Robotics Lab - MSc in Automation Engineering and Robotics

Homework 01

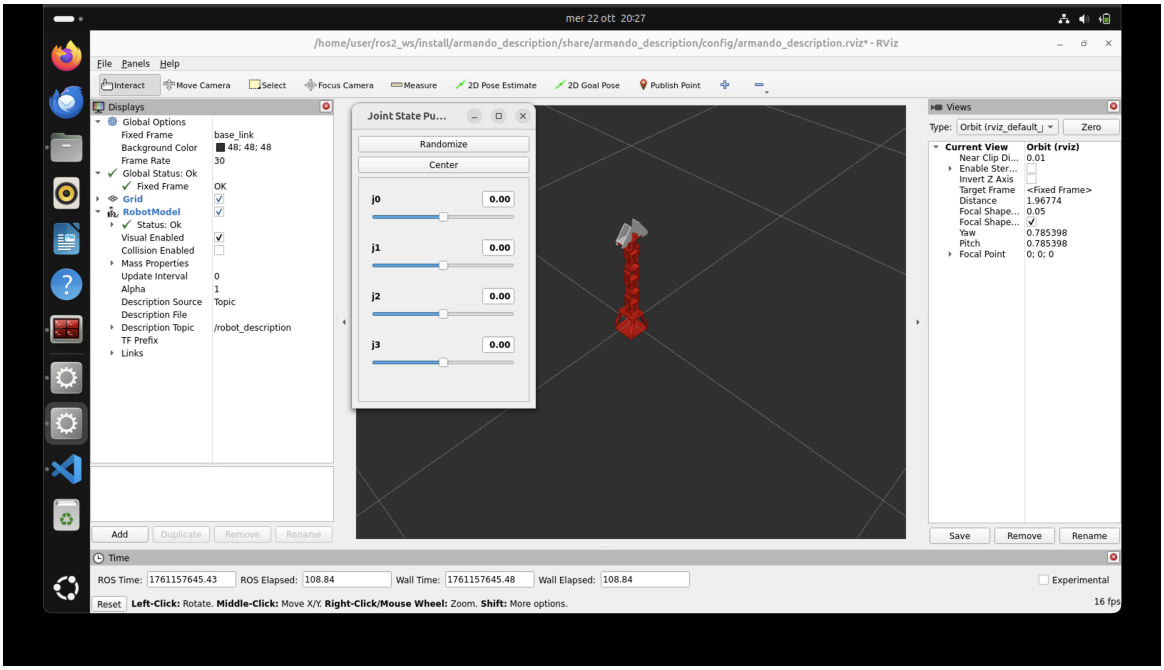
Antonio Polito*P38000330 Maria Rosaria Imperato†P38000351

Gianmarco Ferrara‡P38000343

29 October 2025

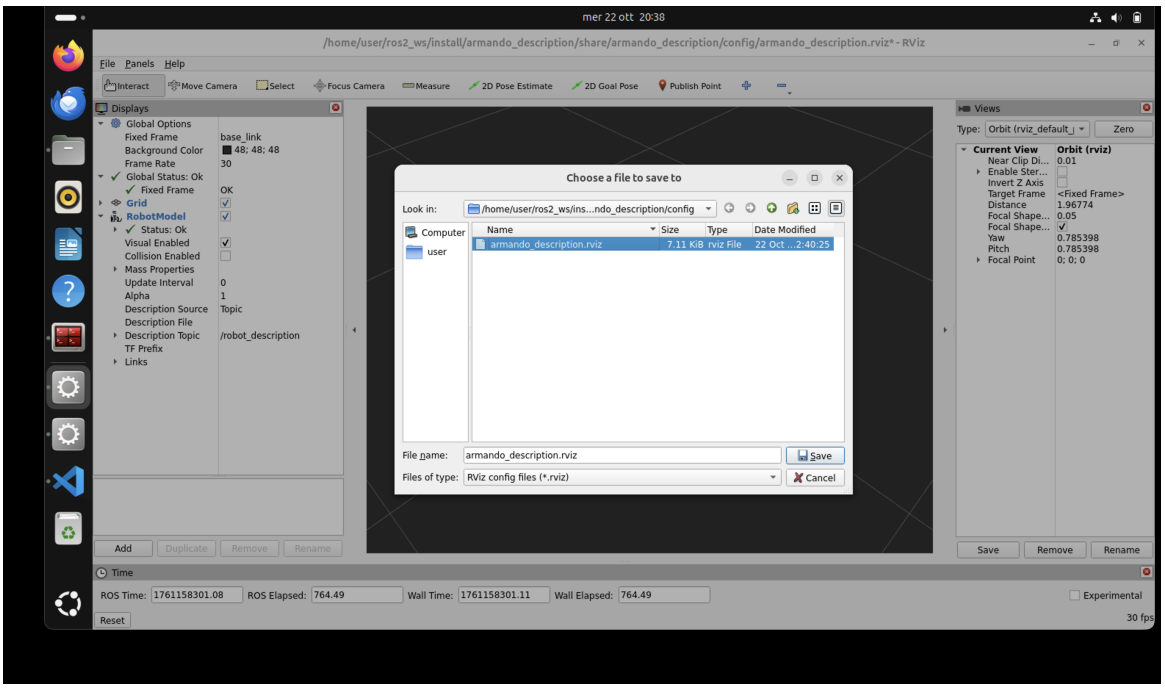
1 Modify the URDF description of your robot and visualize it in Rviz

(a) We created a launch folder within the `armando_description` package containing a launch file named `armando_display.launch.py`. This launch file loads the URDF as a `robot_description` ROS param, starts the `robot_state_publisher` node, the `joint_state_publisher_gui` node, and the `rviz2` node. Furthermore, we added run time dependances into the `package.xml` and installed the necessary folders into the `CMakeLists.txt` file.
To visualize the robot in `rviz` we changed the Fixed Frame in the lateral bar and add the RobotModel display.



(b) We created a config folder and saved a `.rviz` configuration file called `armando_description.rviz` that automatically loads the RobotModel display.

*https://github.com/P0l1702/
†https://github.com/MariaRosaria1
‡https://github.com/gianmarco-ferrara



Then we passed this configuration file as an argument to the node into `armando_display.launch` file.

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

After specifying the `rviz_config_file` in `declared_arguments` list:

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

Clearly, before building we specified the directory into the `CMakeLists.txt`.

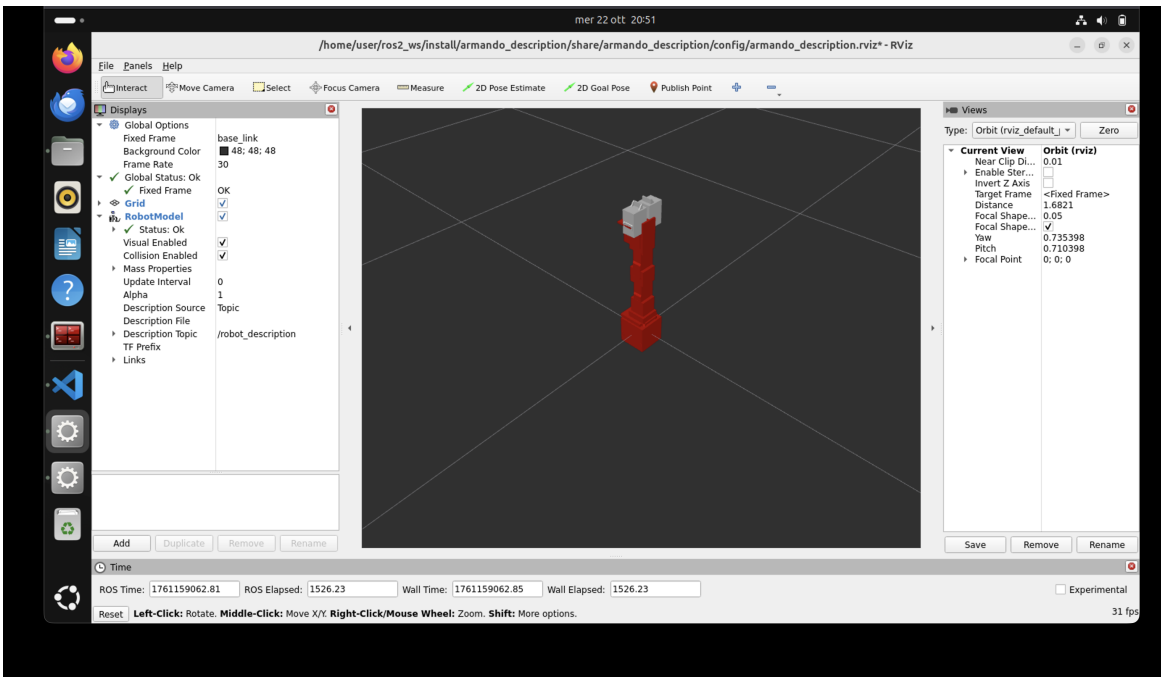
```
install(
DIRECTORY config
DESTINATION share${PROJECT_NAME})
```

(c) We substituted the collision meshes of our URDF with primitive shapes. As required, we used `<box>` geometries to approximate the bounding box of the links.

For each collision we created a box of proper dimensions substituting the collision meshes by using the following code (with different numerical values):

```
<collision>
  <geometry>
    <box size="0.09 0.09 0.09"/>
  </geometry>
    <origin rpy="0 0 0" xyz="0 0 0"/>
</collision>
```

In rviz we enabled collision visualization to adjust the collision meshes size to match to the bounding box of the visual meshes.



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

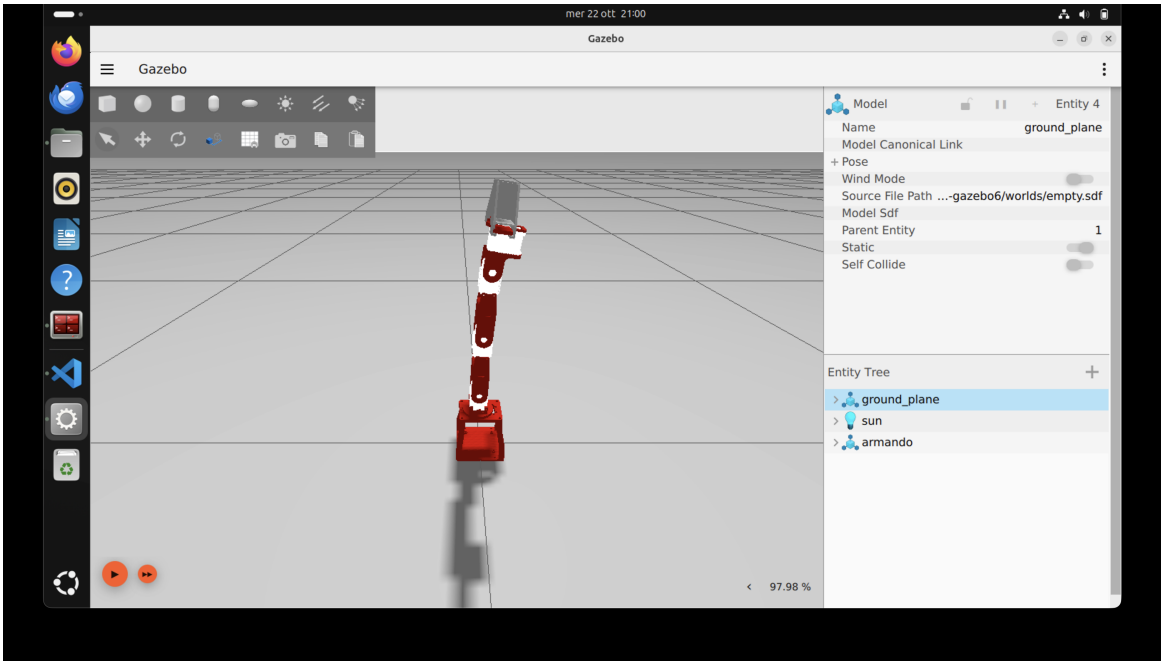
(a) We created a package named `armando_gazebo` using ros2 CLI (Command Line Interface):

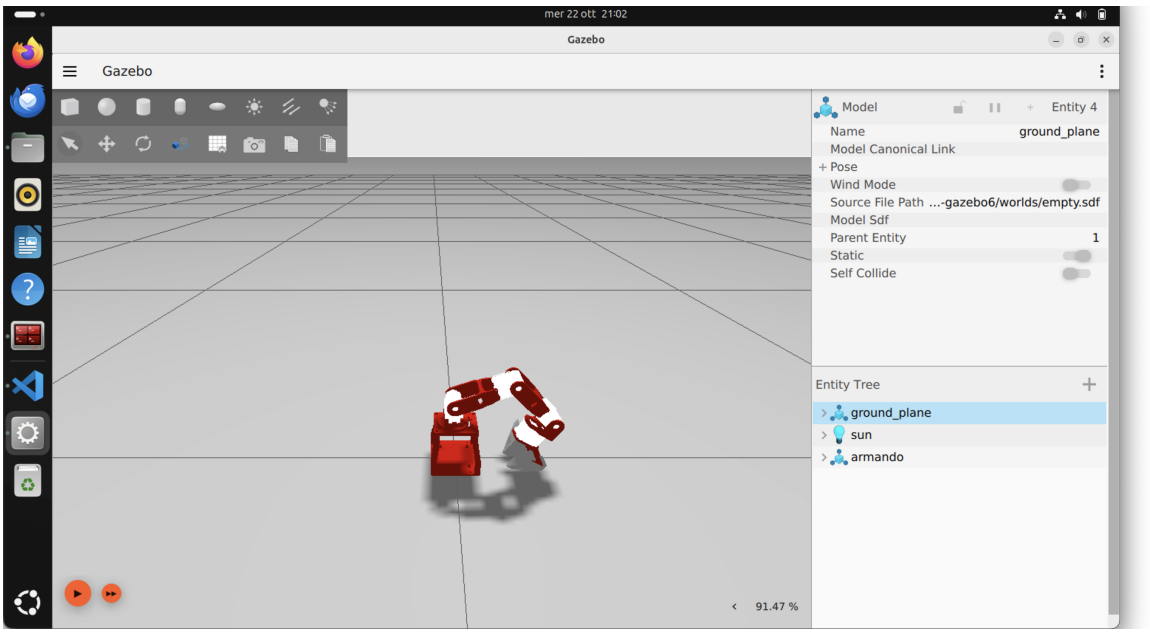
```
$ ros2 pkg create --build-type ament_cmake armando_gazebo
```

Within this package we created a launch folder containing a launch file named `armando_world.launch.py`. Through the launch file we load the URDF into the `/robot_description` topic and spawn our robot using the created node in the `ros_gz_sim` package.

In particular, we had to modify the file `package.xml` in the `armando_description` package in order to make the model contained in it to be visulizable by gazebo. The export section will appear like that:

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





Through these two pictures can be hilighted the fact that gazebo is a dynamic simulator differently from rviz. Indeed due to the lack of actuation the robot falls down. Furthermore, it is important to remember that it is very important to keep ROS2 and Gazebo clock environments synchronized. This can be achieved by adding the clock bridge in the launch file:

```
    clock_bridge = Node(
     package="ros_ign_bridge",
     executable="parameter_bridge",
     arguments=['/clock@rosgraph_msgs/msg/Clock[ignition.msgs.Clock'],
     parameters=[{
         "qos_overrides./tf_static.publisher.durability": "transient_local"
     }],
     output="screen",
)
```

And through the execution of:

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

This confirms that the `/clock` topic is being published, indicating successful synchronization between Gazebo and ROS2. This ensures that the simulation time in Gazebo is properly shared with ROS2 nodes, preventing timing inconsistencies.

(b) We added a PositionJointInterface as a hardware interface to our robot by creating an `armando_hardware_interface.xacro` file in the `armando_description/urdf` folder containing a macro that defines the hardware interface for the joints:

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

  <xacro:macro name="PositionJointInterface" 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>
```

Then we copied (in order to not loose the `.urdf` file) the `.urdf` file into a `.urdf.xacro` substituting the initial `<robot>` tag into:

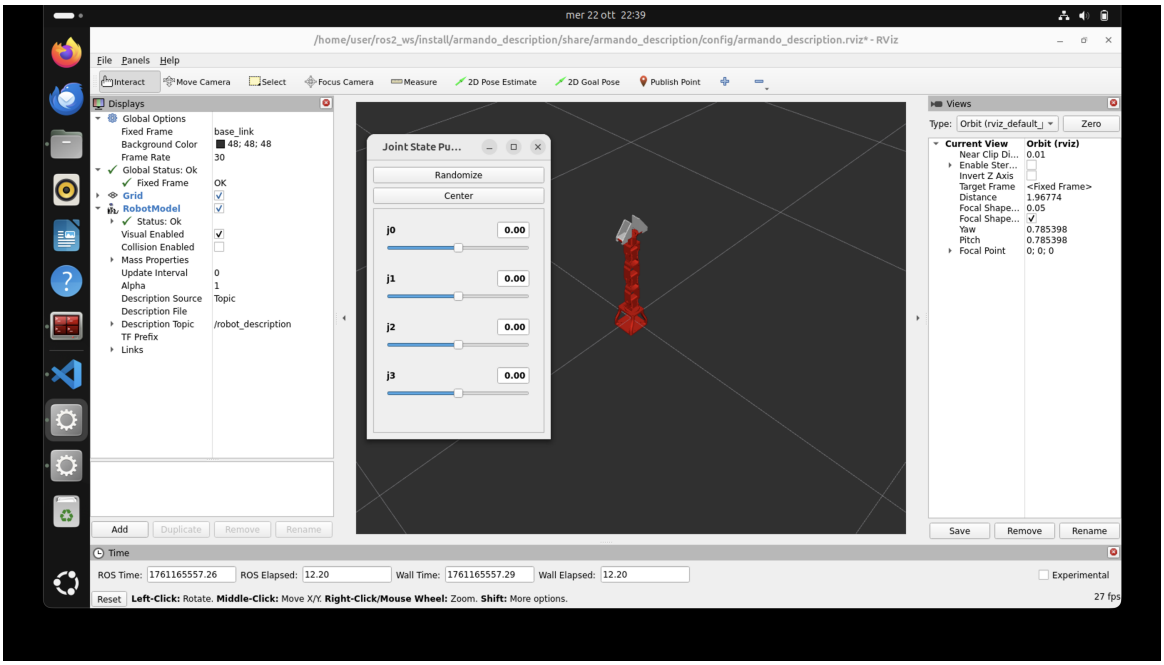
```
<robot xmlns:xacro="http://www.ros.org/wiki/xacro" name="arm">
```

We included the `armando_hardware_interface.xacro` file into our main `armando.urdf.xacro` file using `xacro:include`:

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

Then we loaded the URDF into the launch file of the `armando_description` package using the xacro routine:

```
xacro_file_name = 'arm.urdf.xacro'
xacro = os.path.join(get_package_share_directory('armando_description'), "urdf", xacro_file_name)

robot_description_links = {"robot_description": Command(['xacro ', xacro])}
```



(c) We added inside the `armando.urdf.xacro` the commands to enable the Gazebo ROS2 control plugin and load the joint position controllers from the file `armando_controller.yaml`.

```
<gazebo>
  <plugin filename="ign_ros2_control-system" name="ign_ros2_control::IgnitionROS2ControlPlugin">
    <parameters>$(find ros2_sensors_and_actuators)/config/armando_controller.yaml</parameters>
    <controller_manager_prefix_node_name>controller_manager</controller_manager_prefix_node_name>
  </plugin>
</gazebo>
```

In the `armando.urdf.xacro` file we also specified an arbitrary position for each joint:

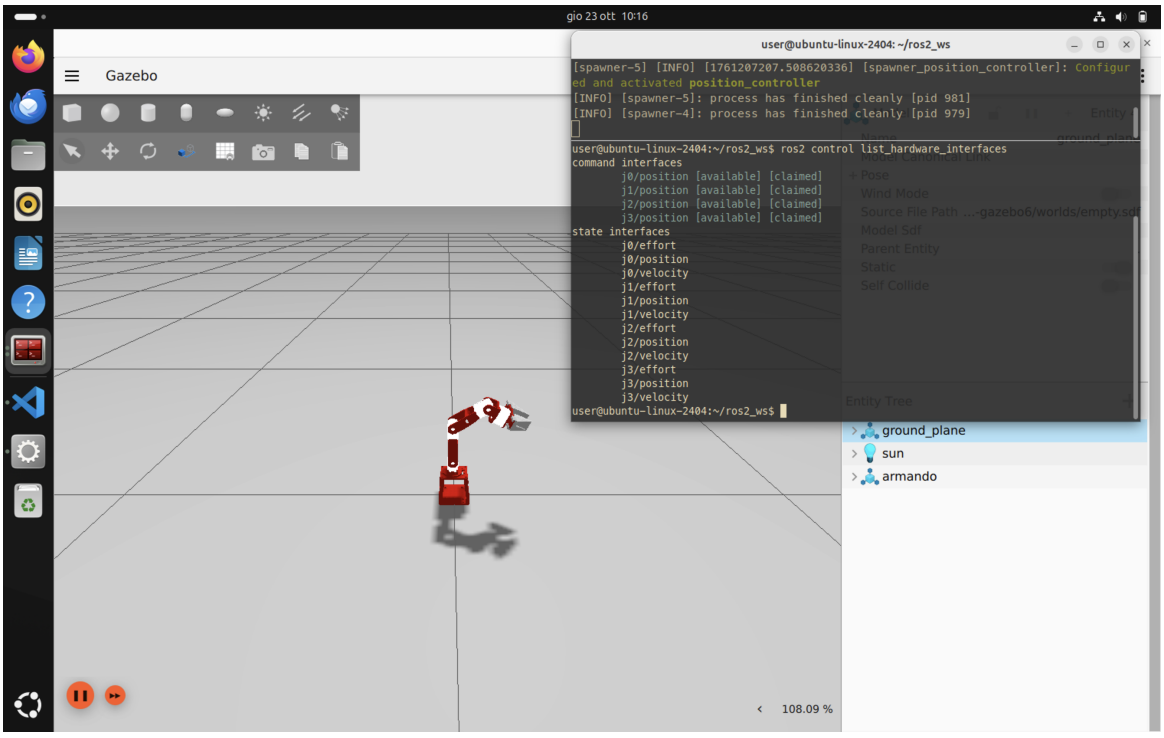
```
  <xacro:arg name="joint_j0_pos" default="0.0"/>
  <xacro:arg name="joint_j1_pos" default="0.0"/>
  <xacro:arg name="joint_j2_pos" default="1.0"/>
  <xacro:arg name="joint_j3_pos" default="1.0"/>

  <ros2_control name="HardwareInterface_Ignition" type="system">
    <hardware>
    <plugin>ign_ros2_control/IgnitionSystem</plugin>
    </hardware>
    <xacro:PositionJointInterface name="j0" initial_pos="$(arg joint_j0_pos)"/>
    <xacro:PositionJointInterface name="j1" initial_pos="$(arg joint_j1_pos)"/>
    <xacro:PositionJointInterface name="j2" initial_pos="$(arg joint_j2_pos)"/>
    <xacro:PositionJointInterface name="j3" initial_pos="$(arg joint_j3_pos)"/>
  </ros2_control>
```

The file `armando_controller.yaml` defines two types of controllers: `JointStateBroadcaster` and `JointGroupPositionController`.

Then we spawned the joint state broadcaster and the position controllers using the `controller_manager` package from the `armando_world.launch`. The spawn needs to be done after that gazebo loaded the model of the robot (using `RegisterEventHandler()`).

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

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

delay_spawners_after_spawn = RegisterEventHandler(
event_handler=OnProcessExit(
target_action=spawn_entity_node,
on_exit=[joint_state_broadcaster_node,
    position_controller_node],))
```

The hardware interface is finally correctly loaded and connected.

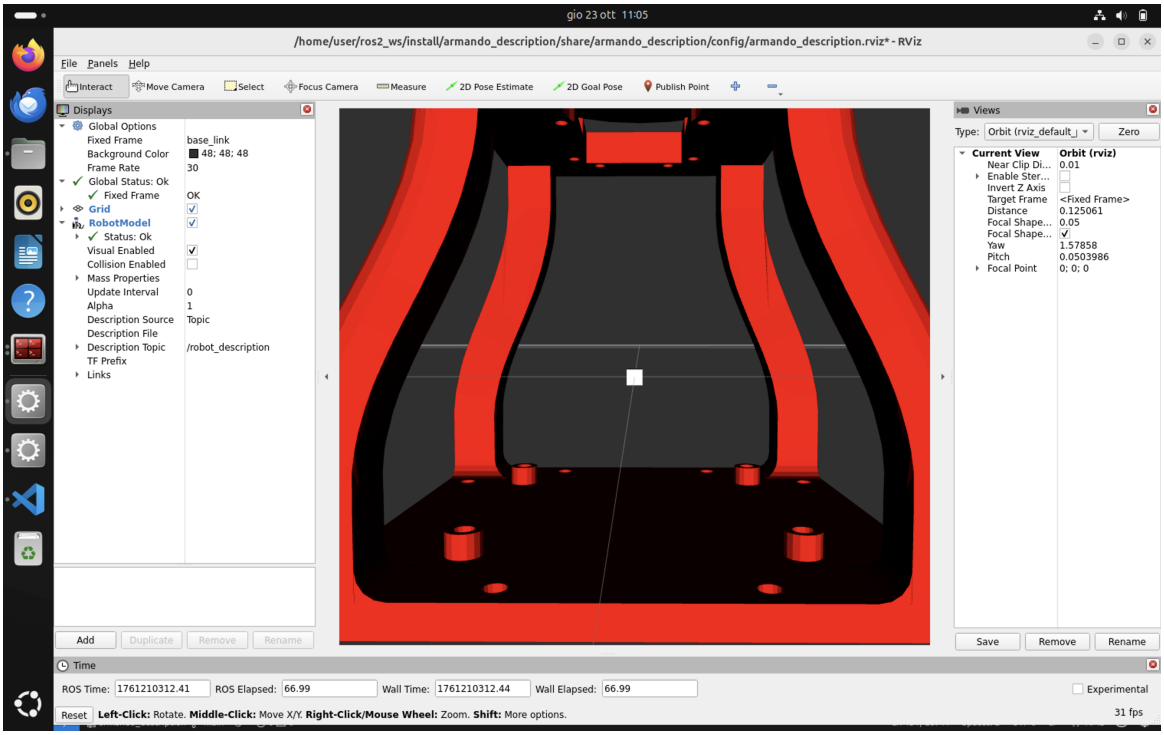


3 Add a camera sensor to your robot

(a) We added into the `armando.urdf.xacro` file a `camera_link` and a fixed `camera_joint` with `base_link` as a parent link. We sized and positioned the camera link opportunely at the base of the robot.

```
<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 -1.57"/>
</joint>
```



(b) We created an `armando_camera.xacro` file in the `armando_gazebo/urdf` folder. The `armando_camera.xacro` file contains the sensor specifications within a `xacro:macro` and the `gz-sim-sensors-system` plugin.

```
  <?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>
```

Then we imported the `armando_camera.xacro` file in `armando.urdf.xacro` using the `xacro:include` command

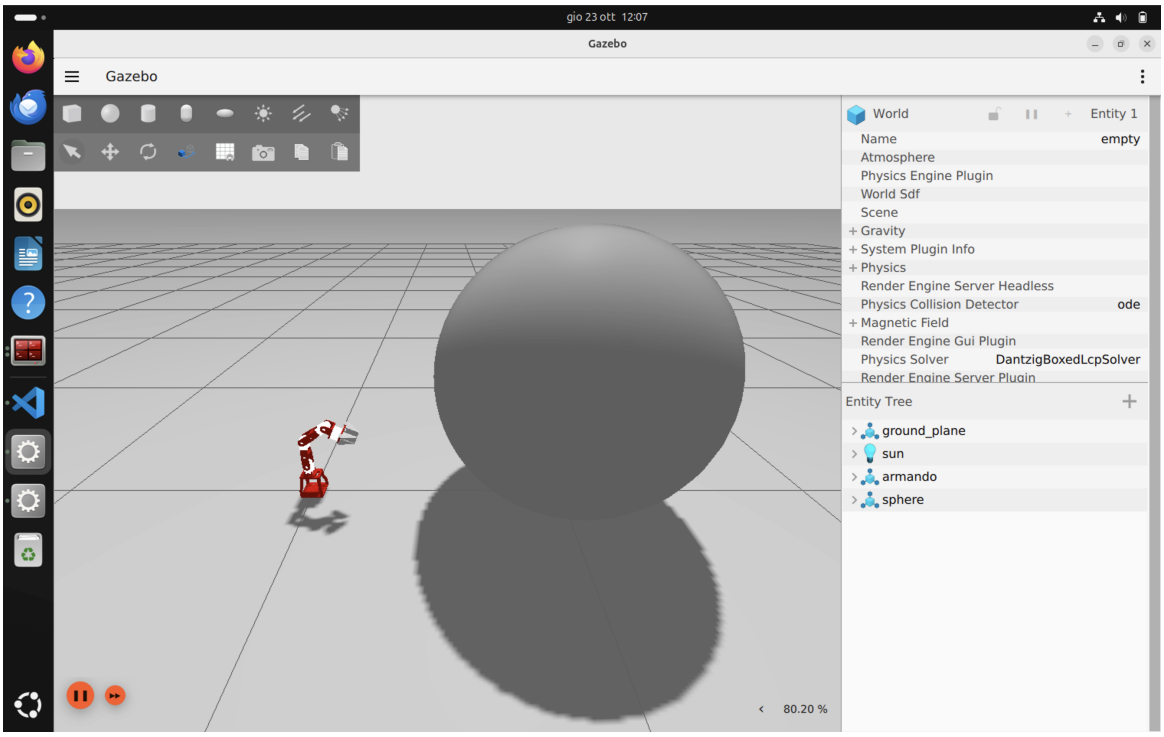
```
<xacro:include filename="$(find armando_gazebo)/urdf/armando_camera.xacro"/>
<xacro:armando_camera link="camera_link"/>
```

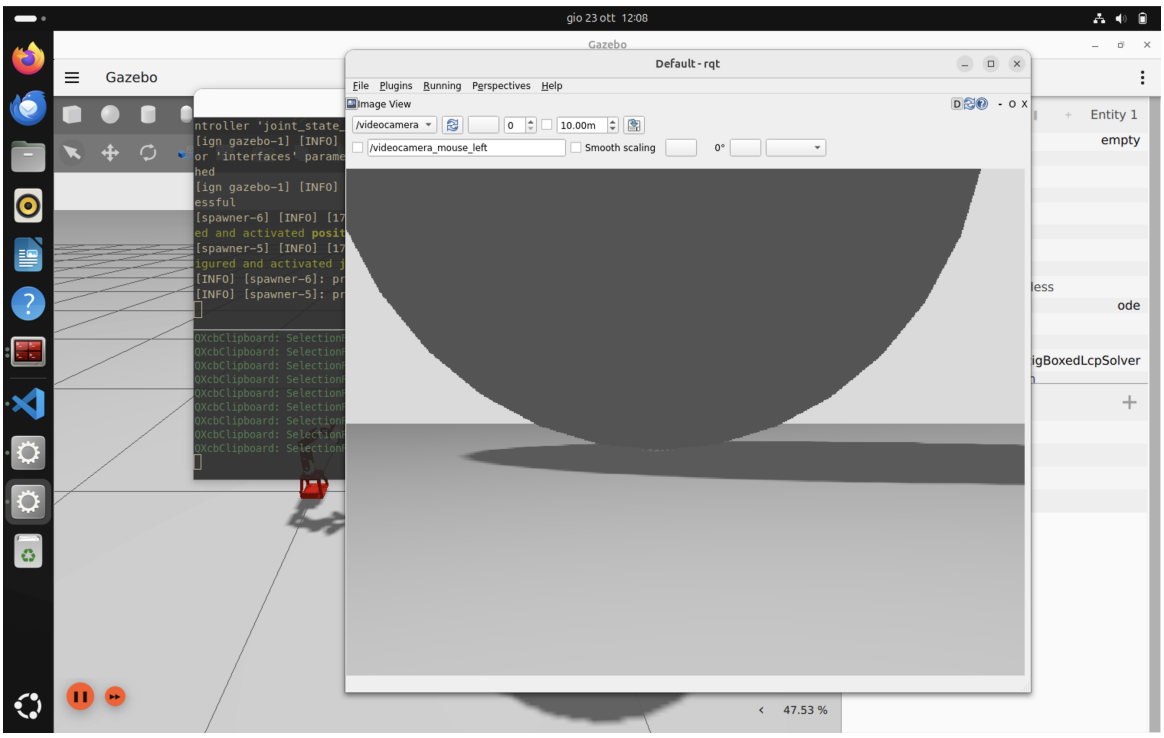
(c) Before launching the Gazebo simulation, we added the `ros_ign_bridge` commands into the launch file `armando_world.launch.py`

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

We started the Gazebo simulation using `armando_gazebo.launch` file. To check if the image topic is correctly published we inserted a sphere into the simulation environment; then we opened a new terminal and used the command:

```
$  ros2 run rqt_image_view rqt_image_view
```





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

(a) We created the `armando_controller` package with a ROS C++ node named `arm_controller_node` with the following instructions executed in the terminal:

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

The dependencies are `rclcpp`, `sensor_msgs` and `std_msgs`. Thus, the `CMakeLists.txt` and the `package.xml` files have been opportunely modified to compile the new node according to the specified dependencies. In particular:

- `CMakeLists.txt`:

```
find_package(rclcpp REQUIRED)
find_package(sensor_msgs REQUIRED)
find_package(std_msgs REQUIRED)
add_executable(${PROJECT_NAME}_node src/arm_controller_node.cpp)
ament_target_dependencies(${PROJECT_NAME}_node rclcpp std_msgs sensor_msgs)

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

- `package.xml`:

```
<build_depend>std_msgs</build_depend>
<build_depend>sensor_msgs</build_depend>
<build_depend>rclcpp</build_depend>

<exec_depend>sensor_msgs</exec_depend>
<exec_depend>std_msgs</exec_depend>
<exec_depend>rclcpp</exec_depend>
```

(b) In the node `arm_controller_node`, we created a subscriber to the topic `joint_states` and a callback function that prints the current joint positions of the robot.

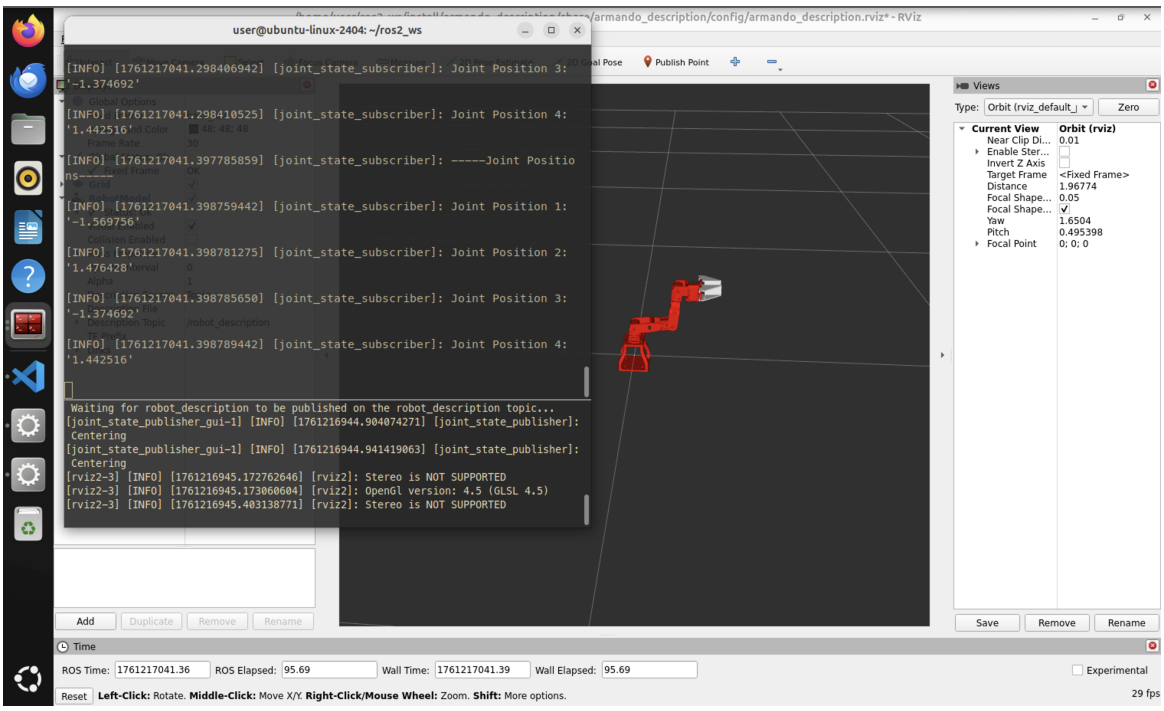
We created the node class `JointStateSubscriber` by inheriting from `rclcpp::Node`. The constructor uses the node's `create_subscription` class to execute the callback. There is no timer because the subscriber

simply responds whenever data is published to the topic `joint_states`. The `topic_callback` function receives the array of joint position published over the topic, and simply writes it to the console using the `RCLCPP_INFO` macro.

Finally, the `JointStateSubscriber` class is called in the main function, where the node actually executes.

```
class JointStateSubscriber : public rclcpp::Node
{
  public:
    JointStateSubscriber()
    : Node("joint_state_subscriber")
    {
      subscription_ = this-> create_subscription<sensor_msgs::msg::JointState>(
      "joint_states", 10, 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-----\n");
      for (int i = 0; i < 4; i++)
        RCLCPP_INFO(this->get_logger(), "Joint Position %d: '%6f'\n", 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;
}
```

From the terminal:



(c) Initially, we created the node class `PositionControllerPublisher`, similarly to the joint state subscriber node. The main difference is that the node declares a parameter named `joint_positions`, which stores the target 4 joints variables as 4 vectors of doubles. The publisher writes sequentially the positions every 10 seconds onto the `/position_controllers/command` topics. Moreover, the public constructor names

the node `position_controller_publisher` and initializes `current_sequence_index_` to 0. Inside the constructor, the publisher is initialized with the `Float64MultiArray` message type, the topic name `/position_controllers/command`, and the required queue size to limit messages.

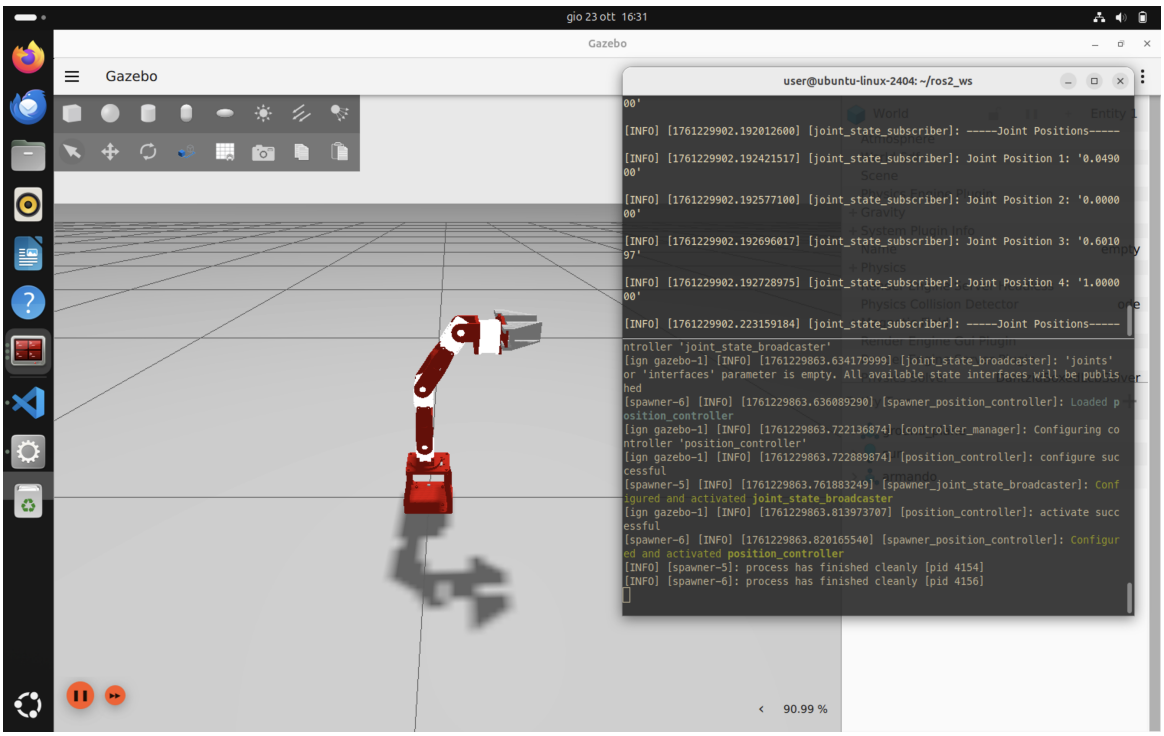
Next, `timer_` is initialized, which causes the `timer_callback` function to be executed every time there is a timer timeout. The `timer_callback` function is where the message data is set and the messages are actually published. The `RCLCPP_INFO` macro ensures every published message is printed to the console.

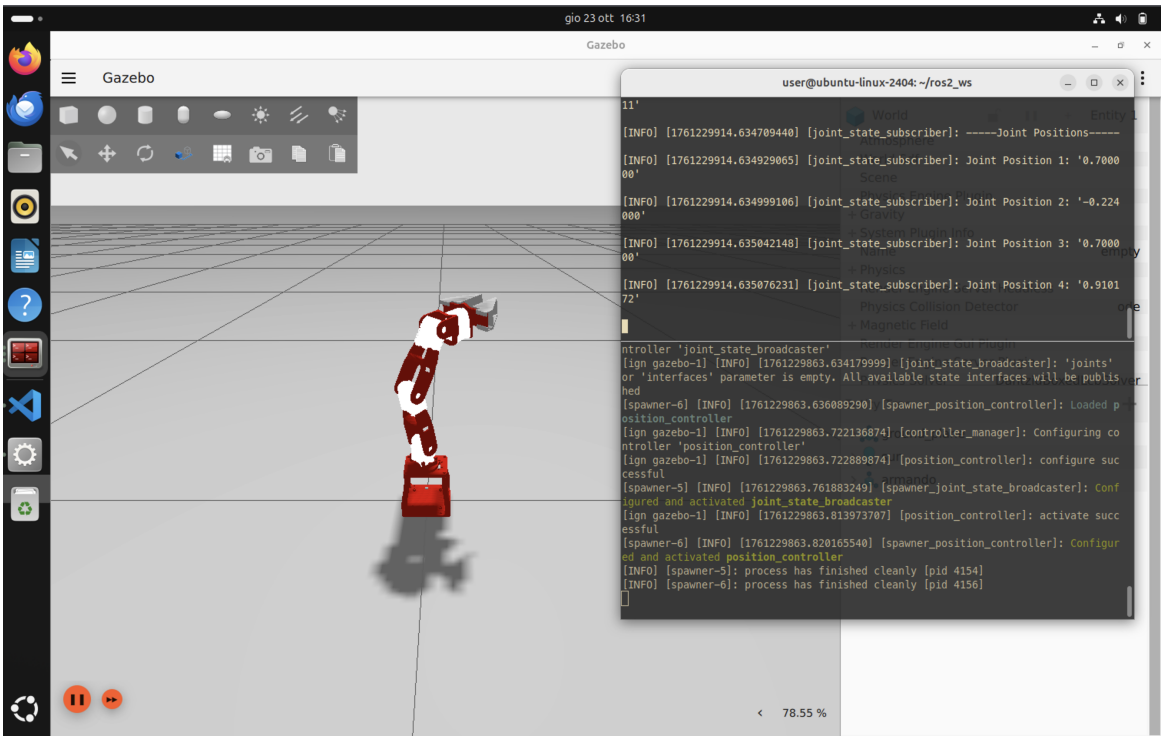
```
class PositionControllerPublisher : public rclcpp::Node
{
public:
PositionControllerPublisher()
: Node("position_controller_publisher"), current_sequence_index_(0)
{
    target_positions_sequence_ = {
        {0.0, 0.0, 0.0, 1.0},
        {0.7, 0.0, 0.7, 1.0},
        {0.7, -1.0, 0.7, 0.0},
        {0.0, 0.0, 0.0, 0.0}
      };
    publisher_ = this->create_publisher<std_msgs::msg::Float64MultiArray>("/position_controller/commands",
    timer_ = this->create_wall_timer(
    10000ms, std::bind(&PositionControllerPublisher::timer_callback, this));
  }
private:
void timer_callback()
{
const auto& current_positions = target_positions_sequence_[current_sequence_index_];
auto commands = std_msgs::msg::Float64MultiArray();
commands.data = current_positions;
RCLCPP_INFO(this->get_logger(), "Publishing sequence step: %zu", current_sequence_index_);
std::string positions_str = "[";
for (size_t i = 0; i < current_positions.size(); ++i) {
positions_str += std::to_string(current_positions[i]) +
(i < current_positions.size() - 1 ? ", " : "]");
}
RCLCPP_INFO(this->get_logger(), "  Positions: %s", positions_str.c_str());
publisher_->publish(commands);
current_sequence_index_ = (current_sequence_index_ + 1) % target_positions_sequence_.size();
}
rclcpp::TimerBase::SharedPtr timer_;
rclcpp::Publisher<std_msgs::msg::Float64MultiArray>::SharedPtr publisher_;
std::vector<std::vector<double>> target_positions_sequence_;
size_t current_sequence_index_;
};
```

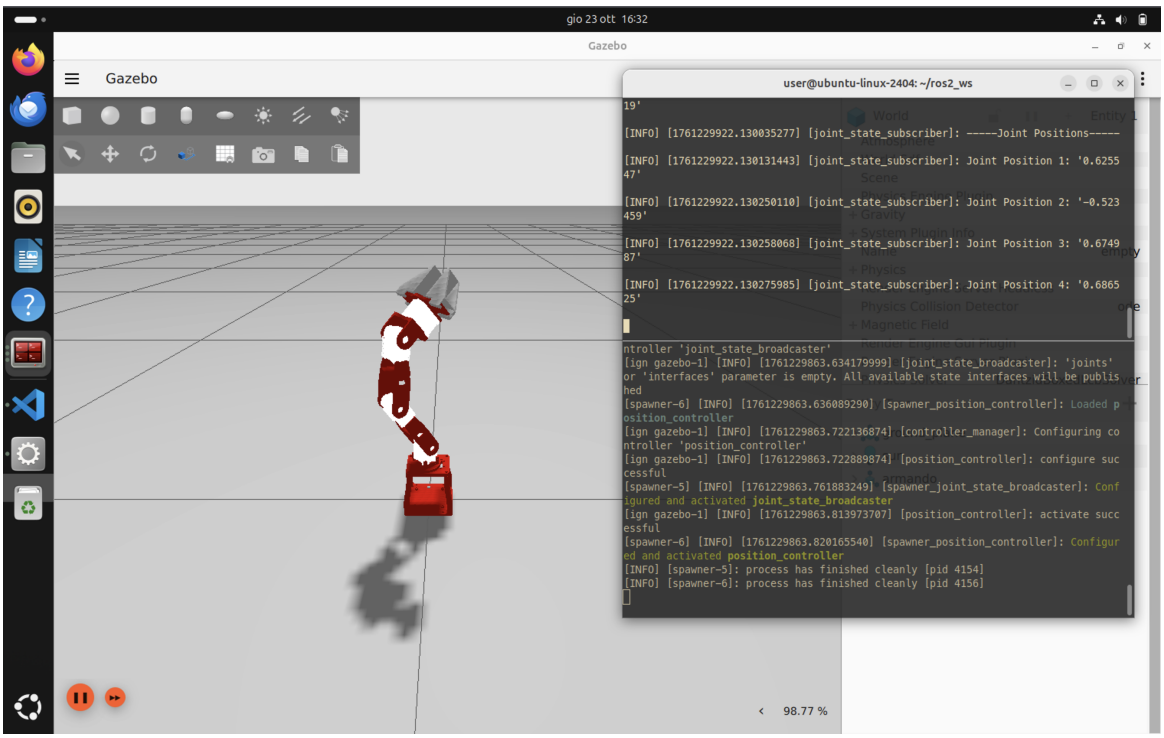
To run two nodes into the same process the main function has been modified as follows.

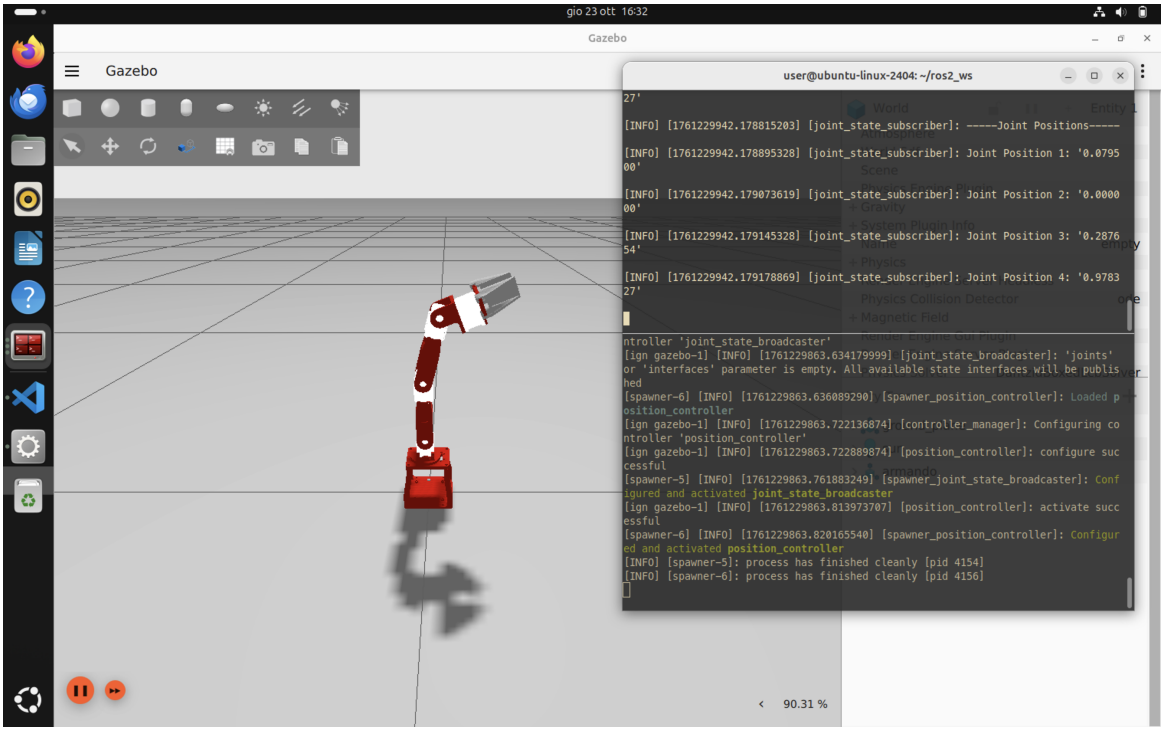
```
int main(int argc, char * argv[])
{
  rclcpp::init(argc, argv);
  auto joint_state_subscriber = std::make_shared<JointStateSubscriber>();
  auto position_controller_publisher = std::make_shared<PositionControllerPublisher>();
  rclcpp::executors::MultiThreadedExecutor executor;
  executor.add_node(joint_state_subscriber);
  executor.add_node(position_controller_publisher);
  executor.spin();
  rclcpp::shutdown();
  return 0;
```

Finally we started the Gazebo simulation. We report here some snapshots.


gio 23 ott 16:31
Gazebo
Gazebo
user@ubuntu-linux-2404: ~/ros2_ws
00'
[INFO] [1761229902.192012600] [joint_state_subscriber]: -----Joint Positions-----
[INFO] [1761229902.192421517] [joint_state_subscriber]: Joint Position 1: '0.049000'
[INFO] [1761229902.192577100] [joint_state_subscriber]: Joint Position 2: '0.000000'
[INFO] [1761229902.192696017] [joint_state_subscriber]: Joint Position 3: '0.601097'
[INFO] [1761229902.192728975] [joint_state_subscriber]: Joint Position 4: '1.000000'
[INFO] [1761229902.223159184] [joint_state_subscriber]: -----Joint Positions-----
ntroller 'joint_state_broadcaster'
[ign gazebo-1] [INFO] [1761229863.634179999] [joint_state_broadcaster]: 'joints' or 'interfaces' parameter is empty. All available state interfaces will be published
[spawner-6] [INFO] [1761229863.636089290] [spawner_position_controller]: Loaded position_controller
[ign gazebo-1] [INFO] [1761229863.722136874] [controller_manager]: Configuring controller 'position_controller'
[ign gazebo-1] [INFO] [1761229863.722889874] [position_controller]: configure successful
[spawner-5] [INFO] [1761229863.761883249] [spawner_joint_state_broadcaster]: Configured and activated joint_state_broadcaster
[ign gazebo-1] [INFO] [1761229863.813973707] [position_controller]: activate successful
[spawner-6] [INFO] [1761229863.820165540] [spawner_position_controller]: Configured and activated position_controller
[INFO] [spawner-5]: process has finished cleanly [pid 4154]
[INFO] [spawner-6]: process has finished cleanly [pid 4156]
90.99 %


gio 23 ott 16:31
Gazebo
Gazebo
user@ubuntu-linux-2404: ~/ros2_ws
11'
[INFO] [1761229914.634709440] [joint_state_subscriber]: -----Joint Positions-----
[INFO] [1761229914.634929065] [joint_state_subscriber]: Joint Position 1: '0.700000'
[INFO] [1761229914.634999106] [joint_state_subscriber]: Joint Position 2: '-0.224000'
[INFO] [1761229914.635042148] [joint_state_subscriber]: Joint Position 3: '0.700000'
[INFO] [1761229914.635076231] [joint_state_subscriber]: Joint Position 4: '0.910172'
ntroller 'joint_state_broadcaster'
[ign gazebo-1] [INFO] [1761229863.634179999] [joint_state_broadcaster]: 'joints' or 'interfaces' parameter is empty. All available state interfaces will be published
[spawner-6] [INFO] [1761229863.636089290] [spawner_position_controller]: Loaded position_controller
[ign gazebo-1] [INFO] [1761229863.722136874] [controller_manager]: Configuring controller 'position_controller'
[ign gazebo-1] [INFO] [1761229863.722889874] [position_controller]: configure successful
[spawner-5] [INFO] [1761229863.761883249] [spawner_joint_state_broadcaster]: Configured and activated joint_state_broadcaster
[ign gazebo-1] [INFO] [1761229863.813973707] [position_controller]: activate successful
[spawner-6] [INFO] [1761229863.820165540] [spawner_position_controller]: Configured and activated position_controller
[INFO] [spawner-5]: process has finished cleanly [pid 4154]
[INFO] [spawner-6]: process has finished cleanly [pid 4156]
78.55 %





(d) We modified the `armando_controller.yaml` by adding the declaration of a new type of controller, that is the trajectory controller with specified parameters.

```
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
```

We also modified the `armando_world.launch.py` in order to allow the user to choose the type of controller by means of an argument that must be specified by the user (using `DeclareLaunchArgument()`) in the command line when launching the launch file:

```
$ ros2 launch armando_gazebo armando_world.launch.py controller_type:=<type>
```

where `type` can be `position` or `trajectory`. If nothing is specified the default choose will be `position`.

Into the launch file there is the definition of two nodes, one for the trajectory controller and another one for the position controller. Basing on the argument specified by the user, only one node will be created at run time (there is an `IfCondition()` into the nodes definitions). For example, we report the definition of the node for the trajectory controller.

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

The publisher of the `arm_controller_node.cpp` was modified in order to select the correct message to be sent through the respective topic.

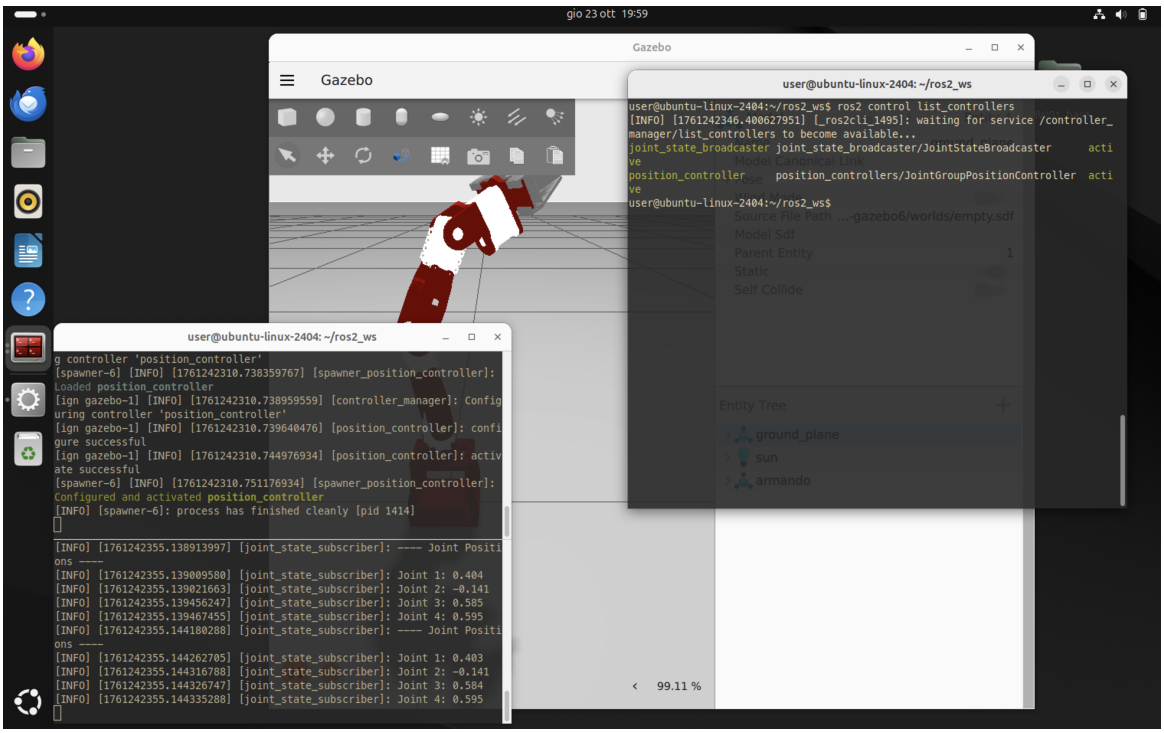
```
class ControllerPublisher : public rclcpp::Node
{
public:
ControllerPublisher()
: Node("controller_publisher"),
current_step_(0)
{
this->declare_parameter<bool>("use_trajectory", false);
use_trajectory_ = this->get_parameter("use_trajectory").as_bool();

target_pos_ = {
{0.0, 0.0, 0.0, 1.0},
{0.7, 0.0, 0.7, 1.0},
{0.7, -1.0, 0.7, 0.0},
{0.0, 0.0, 0.0, 0.0}
};
if (use_trajectory_)
{
    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");
  }

timer_ = this->create_wall_timer(
5000ms, std::bind(&ControllerPublisher::timer_callback, this)
);
```

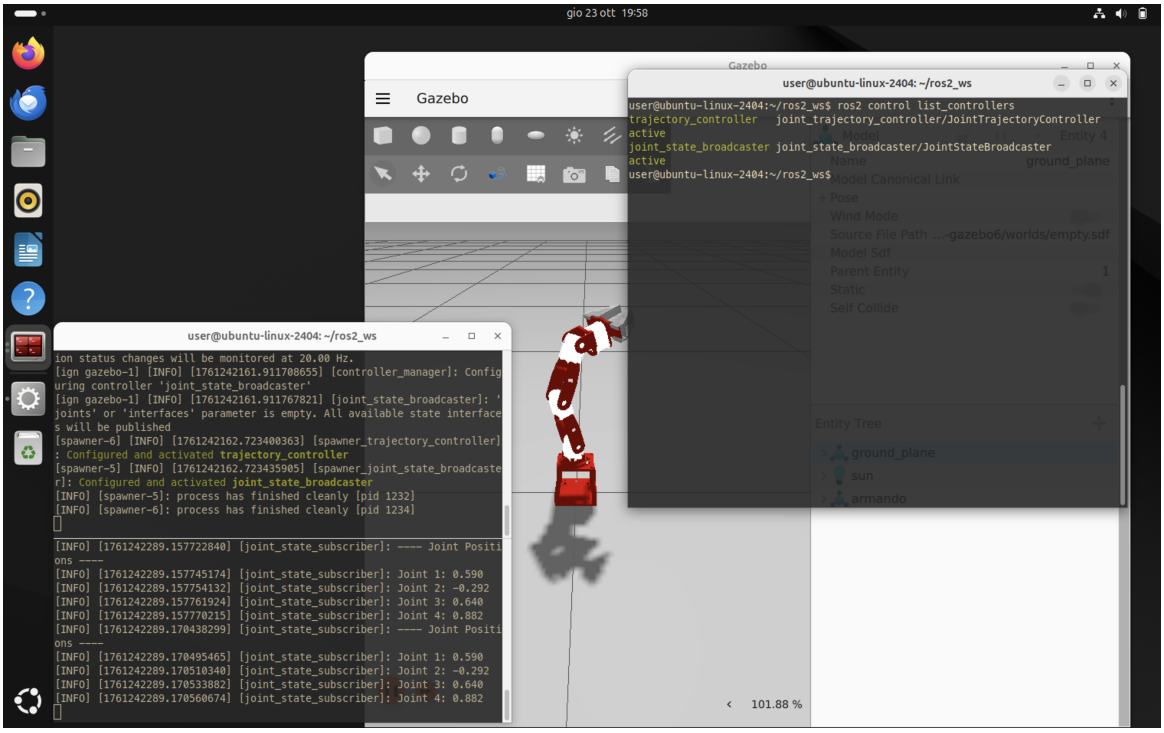
```
  }
  private:
  void timer_callback()
  {
  const auto &pos = target_pos_[current_step_];
  if (use_trajectory_)
  {
  trajectory_msgs::msg::JointTrajectory traj;
  traj.joint_names = {"j0", "j1", "j2", "j3"};
  traj.points.resize(1);
  traj.points[0].positions = pos;
  traj.points[0].time_from_start.sec = 5;
  traj.points[0].time_from_start.nanosec = 0;
  publisher_traj_->publish(traj);
  RCLCPP_INFO(this->get_logger(),
  "Sent Trajectory step %zu", current_step_);
  }
  else
    {
      std_msgs::msg::Float64MultiArray cmd;
      cmd.data = pos;
      publisher_pos_->publish(cmd);
      RCLCPP_INFO(this->get_logger(),
      "Sent Position step %zu", current_step_);
    }

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

  bool use_trajectory_;
  size_t current_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_;
  };
```

When executing from the command line the `arm_controller_node.cpp`, the user has to specify the value of the boolean argument `use_trajectory` (`true` for trajectory control, `false` for position control).

```
$ ros2 run armando_controller arm_controller_node --ros-args -p use_trajectory:=true
```

We report two snapshots: the first one for position control, the second one for trajectory control.





Note: In the Git version (see README.md), to improve the management of controllers and Gazebo — since switching controllers previously required restarting Gazebo — we separated the Gazebo launch files from the controller launch files. As a result, the controller package now takes an argument specifying which controller to launch.