Homework_01

Bring Up Your Robot

Mario Selvaggio

Federica Pirozzi P38000350: `https://github.com/Federica2103`
Francesco Lionetti P38000365: `https://github.com/FrancescoLionetti`

29 October 2025

Bring up *your* robot

The first objective of this homework is to visualize the 4-DOF manipulator **Armando** in **RViz2**, starting from the provided `armando_description` package, which contains the robot model, meshes, and URDF file.

To begin, download the package into the ROS 2 workspace using the following commands:

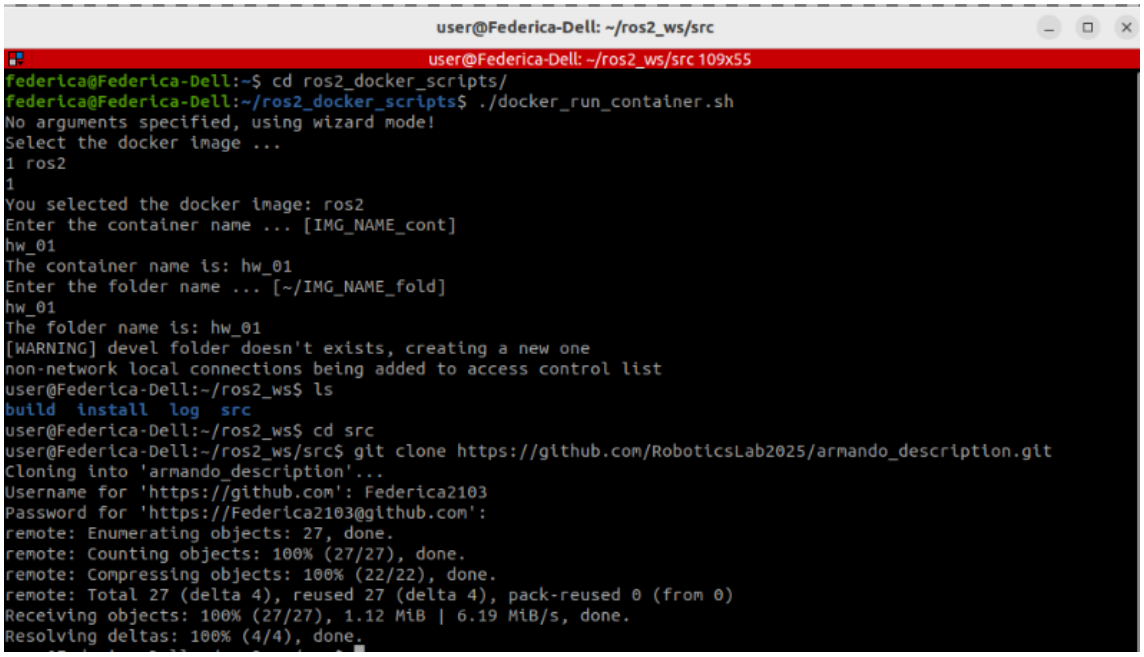


Figura 1: Commands to clone the repositories from github

1. **Modify the URDF description of your robot and visualize it in Rviz.**

   (a) *Request: Create a* `launch` *folder within the* `armando_description` *package containing a launch file named* `armando_display.launch` *that loads the URDF as a* `robot_description` *ROS parameter, starts the* `robot_state_publisher` *node, the* `joint_state_publisher` *node, and the* `rviz2` *node. Launch the file using* `ros2 launch`*.*
   Note: *To visualize your robot in Rviz, you have to change the Fixed Frame in the lateral bar and add the RobotModel display.*

   A new folder named `launch` was created inside the `armando_description` package, containing a Python launch file called `armando_display.launch.py that allows us to start up and configure a number of ROS2 nodes simultaneously`.

   ```
   $ cd armando_description
   $ mkdir launch
   $ cd launch
   $ touch armando_display.launch.py
   ```

   The process begins by locating the robot's URDF (Unified Robot Description Format) file, `arm.urdf`, within the package's shared directory. The entire content of this file is read and loaded into a string, which is then assigned to the `robot_description` parameter, making the robot's model available to all nodes in the ROS system.
   Once the model is loaded, the file simultaneously launches three essential nodes:

- `joint_state_publisher_gui`: This node provides a graphical user interface (GUI) featuring sliders. It allows a user to manually adjust the position of each robot joint and publishes these values to the `/joint_states` topic.
- `robot_state_publisher`: This core node is initialized with the `robot_description` parameter. It subscribes to the `/joint_states` topic (listening for data from the GUI) and uses the robot's kinematic model (defined in the URDF) to calculate and broadcast the 3D pose and orientation of all the robot's links. This information is published as a series of transformations (TF) to the `/tf` topic.
- `rviz2`: This node launches the primary ROS 2 visualization tool. RViz automatically subscribes to the `robot_description` parameter to render the robot's 3D geometry and to the `/tf` topic to correctly position and animate the robot's links based on the data from the `robot_state_publisher`.

armando_display.launch.py

```
def generate_launch_description():

    urdf_file_name = 'arm.urdf'
    urdf = os.path.join(
        get_package_share_directory('armando_description'), '
            urdf',
        urdf_file_name
    )
    with open(urdf, 'r') as infp:
        robot_desc = infp.read()

    params = {"robot_description": robot_desc}

    # Joint State Publisher Node
    joint_state_publisher_node = Node(
        package='joint_state_publisher_gui',
        executable='joint_state_publisher_gui',
    )

    # Robot State Publisher Node
    robot_state_publisher_node = Node(
        package='robot_state_publisher',
        executable='robot_state_publisher',
        output='both',
        parameters=[params],
    )

    # RViz2 Visualization Node
    rviz_node = Node(
        package="rviz2",
        executable="rviz2",
        name="rviz2",
        output="log",
    )

    # Return the Launch Description
    return LaunchDescription( [
        joint_state_publisher_node,
        robot_state_publisher_node,
        rviz_node
    ])
```

The corresponding **CMakeLists.txt** file was updated to ensure that all necessary directories — **launch**, **urdf**, and **meshes** — are properly installed.

CMakeLists.txt

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

In the **package.xml**, dependencies were added to include the required **ROS 2 packages**.

package.xml

```
<exec_depend>robot_state_publisher</exec_depend>
<exec_depend>joint_state_publisher_gui</exec_depend>
<exec_depend>rviz2</exec_depend>
```

After compiling and sourcing the workspace, the launch file can be executed using:

```
$ cd ~/ros2_ws/
$ colcon build
$ source install/setup.bash
$ ros2 launch armando_description armando_display.launch.py
```

At this stage, the manipulator is correctly visualized in **RViz**. By setting the **Fixed Frame** to the **base_link** and adding the **RobotModel** display from the left panel, the robot's structure can be seen and interactively actuated through the joint sliders.

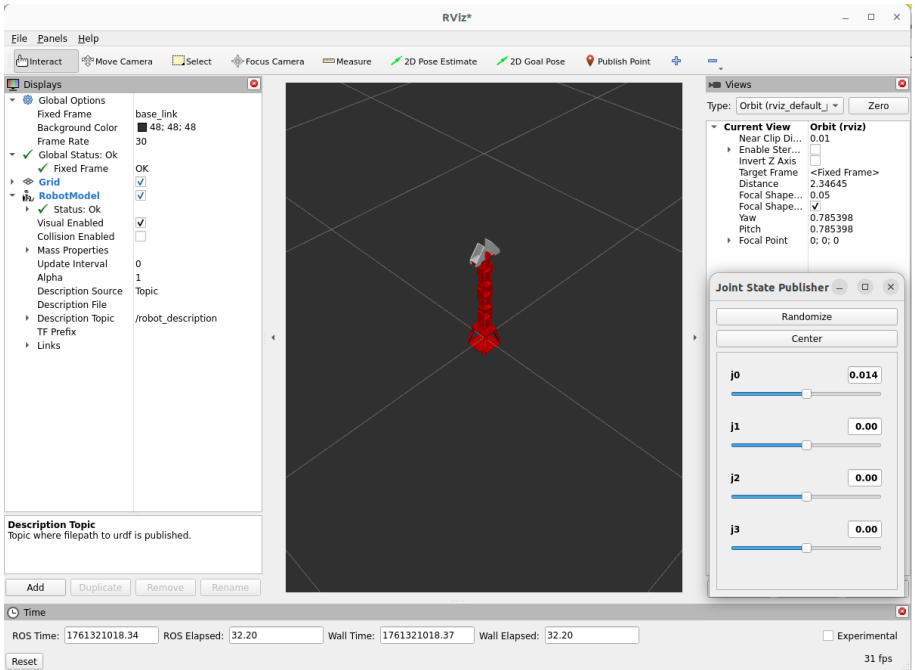


Figura 2: Visualization of Armando in Rzviz

(b) *Request: Create a* `config` *folder and save a* `.rviz` *configuration file within, that automatically loads the* `RobotModel` *display, and pass it as an argument to your node in the* `armando_display.launch` *file.*

To automatically load the correct visualization layout in **RViz**, a configuration file was created. Inside the package, a new folder **config** was added, where a customized **.rviz** configuration file was saved after manually adjusting the visualization parameters.

```
$ cd src
$ cd armando_description
$ mkdir config
```

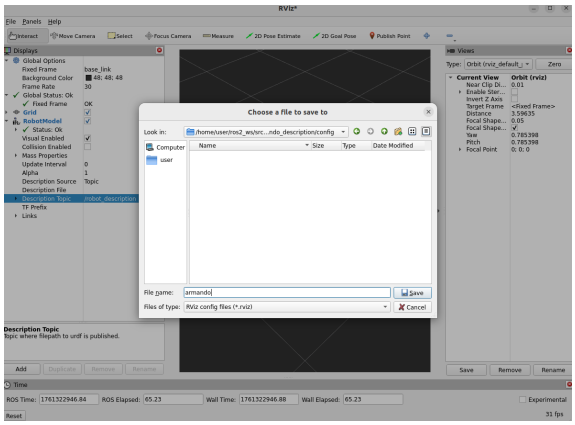


Figura 3: Save configuration

The **CMakeLists.txt** file was modified accordingly to include the new **config** folder.

CMakeLists.txt

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

To load this configuration automatically when launching **RViz**, the following argument was added to the **armando_display.launch.py** file:

armando_display.launch.py

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

```
    )

...

# RViz2 Visualization Node
    rviz_node = Node(
        package="rviz2",
        executable="rviz2",
        name="rviz2",
        output="log",
        arguments=["-d", LaunchConfiguration("
            rviz_config_file")],
    )

# Return the Launch Description
    return LaunchDescription(declared_arguments + [
        joint_state_publisher_node,
        robot_state_publisher_node,
        rviz_node
    ])
```

This modification allows the RViz environment to automatically load the desired layout, saving time and ensuring consistent visualization each time the file is launched.

(c) *Request: Substitute the collision meshes of your URDF with primitive shapes. Use* `<box>` *geometries to approximate the bounding box of the links.* ***Hint:*** *Enable collision visualization in* ***rviz*** *(go to the lateral bar →* `RobotModel` *→* `Collision Enabled`*) to adjust the collision meshes size to match (approximately) the bounding box of the visual meshes.*

To improve physical simulation and collision detection, the robot's **URDF** file was modified by substituting its detailed mesh geometries with simplified primitive shapes (`<box>`). This step allows the simulation engine to compute collisions more efficiently.

Example modification in the **arm.urdf** file:

arm.urdf

```
<link name="base_link">
    <visual>
      <geometry>
        <mesh filename="package://armando_description/meshes/
            base_link.stl" scale="0.001␣0.001␣0.001"/>
      </geometry>
      <origin rpy="0␣0␣0" xyz="0␣0␣0"/>
      <material name="red"/>
    </visual>
    <collision>
      <geometry>
        <box size="0.09␣0.09␣0.09"/>
      </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>
```

By enabling **Collision Visualization** in **RViz** (via **Displays** → **RobotModel** → **Collision Enabled**), the simplified shapes can be compared and adjusted to roughly match the visual meshes of the robot. This ensures that the simulation is both computationally efficient and physically accurate.

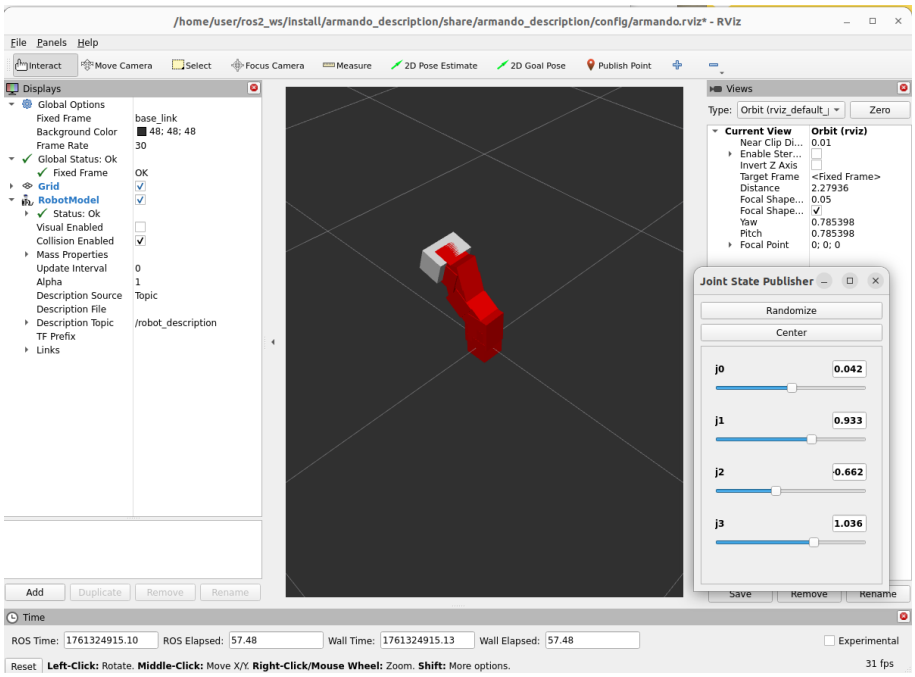


Figura 4: Armando represented by box shapes

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

   After successfully visualizing the robot in **RViz2**, the next objective was to bring the same model into **Gazebo**, adding controllers and interfaces to simulate its kinematic behavior. This required creating a new **package** for the **Gazebo** environment, defining a proper **launch file**, and integrating the hardware interface through the **ros2_control** framework.

   (a) *Request: Create a package named* `armando_gazebo` *using the ROS 2 CLI. Within this package, create a* `launch` *folder containing an* `armando_world.launch` *file and fill it with commands that load the URDF into the* `/robot_description` *topic and spawn your robot using the* `create` *node in the* `ros_gz_sim` *package.* ***Hint:*** *Follow the* `gazebo.launch.py` *file from the* `ros2_urdf_tutorial` *package as a reference. Launch the* `armando_world.launch` *file to start the simulation of your robot in Gazebo.*

   A new package named **armando_gazebo** was created using the **ROS 2 command line interface**. This package contains the files necessary to start **Gazebo**, load the robot's description, and spawn it into the simulation world.

```
$ cd ~/ros2_ws/src
$ ros2 pkg create --build-type ament_cmake --license Apache-2.0
    armando_gazebo
$ cd armando_gazebo
$ mkdir launch
$ cd launch
$ touch armando_world.launch.py
```

   The launch file **armando_world.launch.py** was implemented to load the robot's **URDF**, publish it on the **/robot_description** topic, and spawn the robot into **Gazebo** using the **ros_gz_sim** package. This file also includes the standard **Gazebo simulator launch file** (**gz_sim.launch.py**) and defines the robot's initial position within the world.

armando_world.launch.py

```
def generate_launch_description():

    urdf_file_name = 'arm.urdf'

    urdf = os.path.join(
        get_package_share_directory('armando_description'), '
            urdf',
        urdf_file_name
    )

    with open(urdf, 'r') as infp:
        robot_desc = infp.read()

    params = {"robot_description": robot_desc}

    # Define and configure the robot_state_publisher node
    robot_state_publisher_node = Node(
        package="robot_state_publisher",
        executable="robot_state_publisher",
        output="both",
        parameters=[params,
                    {"use_sim_time": True},
            ],
    )
```

```
    #Declaration of arguments for Gazebo (gz sim)
    declared_arguments = [
        DeclareLaunchArgument(
            'gz_args',
            default_value='-r -v 1 empty.sdf',
            description='Arguments for gz_sim',
        )
    ]

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

    # Set the initial spawn position of the robot in Gazebo
    position = [0.0, 0.0, 0.060]

    spawn_rob = Node(
        package='ros_gz_sim',
        executable='create',
        output='screen',
        arguments=[
            '-topic', 'robot_description',
            '-name', 'armando',
            '-allow_renaming', 'true',
            '-x', str(position[0]),
            '-y', str(position[1]),
            '-z', str(position[2]),
        ],
    )

    return LaunchDescription(
        declared_arguments
        + [ gz_ign,
            robot_state_publisher_node,
            spawn_rob,
        ]
    )
```

After completing the **launch file**, the **CMakeLists.txt** and **package.xml** were updated to include all necessary **dependencies**.

Before launching the simulation, we need to modify the package.xml file to ensure that the robot can be visualized in Gazebo.

package.xml

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

We also need to synchronize the clocks of Rviz and Gazebo by launching the following

node:

armando_world.launch.py

```
clock_bridge = Node(
    package="ros_ign_bridge",
    executable="parameter_bridge",
    arguments=['/clock@rosgraph_msgs/msg/Clock[ignition.msgs
        .Clock'],
    output="screen",
)
```

To prove this we have to launch the following command in the terminal:

```
$ ros2 topic echo /clock
```

Once the **workspace** was rebuilt and sourced, the **Gazebo world** could be launched with:

```
$ cd ~/ros2_ws
$ colcon build --packages-select armando_description
    armando_gazebo
$ source install/setup.bash
$ ros2 launch armando_gazebo armando_world.launch.py
```

The result was the successful spawning of the robot **"Armando"** in the **Gazebo environment**, positioned at the specified coordinates and ready for **control**.

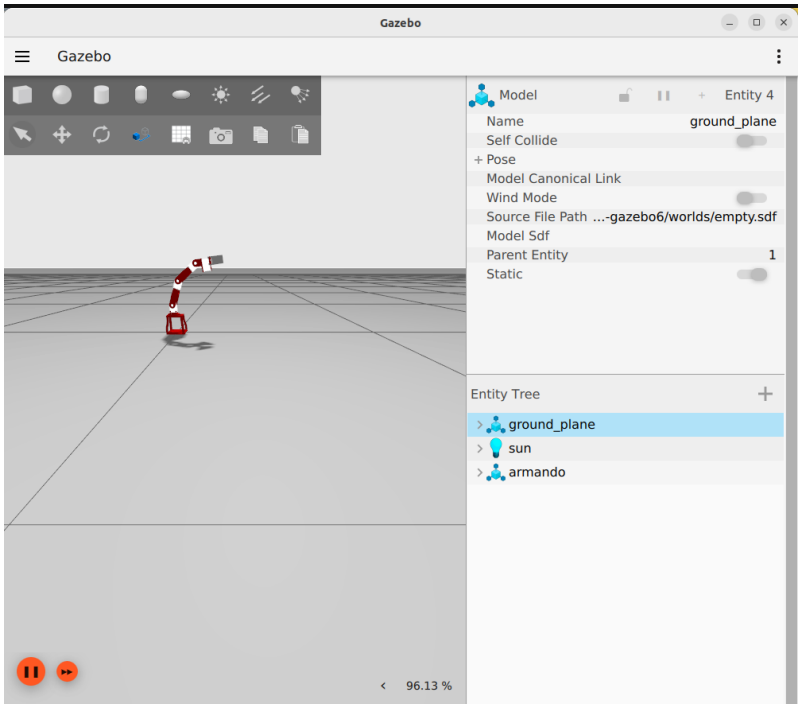


Figura 5: Armando in Gazebo

We can notice from the simulation that Gazebo is a simulation environment that differs from RViz because it considers what really happens in the simulated world. Indeed, we see that the robot is affected by gravity and bends since no joint controls are active.

(b) *Request: Add a* `PositionJointInterface` *as a hardware interface to your robot using the* `ros2_control` *framework. Create an* `armando_hardware_interface.xacro` *file in the* `armando_description/urdf` *folder, containing a macro that defines the hardware interface for the joints of your robot, and include it in your main* `armando.urdf.xacro` *file using* `xacro:include`.
Hint: *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 as shown in here.*

The first step is to convert the URDF file into a URDF.xacro file.

```
$ cd src
$ cd armando_description
$ cd urdf
$ cp arm.urdf arm.urdf.xacro
$ touch armando_hardware_interface.xacro
```

After that, a PositionJointInterface can be added as the robot's hardware interface by creating a dedicated **armando_hardware_interface.xacro** file.

armando_hardware_interface.xacro

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

<!-- Declare this file as a Xacro file -->
<robot xmlns:xacro="http://ros.org/wiki/xacro">

    <xacro:macro name="position_joint_interface" params="
        joint_name␣initial_pos">

        <!-- Define the joint block used -->
        <joint name="${joint_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>
```

This **macro** is then included in the main **arm.urdf.xacro** file and instantiated for each **joint** of the manipulator:

arm.urdf.xacro

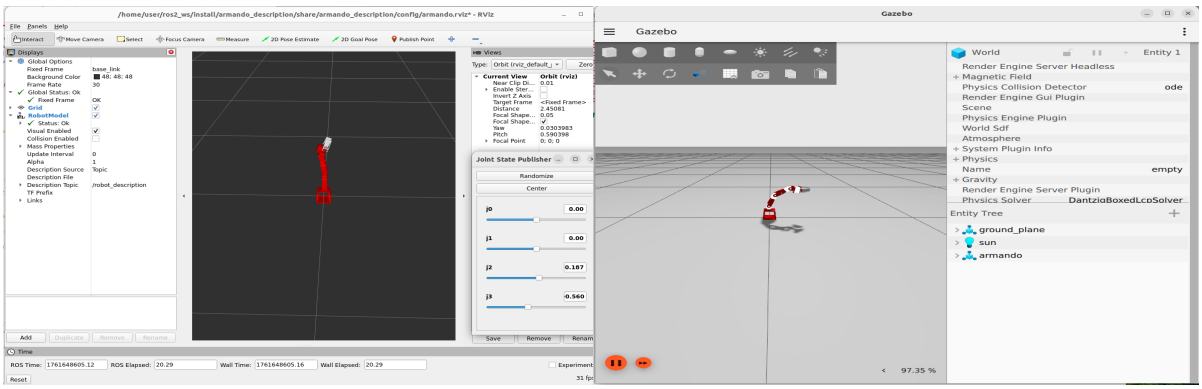
```
<!-- Hardware Interface macro include -->
<xacro:include filename="$(find␣armando_description)/urdf/
    armando_hardware_interface.xacro"/>

<ros2_control name="ArmandoSystem" type="system">
  <!-- Declare joint interfaces -->
  <xacro:position_joint_interface joint_name="j0"
      initial_pos="2.0"/>
  <xacro:position_joint_interface joint_name="j1"
      initial_pos="0.0"/>
  <xacro:position_joint_interface joint_name="j2"
      initial_pos="1.0"/>
  <xacro:position_joint_interface joint_name="j3"
      initial_pos="1.0"/>
</ros2_control>
```

When working with Xacro files, the robot model is loaded from the `.xacro` file instead of directly from the `.urdf`. Therefore, the important step is to modify the launch file to include the Xacro processing routine.

armando_world.launch.py

```
#XACRO version
    # Build the complete path to the xacro file
    xacro_file_name = "arm.urdf.xacro"
    xacro = os.path.join(
        get_package_share_directory('armando_description'), "
            urdf", xacro_file_name)

    robot_description_xacro = {
        "robot_description": ParameterValue(
            Command(['xacro␣', xacro]),
            value_type=str
        )
    }
```

In the following images we can see that the robot is correctly visualized both in rzviz and in gazebo:



(c) *Request: Add inside* ***armando.urdf.xacro*** *the commands to enable the Gazebo ROS 2 control plugin and load joint position controllers from a* ***.yaml*** *file. Then, spawn the joint state broadcaster and the position controllers using the* ***controller_manager*** *package from* ***armando_world.launch****. Launch the Gazebo robot simulation and demonstrate how the hardware interface is correctly loaded and connected.*

Hint: *use the* ***RegisterEventHandler*** *function to load controllers* after *Gazebo is started.*

Inside the `armando.urdf.xacro` file, a Gazebo plugin named `ign_ros2_control-system` was introduced. This plugin establishes the connection between Gazebo and `ROS 2 Control`, allowing the simulated robot to respond to control commands. The plugin loads the controller configuration from the `armando_controller.yaml` file, which defines both the `JointStateBroadcaster`—used to publish the robot joint states—and the `JointGroupPositionController`, responsible for sending position commands to the actuated joints.

arm.urdf.xacro

```
<!-- Controller Configuration Plugin -->
<gazebo>
  <plugin filename="ign_ros2_control-system" name="
      ign_ros2_control::IgnitionROS2ControlPlugin">
  <parameters>$(find armando_description)/config/
      armando_control.yaml</parameters>
    <controller_manager_prefix_node_name>controller_manager
      </controller_manager_prefix_node_name>
  </plugin>
</gazebo>
```

YAML FILE:

armando_description/config/armando_control.yaml

```
controller_manager:
  ros__parameters:
    update_rate: 100  # Hz

    joint_state_broadcaster:
      type: joint_state_broadcaster/JointStateBroadcaster

    position_controller:
      type: position_controllers/JointGroupPositionController

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

[**Note:** From the command line, a new folder named `armando_controller` was created. This folder contains both the launch file that defines the nodes for position control and the elements required for point 4, as will be explained later, in order to implement runtime control switching.]

Once the robot description and control interfaces were properly set up, the controllers were launched from the `armando_control.launch.py` file. This was achieved by spawning the controller nodes from the `controller_manager` package. Two nodes were created: one to start the `joint_state_broadcaster` and another to initialize the position controller. To ensure that the controllers were activated only after the robot model had been successfully spawned in Gazebo, a `RegisterEventHandler` function was used. This mechanism delays the controller activation until the spawning process completes, preventing initialization conflicts and guaranteeing a correct connection between Gazebo and the hardware interface.

armando_control.launch.py

```
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"],
    )

    #Launch the ros2 controllers after the model spawns in
        Gazebo
    delay_joint_traj_controller = RegisterEventHandler(
        event_handler=OnProcessExit(
            target_action=spawn_rob,
            on_exit=[position_controller],
        )
    )

    delay_joint_state_broadcaster = (
        RegisterEventHandler(
            event_handler=OnProcessExit(
                target_action=spawn_rob,
                on_exit=[joint_state_broadcaster],
            )
        )
    )
```

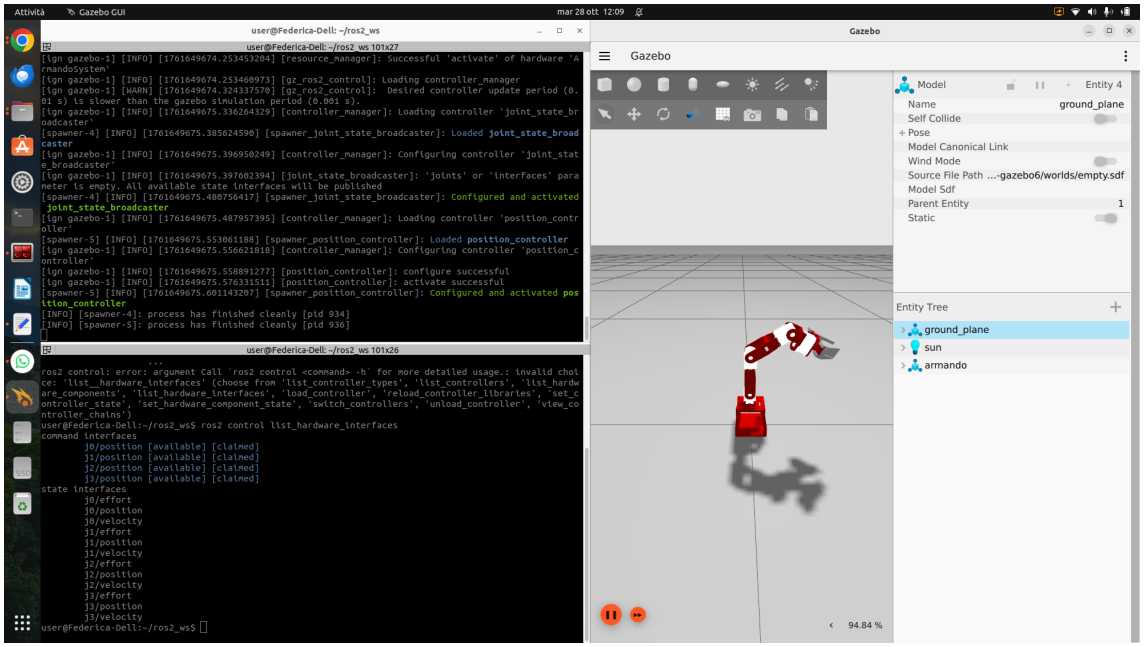


Figura 6: PositionJointInterface

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

(a) *Request: Go into your* ***armando.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 at the base of your robot.*

To extend the robot model with a vision component, a new link representing the camera was added to the `armando_urdf.xacro` file. The new element, named `camera_link`, was modeled as a small box with predefined dimensions and a white material to make it clearly visible in the simulation environment. This link does not include any collision or inertial properties, as its purpose is purely visual. The camera link was then attached to the robot base by introducing a fixed joint, `camera_joint`, with the `base_link` defined as the parent and the `camera_link` as the child.

Through the `<origin>` tag, both the position and orientation of the camera were set relative to the base of the robot. By defining this fixed connection, the camera remains rigidly attached to the base of the manipulator, allowing it to maintain a stable viewpoint as the robot moves.

arm.urdf.xacro

```
<link name="camera_link">
    <visual>
        <geometry>
            <box size="0.005 0.003 0.003"/>
        </geometry>
        <material name="white"/>
    </visual>
</link>

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

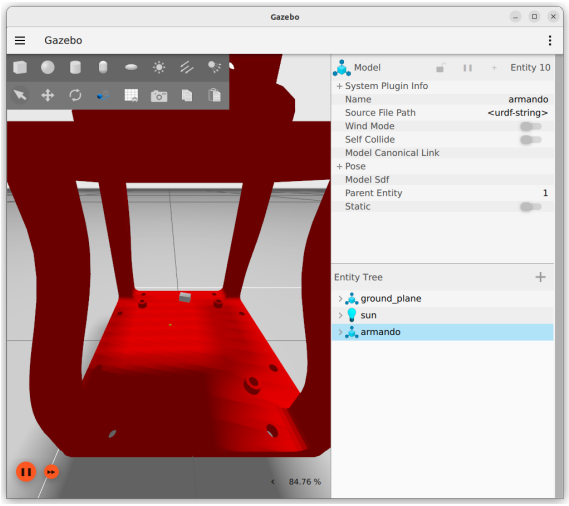


Figura 7: Camera at the base frame

(b) *Request: Create an* `armando_camera.xacro` *file in the* `armando_gazebo/urdf` *folder, add the sensor specifications within a* `xacro:macro` *and the* `gz-sim-sensors-system` *plugin. Import it into* `armando.urdf.xacro` *using the* `xacro:include` *command.*

After defining the physical link and joint of the camera in the previous step, a new file named `armando_camera.xacro` was created within the `armando_gazebo/urdf` folder to describe the characteristics of the camera sensor used in simulation. This file defines a `xacro:macro` called `armando_camera`, which receives the link name as an argument and embeds the camera specifications inside a Gazebo sensor plugin. The plugin used, `gz-sim-sensors-system`, enables the simulation of vision sensors and was configured with the rendering engine `ogre2`. The camera sensor continuously publishes images to the `/camera` topic, making them accessible to other ROS 2 nodes.

armando_camera.xacro

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

<xacro:macro name="armando_camera" params= "link">

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

  <gazebo reference="${link}">
  <sensor name="camera" type="camera">
    <camera>
      <horizontal_fov>1.047</horizontal_fov>
      <image>
        <width>640</width>
        <height>480</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>
</xacro:macro>

</robot>
```

Once the macro was defined, it was imported into the main `armando.urdf.xacro` file through the `xacro:include` command. By calling the macro and passing the `camera_link` as the reference link, the camera sensor becomes attached to the robot model and is automatically loaded when the simulation starts.

arm.urdf.xacro

```
<xacro:include filename="$(find␣armando_gazebo)/urdf/
    armando_camera.xacro"/>
<xacro:armando_camera link="camera_link"/>
```

(c) *Request: Launch the Gazebo simulation using* ***armando_gazebo.launch****, and check if the image topic is correctly published using* ***rqt_image_view****. Hint: remember to add the correct* ***ros_ign_bridge*** *commands into the launch file.*

To enable communication between the simulated camera in Gazebo and ROS 2, a bridge was created using the `ros_ign_bridge` package. This bridge converts the sensor data published in Ignition message format into standard ROS 2 messages that can be accessed by other nodes in the system. Through this configuration, the `/camera` topic was bridged from `ignition.msgs.Image` to `sensor_msgs/msg/Image`, and the `/camera_info` topic from `ignition.msgs.CameraInfo` to `sensor_msgs/msg/CameraInfo`. An additional argument was used to remap the camera topic to `/videocamera` for consistency with the rest of the ROS 2 environment.

armando_world.launch.py

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

To visualize this sensor we simulated the robot in gazebo, put a shape in front of the camera and used the command:

```
$ ros2 run rqt_image_view rqt_image_view
```

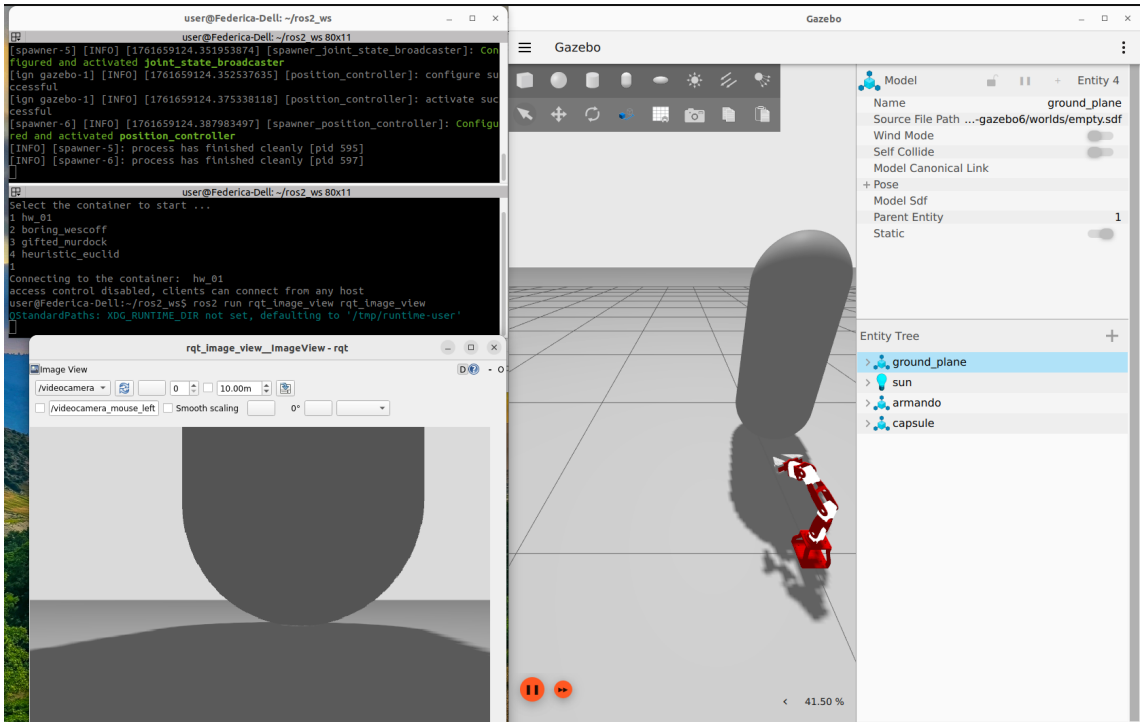


Figura 8: Camera visualization

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

   (a) *Request: Create the* `armando_controller` *package with a ROS C++ node named* `arm_controller_node`*. The dependencies are* `rclcpp`*,* `sensor_msgs` *and* `std_msgs`*. Modify opportunely the* `CMakeLists.txt` *and the* `package.xml` *files to compile your node. Hint: adjust the* `add_executable` *and* `ament_target_dependencies` *commands.*

   A new ROS 2 package named `armando_controller` was created using the `ament_cmake` build system. This package contains a C++ node, `arm_controller_node`, which will later be used to implement the control logic for the manipulator.

```
$ cd src
$ $ros2 pkg create --build-type ament_cmake
    armando_controller
$ cd armando_controller/src/
$ touch arm_controller_node.cpp
```

   The node requires three main dependencies: `rclcpp`, `sensor_msgs`, and `std_msgs`. To correctly build and link the node, both the `CMakeLists.txt` and the `package.xml` files were updated to include the corresponding dependencies.

CMakeLists.txt

```
# find dependencies
find_package(ament_cmake REQUIRED)
find_package(rclcpp REQUIRED)
find_package(sensor_msgs REQUIRED)
find_package(std_msgs REQUIRED)

install(TARGETS
  arm_controller_node
  DESTINATION lib/${PROJECT_NAME}
)
```

armando_world.launch.py

```
  <build_depend>std_msgs</build_depend>
  <build_depend>sensor_msgs</build_depend>
  <build_depend>rlcpp</build_depend>

  <exec_depend>sensor_msgs</exec_depend>
  <exec_depend>std_msgs</exec_depend>
  <exec_depend>rlcpp</exec_depend>
```

   (b) *Request: Within the node, create a subscriber to the topic* `joint_states` *and a callback function that prints the current joint positions of the robot. Note: the topic contains a* `sensor_msgs/JointState`*.*

   To monitor the joint positions of the robot, a C++ node named `arm_controller_node` was implemented. This node subscribes to the `/joint_states` topic, which is published by the `joint_state_broadcaster` and contains the current position, velocity, and effort of each joint. A custom class called `JointStateSubscriber` was defined by inheriting

from `rclcpp::Node`. Inside its constructor, the `create_subscription` method was used to subscribe to the topic and bind it to a callback function that handles the incoming messages.

The callback function, `topic_callback`, is automatically triggered whenever new data is received on the `/joint_states` topic. It extracts the array of joint positions from the incoming message and prints them to the terminal using the `RCLCPP_INFO` macro. Since this node only needs to react when new messages are published, no timer was required. The main function simply initializes the ROS 2 node, spins it to keep it active, and shuts it down gracefully when execution ends.

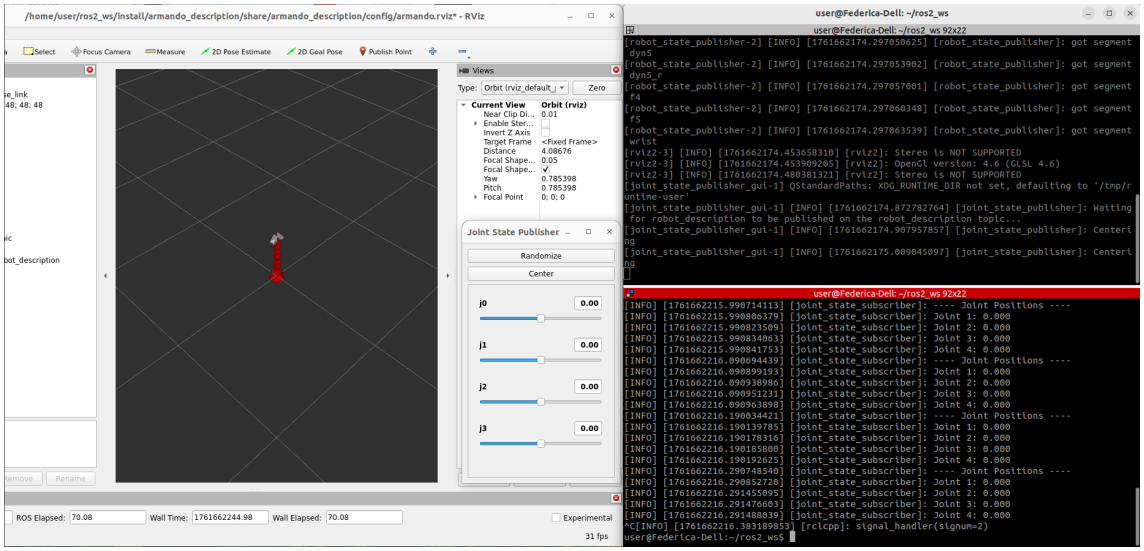
Through this subscriber node, the current joint positions of the manipulator can be monitored directly in the console, confirming the correct connection between the hardware interface and the control system.

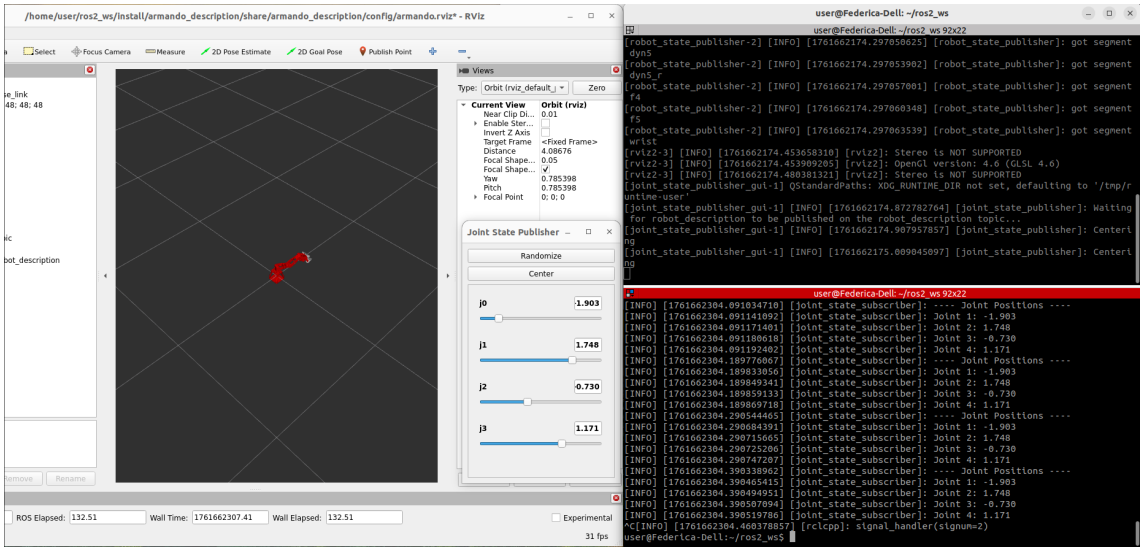
arm_controller_node.cpp

```
  class JointStateSubscriber : public rclcpp::Node
{
public:
  JointStateSubscriber()
  : Node("joint_state_subscriber")
  {
    subscription_ = this->create_subscription<sensor_msgs::
        msg::JointState>(
      "joint_states", rclcpp::SensorDataQoS(),
      std::bind(&JointStateSubscriber::topic_callback, this,
          _1));
  }

private:
  void topic_callback(const sensor_msgs::msg::JointState &
      msg) const
  {
    RCLCPP_INFO(this->get_logger(), "-----␣Joint␣Positions␣
        -----");
    for (size_t i = 0; i < msg.position.size(); i++)
      RCLCPP_INFO(this->get_logger(), "Joint␣%zu:␣%.3f", i +
          1, msg.position[i]);
  }

  rclcpp::Subscription<sensor_msgs::msg::JointState>::
      SharedPtr subscription_;
};

...

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

  return 0;
}
```

To prove this Subscriber we started the simulation. then we opened another terminal and put the following commands:

```
$ ros2 run armando_controller arm_controller_node
```





(c) *Request: Create a publisher that writes sequentially at least four different position commands onto the* `/position_controller/command` *topics. Note: the command is a* `std_msgs/msg/Float64MultiArray`*.*

To drive the manipulator with position targets, a ROS 2 C++ node named `PositionControllerPublisher` was implemented. The node encapsulates a small command scheduler: it stores a sequence of joint configurations (each configuration is a vector of four doubles, one per joint) and exposes a public parameter for the sequence, allowing the targets to be replaced without recompilation. A periodic timer triggers the publication every 3 seconds. At each tick, the node selects the next configuration in the sequence, logs it for traceability, and publishes a `Float64MultiArray` message on the `/position_controllers/command` topic. This topic is the input of the `JointGroupPositionController` configured earlier, so the array order must match the

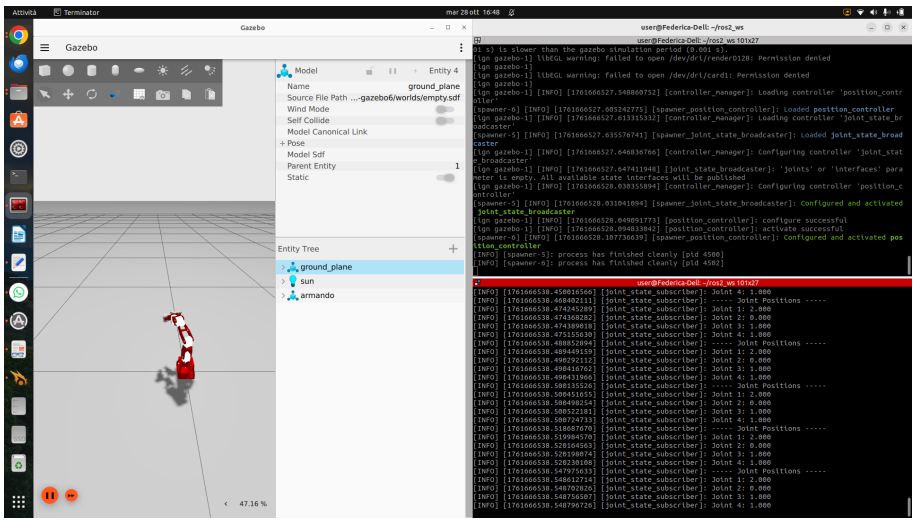
joint order defined for that controller. The timer advances the index in a circular fashion, yielding a deterministic and repeatable motion pattern.
Since the feedback from the robot had to be monitored while sending commands, two nodes were executed within the same process: the previously defined joint-state subscriber and the new position-command publisher. A multi-threaded executor was used to host both nodes simultaneously, allowing the subscriber to process incoming `/joint_states` messages while the publisher maintained its three-second update rate. This structure prevents race conditions and ensures that both the commanded targets and the measured positions are logged coherently, confirming the correct connection between the hardware interface, the controller manager, and the control pipeline.
Finally, with the Gazebo simulation running, the console output showed each scheduled command alongside the corresponding joint positions, demonstrating that the controller properly received the commands and the simulated manipulator followed the desired motion trajectory.

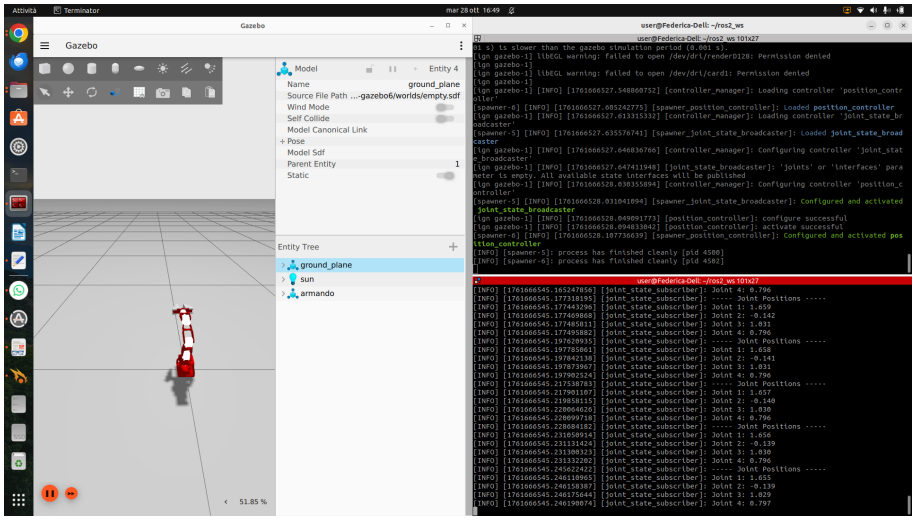
arm_controller_node.cpp

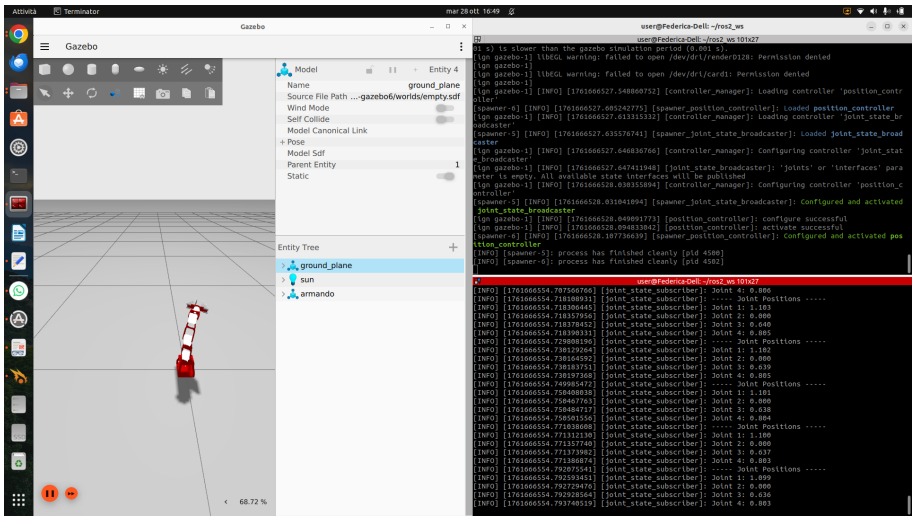
```
class PositionControllerPublisher : public rclcpp::Node
{
public:
  PositionControllerPublisher()
  : Node("position_controller_publisher"), count_(0)
  {
    // Sequenza target dei giunti
    target_pos_ = {
  { 0.0,  0.0,  0.0,  0.0},
  { 1.2, -1.0,  1.2,  0.6},
  {-1.2,  1.0, -1.2, -0.6},
  { 0.6,  0.0, -0.6,  1.0},
  { 0.0,  0.0,  0.0,  0.0}
};

    publisher_ = this->create_publisher<std_msgs::msg::
        Float64MultiArray>("/position_controller/commands",
        10);

    // Timer ogni 3 secondi
    timer_ = this->create_wall_timer(
      3000ms, std::bind(&PositionControllerPublisher::
          timer_callback, this));
  }

private:
  void timer_callback()
  {
    // Prende la configurazione corrente
    const auto & current_positions = target_pos_[count_];

    std_msgs::msg::Float64MultiArray commands;
    commands.data = current_positions;

    // Stampa di debug
    std::string positions_str = "[";
    for (size_t i = 0; i < current_positions.size(); ++i) {
      positions_str += std::to_string(current_positions[i]);
      if (i + 1 < current_positions.size())
        positions_str += ",␣";
    }
```

```
    positions_str += "]";

    RCLCPP_INFO(this->get_logger(),
                "Publishing sequence step %zu: %s",
                count_, positions_str.c_str());

    // Pubblica il comando
    publisher_->publish(commands);

    // Passa al prossimo step ciclicamente
    count_ = (count_ + 1) % target_pos_.size();
  }

  rclcpp::TimerBase::SharedPtr timer_;
  rclcpp::Publisher<std_msgs::msg::Float64MultiArray>::
      SharedPtr publisher_;
  std::vector<std::vector<double>> target_pos_;
  size_t count_;
};

int main(int argc, char * argv[])
{
  rclcpp::init(argc, argv);

  auto joint_state_subscriber = std::make_shared<
      JointStateSubscriber>();
  auto position_controller_publisher = std::make_shared<
      PositionControllerPublisher>();

  rclcpp::executors::MultiThreadedExecutor executor;
  executor.add_node(joint_state_subscriber);
  executor.add_node(position_controller_publisher);
  executor.spin();

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

Simulations:







(d) *Request: Create a joint trajectory publisher that sends the same position commands and make it work by loading the corresponding controllers. Allow the user to select which publisher/controllers to use (position or trajectory) by adding ROS arguments to your node and to your launch files.*

To extend the control pipeline beyond single set-points, a `JointTrajectoryController` was added to the `armando_controller.yaml`. The controller is configured to command the `position` interface, expose both `position` and `velocity` as state interfaces, and operate on the four arm joints (`j0--j3`).

armando_controller.yaml

```
trajectory_controller:
      type: joint_trajectory_controller/
          JointTrajectoryController

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
```

The `arm_controller_node.cpp` file was then modified to enable the user to choose the desired control mode at launch. A launch argument (`controller_type`) allows the selection between the position and trajectory controller. Depending on the value specified (`controller_type:=position` or `controller_type:=trajectory`), only the corresponding spawner node is started by the controller manager. If no argument is provided, the system defaults to the position controller.

armando_control.launch.py

```
trajectory_controller = Node(
        package="controller_manager",
        executable="spawner",
        arguments=["trajectory_controller", "--controller-
            manager", "/controller_manager"],
        condition=IfCondition(PythonExpression(["'",
            controller_type, "' == 'trajectory'"]))
    )
```

Finally, the main application node was refactored so that it can automatically publish to the correct interface without recompilation. The node declares a boolean parameter, `use_trajectory`, which determines the active mode. When set to true, it publishes `trajectory_msgs/JointTrajectory` messages, including joint names and timestamps, to the trajectory controller topic. Otherwise, it sends `std_msgs/Float64MultiArray` commands to the position controller. A periodic timer triggers the publication of a sequence of predefined joint targets.

arm_controller_node.cpp

```
class PosControllerPublisher : public rclcpp::Node
{
public:
  PosControllerPublisher()
  : Node("controller_publisher"),
    step_(0)
```

```
  {
    // Declare and get parameter to choose controller type
    this->declare_parameter<bool>("use_trajectory", false);
    use_traj_ = this->get_parameter("use_trajectory").as_bool
        ();

    target_pos_ = {
      { 0.0,  0.0,  0.0,  0.0},
      { 1.0,  1.0, -1.0,  0.8},
      {-1.0,  1.0, -1.0, -0.8},
      {-1.0, -1.0,  1.0, -0.8},
      { 1.0, -1.0,  1.0,  0.8},
      { 0.0,  0.0,  0.0,  0.0}
    };
    // Initialize publisher based on controller type choosing
        the appropriate topic
    if (use_traj_)
    {
      publisher_traj_ = this->create_publisher<
          trajectory_msgs::msg::JointTrajectory>(
        "/trajectory_controller/joint_trajectory", 10);
      RCLCPP_INFO(this->get_logger(), "Using␣Trajectory␣
          Controller");
    }
    else
    {
      publisher_pos_ = this->create_publisher<std_msgs::msg::
          Float64MultiArray>(
        "/position_controller/commands", 10);
      RCLCPP_INFO(this->get_logger(), "Using␣Position␣
          Controller");
    }
    // Create a timer to publish commands every 5 seconds
    timer_ = this->create_wall_timer(
      5000ms, std::bind(&PosControllerPublisher::
          timer_callback, this)
    );
  }
private:
// Timer callback to be executed periodically
  void timer_callback()
  {
    const auto &pos = target_pos_[step_];

    if (use_traj_)
    {
      trajectory_msgs::msg::JointTrajectory traj;
      traj.joint_names = {"j0", "j1", "j2", "j3"};
      traj.points.resize(1);
      traj.points[0].positions = pos;
      traj.points[0].time_from_start.sec = 5;
      traj.points[0].time_from_start.nanosec = 0;
      publisher_traj_->publish(traj);
      RCLCPP_INFO(this->get_logger(),
                  "Sent␣Trajectory␣step␣%zu", step_);
    }
    else
    {
      std_msgs::msg::Float64MultiArray cmd;
```

```
        cmd.data = pos;
        publisher_pos_->publish(cmd);
        RCLCPP_INFO(this->get_logger(),
                    "Sent Position step %zu", step_);
      }

      step_ = (step_ + 1) % target_pos_.size();
    }

    bool use_traj_;
    size_t step_;
    std::vector<std::vector<double>> target_pos_;

    rclcpp::TimerBase::SharedPtr timer_;
    rclcpp::Publisher<std_msgs::msg::Float64MultiArray>::
        SharedPtr publisher_pos_;
    rclcpp::Publisher<trajectory_msgs::msg::JointTrajectory>::
        SharedPtr publisher_traj_;
};
```

At runtime, the control mode can be selected via:

```
$ ros2 run armando_controller arm_controller_node --ros-
    args -p use_trajectory:=<value>
```