Robotics Lab: Homework 1

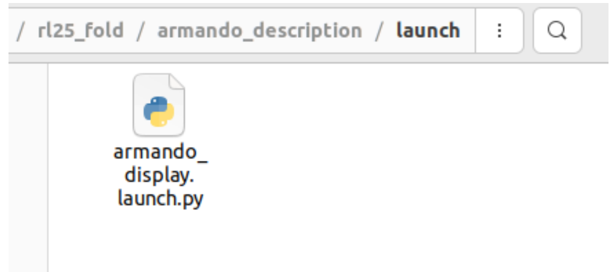
Bring up your robot

Student1: https://github.com/carlobarone00
Student2: https://github.com/Moreno-Zaccara
Student3: https://github.com/Student3
Student4: https://github.com/Student4

1 Robot Visualization Setup in RViz

(a) creating a launch folder within the `armando_description` package containing the launch file named `armando_display.launch`

(a) Launch folder created inside `armando_description`



(b) Content of the `armando_display.launch` file

Figure 1: Creation of the launch folder and corresponding launch file in `armando_description`

Specifically, the launch file has the structure shown in the following image:

```python
from launch import LaunchDescription
from launch_ros.actions import Node
from ament_index_python.packages import get_package_share_directory
import os

def generate_launch_description():
    # Percorso al package armando_description
    arm_description_path = os.path.join(get_package_share_directory('armando_description'))

    # Percorso al file urdf
    urdf_path = os.path.join(arm_description_path, "urdf", "arm.urdf")

    # Percorso al file rviz config
    #rviz_config = os.path.join(arm_description_path, "config", "config.rviz")

    # Legge il file URDF
    with open(urdf_path, 'r') as infp:
        robot_desc = infp.read()

    robot_description = {"robot_description": robot_desc}

    # Nodo robot_state_publisher
    robot_state_publisher_node = Node(
        package="robot_state_publisher",
        executable="robot_state_publisher",
        output="screen",
        parameters=[robot_description]
    )

    # Nodo joint_state_publisher
    joint_state_publisher_node = Node(
        package="joint_state_publisher",
        executable="joint_state_publisher",
        output="screen"
    )

    # Nodo rviz2 con configurazione rviz (se config mancante, togli arguments)
    rviz_node = Node(
        package="rviz2",
        executable="rviz2",
        name="rviz2",
        output="screen",
        #arguments=["-d", rviz_config]
    )

    return LaunchDescription([
        robot_state_publisher_node,
        joint_state_publisher_node,
        rviz_node
    ])
```

Figure 2: Structure of the `armando_display.launch` file.

As shown in the following figure, the launch file initializes three essential ROS 2 nodes:
the `robot_state_publisher`, the `joint_state_publisher`, and the `rviz2` node.
The `robot_state_publisher` node is responsible for broadcasting the transformations between the robot's links based on the URDF description. The `joint_state_publisher` node publishes the current joint positions, either manually specified or read from hardware interfaces, while `rviz2` provides the visualization environment.

To correctly visualize the robot model in RViz, we configured the display by setting the **Fixed Frame** parameter to the robot's reference frame and by adding a `Robot Model` display to render the URDF description. This ensures that all subsequent transformations and sensor data are aligned with the correct reference frame.

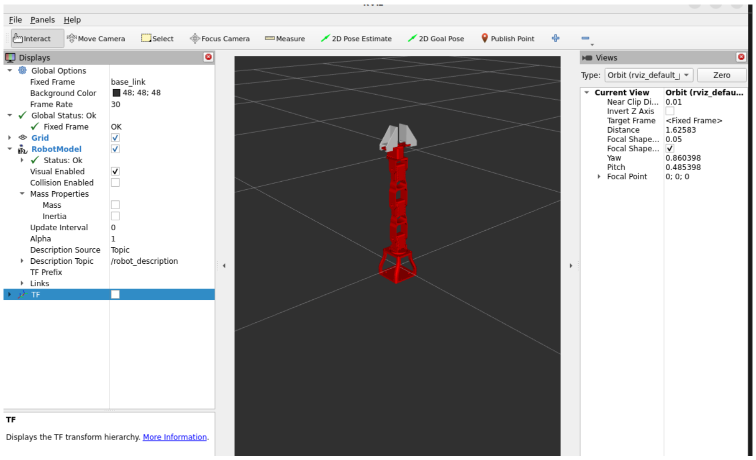
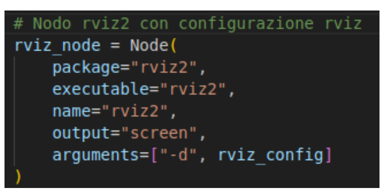


Figure 3: Visualization of the Armando robot model in RViz after configuring the `Robot Model` display.

(b) RViz Configuration File

Inside the `config` directory of the `armando_description` package, we added a configuration file named `config.rviz`. This file stores the customized RViz layout, including the loaded displays, visual parameters, and the selected fixed frame.



```python
# Nodo rviz2 con configurazione rviz
rviz_node = Node(
    package="rviz2",
    executable="rviz2",
    name="rviz2",
    output="screen",
    arguments=["-d", rviz_config]
)
```

Figure 4: RViz node configuration for automatic robot model visualization.

(c) Simplification of Collision Models

To improve simulation performance and stability, we replaced the detailed collision meshes in the URDF with simplified *primitive geometries*. In particular, we used box-shaped elements as approximations of the original link volumes. This approach maintains reasonable physical accuracy while significantly reducing computational cost during collision detection.

The collision properties for each link are defined by the `box size` attribute in the URDF file. An example of one link's collision box definition is shown below, but the same procedure was iteratively applied to every joint and link of the robot model.

```xml
<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="0.1 0.1 0.1"/>
    </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>
```

Figure 5: Example of a collision box definition within the URDF file.

The resulting model, shown in the following figure, demonstrates the simplified collision geometry of the robot. This representation enables efficient collision checking while preserving the robot's kinematic structure for both visualization and physical simulation purposes.

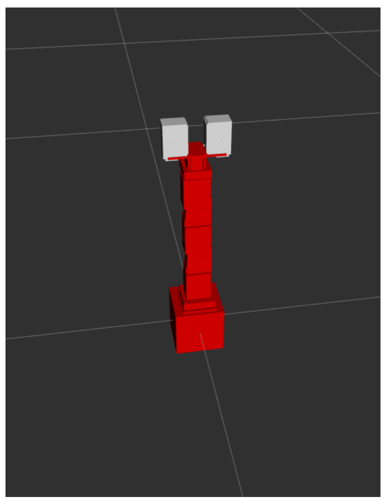


Figure 6: Final simplified collision model of the Armando robot.

2 Setting Up the Gazebo Simulation and Hardware Interface

(a) Creating a package using the terminal by using the following command line:

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

Once created, since the `ros2 pkg create` command do not create all the folders we need, we also made sure to add the launch directory (inside the package that we jsut created) with the following command line:

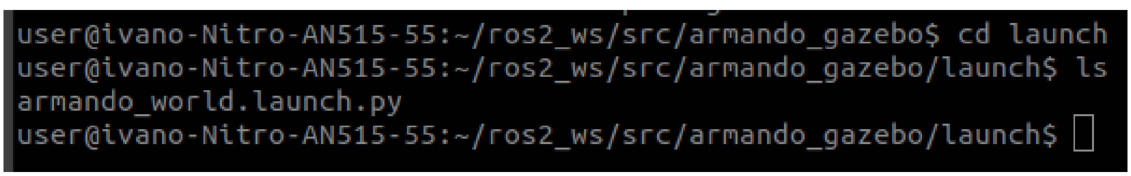
```
mkdir launch
```

Once done the previous steps, our package will have the following stuff inside:

```
user@ivano-Nitro-AN515-55:~/ros2_ws/src/armando_gazebo$ ls
CMakeLists.txt  README.txt  launch  package.xml  urdf
```

Figure 7: Structure of the `armando_gazebo` package with the launch folder added.

The `armando_world.launch.py` file is included inside the launch directory and it's filled with commands to load the URDF file into the `/robot_description` topic and spawn the robot using the `create node` in the `ros_gz_sim` package:



```
user@ivano-Nitro-AN515-55:~/ros2_ws/src/armando_gazebo$ cd launch
user@ivano-Nitro-AN515-55:~/ros2_ws/src/armando_gazebo/launch$ ls
armando_world.launch.py
user@ivano-Nitro-AN515-55:~/ros2_ws/src/armando_gazebo/launch$
```

Figure 8: Launch file for loading the robot model and spawning it in Gazebo.

The robot gets spawned thanks to the following code line:

```python
def generate_launch_description():
    gui_arg = DeclareLaunchArgument(
        name='gui',
        default_value='true',
    )
    path_armando = os.path.join(
        get_package_share_directory('armando_description'))
    package_arg = DeclareLaunchArgument('urdf_package',
                                        description='The package where the robot description is located',
                                        default_value='armando_description')
    model_arg = DeclareLaunchArgument('urdf_package_path',
                                      description='The path to the rfobot description relative to the package root',
                                      default_value='urdf/armando.urdf.xacro')

    choice_controll_arg = DeclareLaunchArgument(
    name='choice_controll',
    default_value='0',
    description='Choose which controller to use (0=position, 1=trajectory)'
    )

    choice_controll = LaunchConfiguration('choice_controll')

    empty_world_launch = IncludeLaunchDescription(
        PathJoinSubstitution([FindPackageShare('ros_gz_sim'), 'launch', 'gz_sim.launch.py']),
        launch_arguments={
            'gui': LaunchConfiguration('gui'),
            'pause': 'true',
            'gz_args': ['-r ', 'empty.sdf'],
        }.items(),
    )

    description_launch_py = IncludeLaunchDescription(
        PathJoinSubstitution([FindPackageShare('urdf_launch'), 'launch', 'description.launch.py']),
        launch_arguments={
            'urdf_package': LaunchConfiguration('urdf_package'),
            'urdf_package_path': LaunchConfiguration('urdf_package_path')}.items()
    )

    xacro_armando = os.path.join(path_armando, "urdf", "armando.urdf.xacro")

    robot_description_armando_xacro = {"robot_description": Command(['xacro ', xacro_armando, ' joint_a3_pos:=2.0', ' joint_a4_pos:=0.2'])}
```

Figure 9: Spawning of the Robot code - 1

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

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

    urdf_spawner_node = Node(
        package='ros_gz_sim',
        executable='create',
        name='urdf_spawner',
        arguments=['-topic', '/robot_description', '-entity', 'robot', '-z', '0.5', '-unpause'],
        output='screen',
    )
```

Figure 10: Spawning of the Robot code - 2

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

position_controller_spawner = Node(
package="controller_manager",
executable="spawner",
arguments=["position_controller", "--controller-manager", "/controller_manager"],
condition=IfCondition(PythonExpression([choice_controll, ' == 0'])),
)

delay_joint_pos_controller = (
    RegisterEventHandler(
        event_handler=OnProcessExit(
            target_action=urdf_spawner_node,
            on_exit=[
                position_controller_spawner,
                ],
        )
    )
)
delay_joint_state_broadcaster = (
    RegisterEventHandler(
        event_handler=OnProcessExit(
            target_action=urdf_spawner_node,
            on_exit=[joint_state_broadcaster_spawner],
        )
    )
)
```

Figure 11: Spawning of the Robot code - 3

(b) To enable position control of the robot's joints in simulation, we added a `PositionJointInterface` named `HardwareInterface_Ignition`, implemented using the `ros2_control` framework. This interface allows the controllers to send and receive commands corresponding to the robot's joint positions, bridging the gap between the simulated hardware (in Gazebo) and the ROS 2 control nodes.

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

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

  <xacro:joint_ros2_control name="j0" initial_pos="0.0"/>
  <xacro:joint_ros2_control name="j1" initial_pos="0.0"/>
  <xacro:joint_ros2_control name="j2" initial_pos="0.0"/>
  <xacro:joint_ros2_control name="j3" initial_pos="0.0"/>

</ros2_control>
```

Figure 12: Definition of the `PositionJointInterface` within the `ros2_control` framework.

The hardware interface is defined inside a dedicated file named `armando_hardware_interface.xacro`, located in the `armando_description/urdf` directory. This file contains a `xacro:macro` definition that describes how the robot's joints are mapped to the ROS 2 control interfaces, specifying which joints are actuated and how they interact with the simulation backend.

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


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

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

</xacro:macro>


</robot>
```

Figure 13: Excerpt from the `armando_hardware_interface.xacro` file showing the macro definition.

The hardware interface macro is then included within the main `armando.urdf.xacro` file (previously renamed from `armando.urdf`) using the `xacro:include` directive. This inclusion makes the hardware interface part of the robot's unified URDF structure, allowing the controller manager to load and initialize the correct control resources during simulation startup.

```xml
<robot name="armando" xmlns:xacro="http://www.ros.org/wiki/xacro">
  <xacro:include filename="$(find armando_description)/urdf/armando_hardware_interface.xacro"/>
```

Figure 14: Inclusion of the hardware interface macro inside the main URDF description.

(c) In the main `armando.urdf.xacro` file, the Gazebo ROS 2 Control plugin is included to load and manage the joint position controllers.

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

Figure 15: Inclusion of the controller configuration YAML file in the URDF.

The YAML file defines the parameters and interfaces for both the `joint_state_broadcaster` and the `position_controller`.

```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

    command_interfaces:
      - position

    state_interfaces:
      - position
      - velocity

    state_publish_rate: 100.0
    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
```

Figure 16: YAML configuration defining the robot controllers.

The controllers are spawned in the `armando_world.launch.py` file inside the `armando_gazebo/launch` directory.

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

position_controller_spawner = Node(
package="controller_manager",
executable="spawner",
arguments=["position_controller", "--controller-manager", "/controller_manager"],
condition=IfCondition(PythonExpression([choice_controll, ' == 0'])),
)
```

Figure 17: Commands for spawning the joint state broadcaster and position controllers.

The Gazebo simulation is launched with:

```
ros2 launch armando_gazebo armando_world.launch.py
```

When started, the terminal confirms that the hardware interface and controllers are correctly loaded.

Figure 18: Verification of the controllers successfully loaded in simulation.

3 Camera Setup and Visualization in Gazebo

(a) Adding a `carmera_link` and a `camera_joint` of fixed type in order to define the link and the join used to setup the camera for the Gazebo environment:

```
<joint name="camera_joint" type="fixed">
    <parent link="base_link"/>
    <child link="camera_link"/>
    <origin xyz="0.0 0 0.017" rpy="-1.57 -1.57 0">
</joint>

<link name="camera_link">
    <visual>
        <geometry>
            <box size="0.5 0.5 0.5"/>
        </geometry>
        <material name="red"/>
    </visual>
</link>
```

Figure 19: Camera Code

The `camera_link` and the `camera_joint` are both included inside the `armando.urdf.xacro` file. The values for size and position have been chosen in orded to let the camera be visible in the simulation environment, while the rpy axis have been chosen in order to let our camera be oriented from the bottom to the top.

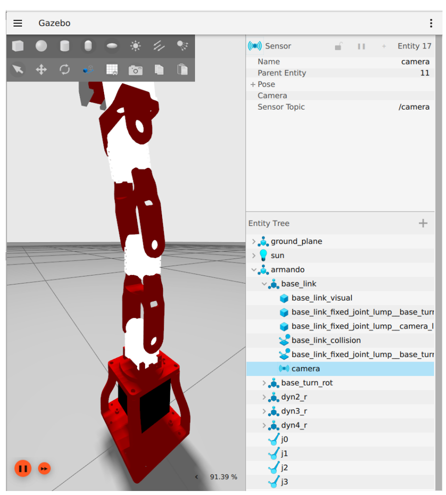


Figure 20: Camera in the Robot

(b) The `armando_camera.xacro` file was created inside the `armando_gazebo/urdf` folder.

```
user@ivano-Nitro-AN515-55:~/ros2_ws/src/armando_gazebo/urdf$ ls
armando_camera.xacro
```

Figure 21: Location of the `armando_camera.xacro` file in the package structure.

This file defines the camera sensor using a `xacro:macro` and includes the necessary Gazebo plugins for image publishing.

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


  <xacro:macro name="camera_sensor" params="parent_link xyz rpy name topic_name">

    <link name="${name}_link">
      <visual>
        <geometry>
          <box size="0.05 0.05 0.05"/>
        </geometry>
        <material name="green"/>
      </visual>
    </link>

    <joint name="${name}_joint" type="fixed">
      <parent link="${parent_link}"/>
      <child link="${name}_link"/>
      <origin xyz="${xyz}" rpy="${rpy}"/>
    </joint>


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

Figure 22: Macro definition and plugin configuration inside `armando_camera.xacro`.

The macro is included in the main `armando.urdf.xacro` file using the `xacro:include` directive.

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

Figure 23: Inclusion of the camera macro in the main URDF description.

(c) By using the command line `ros2 run rqt_image_view rqt_image_view` it's possible to see that the topic is correctly published, in fact the camera also shows what it's pointing to:

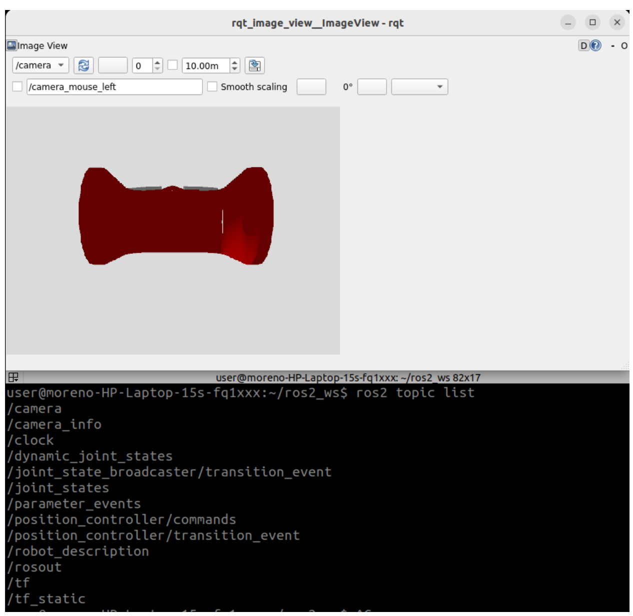


Figure 24: Point of view of the Camera

The brige used to link the camera is given by the following code:

```python
bridge_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',
],
output='screen'
)
```

Figure 25: Bridge to link the camera

4 Design and Implementation of the ROS 2 Control Node

(a) Creating the `armando_controller` package with the specifications given by the homework using the proper command on the terminal:

```
ros2 pkg create --build-type ament-cmake --node-name arm_controler_node --dependencies rclcpp
sensor_msgs std_msgs
```

Once done, we modified the `CMakeLists.txt` and the `package.xml` files in order to compile the node:

```cmake
cmake_minimum_required(VERSION 3.8)
project(armando_controller)

if(CMAKE_COMPILER_IS_GNUCXX OR CMAKE_CXX_COMPILER_ID MATCHES "Clang")
  add_compile_options(-Wall -Wextra -Wpedantic)
endif()

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

set(dependencies
  rclcpp
  std_msgs
  sensor_msgs
)

add_executable(arm_controller_node src/arm_controller_node.cpp)
ament_target_dependencies(arm_controller_node rclcpp sensor_msgs std_msgs)

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

ament_package()
```

```xml
<?xml version="1.0"?>
<?xml-model href="http://download.ros.org/schema/package_format3.xsd" schematypens="http://www.w3.org/2001/XMLSchema"?>
<package format="3">
  <name>armando_controller</name>
  <version>0.0.0</version>
  <description>TODO: Package description</description>
  <maintainer email="user@todo.todo">user</maintainer>
  <license>TODO: License declaration</license>

  <buildtool_depend>ament_cmake</buildtool_depend>

  <depend>rclcpp</depend>
  <depend>sensor_msgs</depend>
  <depend>std_msgs</depend>

  <test_depend>ament_lint_auto</test_depend>
  <test_depend>ament_lint_common</test_depend>

  <export>
    <build_type>ament_cmake</build_type>
  </export>
</package>
```

(b) We created a subscriber to the topic `joint_states` using the message type `sensor_msgs/msg/JointState`. This subscriber is responsible for receiving the current joint positions of the robot and printing them in the terminal each time a new message is published on the topic.

The subscriber was defined as follows:

```cpp
joint_state_sub_ = this->create_subscription<sensor_msgs::msg::JointState>(
```

```
    "joint_states",
    10,
    std::bind(&ArmControllerNode::jointStateCallback, this, _1));
```

The associated callback function, `jointStateCallback()`, iterates through all joint positions and prints their values:

```
for (size_t i = 0; i < msg->position.size(); ++i) {
RCLCPP_INFO(this->get_logger(), "  joint[%zu]: %.3f", i, msg->position[i]);
}
```

In this way, every time a new joint state message is received, the current position of each joint is displayed in the console, allowing real-time monitoring of the robot's motion state.

(c) We defined a publisher named `position_comm_pub` that publishes sequential joint position commands to the topic `/position_controller/commands`. This topic uses the message type `std_msgs/msg/Float64MultiArray`, which allows sending an array of target joint positions to the robot's position controller.

The publisher was created as follows:

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

The predefined sequence of commands was defined in a vector structure:

```
command_sequence_ = {
    {0.0, 0.0, 0.2, 0.0},
    {0.0, 0.3, 0.2, 0.0},
    {0.0, 0.0, 0.0, 0.0},
    {0.0, 0.0, -0.2, 0.0}
};
```

Each command is published through the function `publishCommand()`, which sends the next element of the sequence to the controller:

```
std_msgs::msg::Float64MultiArray msg;
msg.data = command_sequence_[command_index_];
position_comm_pub_->publish(msg);
```

(d) At first, let's modify `armando_controller.cpp` in order to allow the node to publish on the right topic and with the corresponding type of data according to the user's controller choice.

```cpp
    void publishCommand()
    {
        if(choice_controll_ == "0"){
            std_msgs::msg::Float64MultiArray msg;
            msg.data = command_sequence_[command_index_];
            position_comm_pub_->publish(msg);

            RCLCPP_INFO(this->get_logger(),
                    "Published command #%zu: [%.2f, %.2f, %.2f, %.2f]",
                    command_index_ + 1,
                    msg.data[0], msg.data[1], msg.data[2], msg.data[3]);
        }else{
            trajectory_msgs::msg::JointTrajectory msg1;
        msg1.joint_names = {"j0", "j1", "j2", "j3"};

        trajectory_msgs::msg::JointTrajectoryPoint point;
        point.positions = command_sequence_[command_index_];
        point.velocities = command_sequence_vel_[command_index_];
        point.time_from_start = rclcpp::Duration::from_seconds(2.0);  // esempio: 2 secondi per raggiungere la posizione
        msg1.points.push_back(point);

        trajectory_comm_pub_->publish(msg1);

        RCLCPP_INFO(this->get_logger(),
                    "Published command #%zu: [%.2f, %.2f, %.2f, %.2f, Velocities: %.2f, %.2f, %.2f, %.2f]",
                    command_index_ + 1,
                    point.positions[0], point.positions[1], point.positions[2], point.positions[3],
                    point.velocities[0], point.velocities[1], point.velocities[2], point.velocities[3]);
        }
    }

    std::string choice_controll_;
    rclcpp::Subscription<sensor_msgs::msg::JointState>::SharedPtr joint_state_sub_;
    rclcpp::Publisher<std_msgs::msg::Float64MultiArray>::SharedPtr position_comm_pub_;
    rclcpp::Publisher<trajectory_msgs::msg::JointTrajectory>::SharedPtr trajectory_comm_pub_;
    std::vector<std::vector<double>> command_sequence_;
    std::vector<std::vector<double>> command_sequence_vel_;
    size_t command_index_;
    rclcpp::TimerBase::SharedPtr timer_;
    bool goal_reached_;
};
```

Figure 26: Structure of the publishCommand after modification.

```cpp
        if(choice_controll_ == "0"){
            position_comm_pub_ = this->create_publisher<std_msgs::msg::Float64MultiArray>(
            "/position_controller/commands",
            10);
        }else{
            trajectory_comm_pub_ = this->create_publisher<trajectory_msgs::msg::JointTrajectory>(
            "/joint_trajectory_controller/joint_trajectory",
            10);
        }

        command_sequence_ = {
            {0.0, 0.0, 0.2, 0.0},
            {0.0, 0.3, 0.2, 0.0},
            {0.0, 0.0, 0.0, 0.0},
            {0.0, 0.0, -0.2, 0.0}
        };

        command_sequence_vel_ = {
            {0.0, 0.0, 0.0, 0.0},
            {0.0, 0.0, 0.0, 0.0},
            {0.0, 0.0, 0.0, 0.0},
            {0.0, 0.0, 0.0, 0.0}
        };
```

Figure 27: ArmControllerNode constructor editings

In order to use the `joint_trajectory_controller`, it must be loaded and configured in `armando_world.launch.py`, and its configuration must be added in `armando_controllers.yaml`.

armando_description > config > ! armando_controllers.yaml

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

Figure 28: `armando_controllers.yaml` after `joint_trajectory_controller` configuration adding

```
position_controller_spawner = Node(
package="controller_manager",
executable="spawner",
arguments=["position_controller", "--controller-manager", "/controller_manager"],
condition=IfCondition(PythonExpression([choice_controll, ' == 0'])),
)

joint_trajectory_controller_spawner = Node(
package="controller_manager",
executable="spawner",
arguments=["joint_trajectory_controller", "--controller-manager", "/controller_manager"],
condition=IfCondition(PythonExpression([choice_controll, ' == 1'])),
)
```

Figure 29: editings in the controllers loadings to add `joint_trajectory_controller` and make they dependet on `choice_control` parameter in `armando_world.launch.py` .
In the

As with the other two controllers, the `joint_trajectory_controller` also waits for the robot to spawn in Gazebo in the same fashion. Below is the definition of the argument `choice_control_arg`, which allows the user to choose the controller from the command line.

```
choice_controll_arg = DeclareLaunchArgument(
name='choice_controll',
default_value='0',
description='Choose which controller to use (0=position, 1=trajectory)'
)

choice_controll = LaunchConfiguration('choice_controll')
```

Figure 30: Definition of the argument `choice_control_arg` in `armando_world.launch.py`.

Always in `armando_world.launch.py`, the `armando_controller.cpp` is launched by the same delay function as the other controllers, of course after them, passing to it the `choice_control` argument.

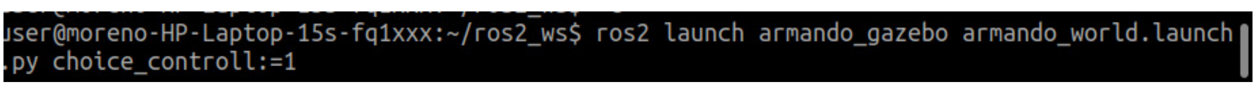
```
arm_controller_node_spawner = Node(
package="armando_controller",
executable="arm_controller_node",
name="arm_controller_node",
arguments=[choice_controll],
)

#Launch the ros2 controllers after the model spawns in Gazebo
delay_joint_pos_controller = (
    RegisterEventHandler(
        event_handler=OnProcessExit(
            target_action=urdf_spawner_node,
            on_exit=[
                position_controller_spawner,
                joint_trajectory_controller_spawner,
                arm_controller_node_spawner,
                ],
        )
    )
)
```

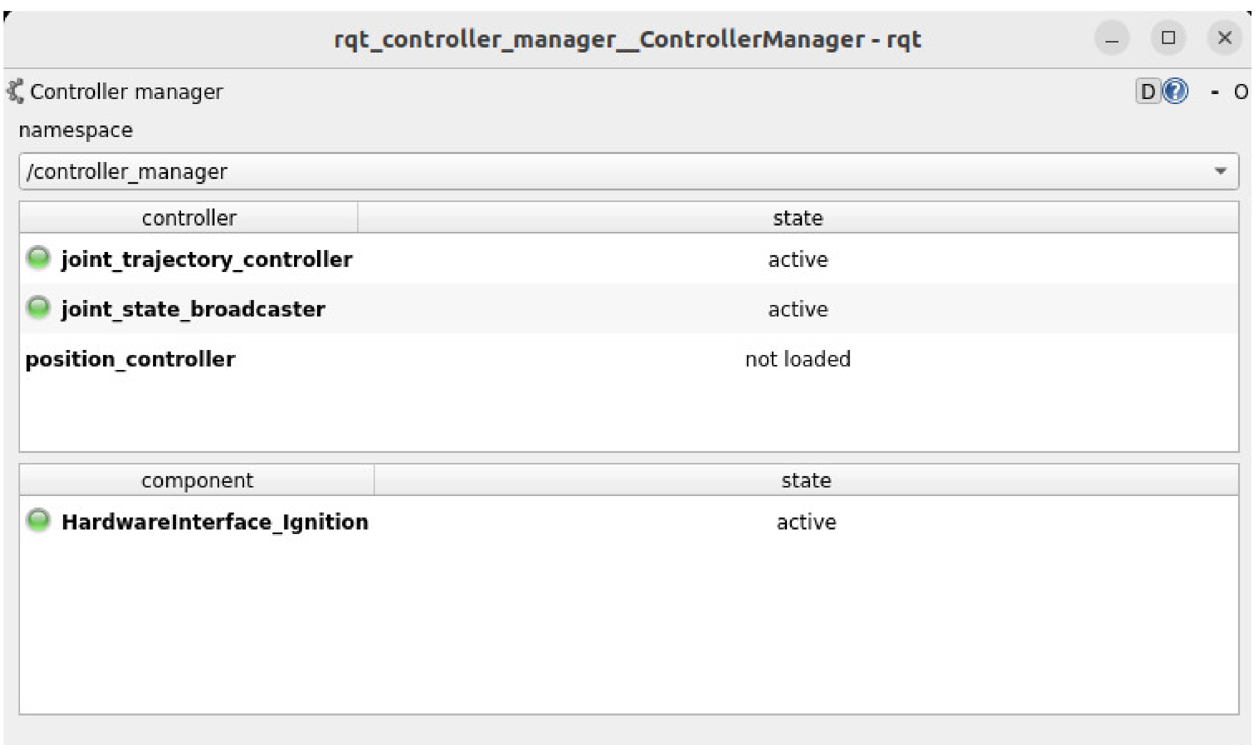
Figure 31: `armando_controller.cpp` launch

If the user selects `choice_control` as 0, the `position_controller_spawner` will be launched; if it is 1, the `joint_trajectory_controller_spawner` will be launched.

```
user@moreno-HP-Laptop-15s-fq1xxx:~/ros2_ws$ ros2 launch armando_gazebo armando_world.launch
.py choice_controll:=0
```


rqt_controller_manager__ControllerManager - rqt
Controller manager
namespace
/controller_manager
controller
state
position_controller
active
joint_state_broadcaster
active
joint_trajectory_controller
not loaded
component
state
HardwareInterface_Ignition
active


user@moreno-HP-Laptop-15s-fq1xxx:~/ros2_ws$ ros2 launch armando_gazebo armando_world.launch
.py choice_controll:=1


rqt_controller_manager__ControllerManager - rqt
Controller manager
namespace
/controller_manager
controller
state
joint_trajectory_controller
active
joint_state_broadcaster
active
position_controller
not loaded
component
state
HardwareInterface_Ignition
active