.227688
[INFO] [1730201951.666864994] [arm_controller]: j1 Position: 0.500000
[INFO] [1730201951.666889820] [arm_controller]: j2 Position: 0.327795
[INFO] [1730201951.666908829] [arm_controller]: j3 Position: 0.804683
[INFO] [1730201951.702629497] [arm_controller]: Joint States:
[INFO] [1730201951.702680756] [arm_controller]: j0 Position: 0.231085
[INFO] [1730201951.702704018] [arm_controller]: j1 Position: 0.500000
[INFO] [1730201951.702730744] [arm_controller]: j2 Position: 0.325085
[INFO] [1730201951.702755096] [arm_controller]: j3 Position: 0.802035
[INFO] [1730201951.742174153] [arm_controller]: Joint States:
[INFO] [1730201951.742294582] [arm_controller]: j0 Position: 0.234483
[INFO] [1730201951.742330255] [arm_controller]: j1 Position: 0.500000
[INFO] [1730201951.742351560] [arm_controller]: j2 Position: 0.322371
[INFO] [1730201951.742370683] [arm_controller]: j3 Position: 0.799395

[INFO] [1730201955.883745561] [arm_controller]: j1 Position: 0.500000
[INFO] [1730201955.883761747] [arm_controller]: j2 Position: 0.101208
[INFO] [1730201955.883780898] [arm_controller]: j3 Position: 0.596904
[INFO] [1730201955.933588348] [arm_controller]: Joint States:
[INFO] [1730201955.933648744] [arm_controller]: j0 Position: 0.567464
[INFO] [1730201955.933679595] [arm_controller]: j1 Position: 0.500000
[INFO] [1730201955.933712506] [arm_controller]: j2 Position: 0.101085
[INFO] [1730201955.933743477] [arm_controller]: j3 Position: 0.595480
[INFO] [1730201955.974043017] [arm_controller]: Joint States:
[INFO] [1730201955.974112413] [arm_controller]: j0 Position: 0.570863
[INFO] [1730201955.974146547] [arm_controller]: j1 Position: 0.500000
[INFO] [1730201955.974176378] [arm_controller]: j2 Position: 0.100974
[INFO] [1730201955.974238659] [arm_controller]: j3 Position: 0.594055
[INFO] [1730201956.014152388] [arm_controller]: Joint States:
[INFO] [1730201956.014215998] [arm_controller]: j0 Position: 0.574262
[INFO] [1730201956.014240769] [arm_controller]: j1 Position: 0.500000
[INFO] [1730201956.014258353] [arm_controller]: j2 Position: 0.100875
[INFO] [1730201956.014275967] [arm_controller]: j3 Position: 0.592628
```



```

[INFO] [1730201959.541557039] [arm_controller]: j1 Position: 0.500000
[INFO] [1730201959.541588929] [arm_controller]: j2 Position: 0.100000
[INFO] [1730201959.541614827] [arm_controller]: j3 Position: 0.500432
[INFO] [1730201959.590965475] [arm_controller]: Joint States:
[INFO] [1730201959.596893826] [arm_controller]: j0 Position: 0.856456
[INFO] [1730201959.596866121] [arm_controller]: j1 Position: 0.500000
[INFO] [1730201959.596130352] [arm_controller]: j2 Position: 0.100000
[INFO] [1730201959.596156400] [arm_controller]: j3 Position: 0.500388
[INFO] [1730201959.633815591] [arm_controller]: Joint States:
[INFO] [1730201959.633867863] [arm_controller]: j0 Position: 0.859856
[INFO] [1730201959.633908363] [arm_controller]: j1 Position: 0.500000
[INFO] [1730201959.633922293] [arm_controller]: j2 Position: 0.100000
[INFO] [1730201959.633945756] [arm_controller]: j3 Position: 0.500348
[INFO] [1730201959.669514597] [arm_controller]: Joint States:
[INFO] [1730201959.669562107] [arm_controller]: j0 Position: 0.863256
[INFO] [1730201959.669579883] [arm_controller]: j1 Position: 0.500000
[INFO] [1730201959.669589727] [arm_controller]: j2 Position: 0.100000
[INFO] [1730201959.669604796] [arm_controller]: j3 Position: 0.500313

[INFO] [1730201967.212712415] [arm_controller]: j1 Position: 0.500000
[INFO] [1730201967.212740955] [arm_controller]: j2 Position: 0.100000
[INFO] [1730201967.212813198] [arm_controller]: j3 Position: 0.500000
[INFO] [1730201967.247750359] [arm_controller]: Joint States:
[INFO] [1730201967.247802442] [arm_controller]: j0 Position: 1.000000
[INFO] [1730201967.247830526] [arm_controller]: j1 Position: 0.500000
[INFO] [1730201967.247854474] [arm_controller]: j2 Position: 0.100000
[INFO] [1730201967.247878018] [arm_controller]: j3 Position: 0.500000
[INFO] [1730201967.302964422] [arm_controller]: Joint States:
[INFO] [1730201967.303022220] [arm_controller]: j0 Position: 1.000000
[INFO] [1730201967.303047170] [arm_controller]: j1 Position: 0.500000
[INFO] [1730201967.303068505] [arm_controller]: j2 Position: 0.100000
[INFO] [1730201967.303084586] [arm_controller]: j3 Position: 0.500000
[INFO] [1730201967.341087013] [arm_controller]: Joint States:
[INFO] [1730201967.341161769] [arm_controller]: j0 Position: 1.000000
[INFO] [1730201967.341197270] [arm_controller]: j1 Position: 0.500000
[INFO] [1730201967.341224897] [arm_controller]: j2 Position: 0.100000
[INFO] [1730201967.341251243] [arm_controller]: j3 Position: 0.500000

```

Figure 25: Subscriber in the Terminal

In addition, I created two other publishers in the **Other_Publishers** folder to enable different behaviors and to allow the robot to perform more movements. One publisher, directly from the terminal, lets me issue joint commands by simply entering the desired joint position values separated by a space, without needing to write the usual **ros2 topic pub** command. The other allows me to set five desired poses, spaced a few seconds apart, so that the robot can reach them sequentially. These can be run in the same way as previously done, with the only difference that they do not include a subscriber within them (the names to execute them are `publisher_terminal` for the first one and `publisher_defined` for the second one). The results of these two new implementations will not be shown as they are outside the scope of this project, but it is possible to verify their functionality by issuing the commands mentioned above.