Robotics Lab: Homework 1 Report

a.y. 2024/2025

Students

Annese Antonio	P38000296
Bosco Stefano	P38000245
Ercolanese Luciana	P38000197
Varone Emanuela	P38000284

Contents

Goal

This document contains a report of Homework 1 of the Robotics Lab class. The goal of this homework is to build ROS packages to simulate a 4-degrees-of-freedom robotic manipulator arm within the Gazebo environment.

1 Robot description and Rviz

In this section, we refer to several key components of ROS2, including launch files, URDF and Rviz. Brief descriptions of these elements are provided below:

- **Launch Files:** allow multiple nodes to start with a single command.
- **Rviz:** is a 3D visualization tool for ROS.
- **URDF:** defines an XML format for representing a robot model.

1.0.1 1.a arm_description package

First of all, we need to download the `arm_description` package from the repository at the following link: https://github.com/RoboticsLab2024/arm_description.git into our `ros2_ws`, using git commands, as seen in Fig. 1.

```
emanuela@emanuela-MacBookAir: ~/HW_RL_1
emanuela@emanuela-MacBookAir:~/HW_RL_1$ git clone  https://github.com/RoboticsLab2024/arm_description.git
```

Figure 1: Cloning arm_description in the terminal

1.0.2 1.b Create launchfile display.launch.py

Within the package we create a `launch` folder, containing a launch file, named `display.launch`, that loads the URDF as a `robot_description` ROS parameter, as seen in Fig.2:

```python
arm_path = get_package_share_directory('arm_description')

arm_urdf = os.path.join(arm_path, "urdf", "arm.urdf")

with open(arm_urdf, 'r') as info:
    arm_desc = info.read()

robot_arm_description = {"robot_description": arm_desc}
```

Figure 2: Including urdf

To start the `robot_state_publisher` node, the `joint_state_publisher` node and the `rviz2` node, we proceed as seen in Fig.3:

```python
joint_state_publisher_node = Node(
    package="joint_state_publisher_gui",
    executable="joint_state_publisher_gui",
)

robot_state_publisher_node_links = Node(
    package="robot_state_publisher", #ros2 run robot_state_publisher robot_state_publisher
    executable="robot_state_publisher",
    output="both",
    parameters=[robot_arm_description,
                {"use_sim_time": True},
        ],
    remappings=[('/robot_description', '/robot_description')]
)


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

nodes_to_start = [
    joint_state_publisher_node,
    robot_state_publisher_node_links,
    rviz_node
]

return LaunchDescription(declared_arguments + nodes_to_start)
```

Figure 3: Start nodes

To launch the file `display.launch`, we use the command:

```
user@emanuela-MacBookAir: ~/ros2_ws 129x18
user@emanuela-MacBookAir:~/ros2_ws$ ros2 launch arm_description display.launch.py
```

Figure 4: Command to launch launch.display.py

The visualization of our robot in Rviz appears as seen in Fig.5:

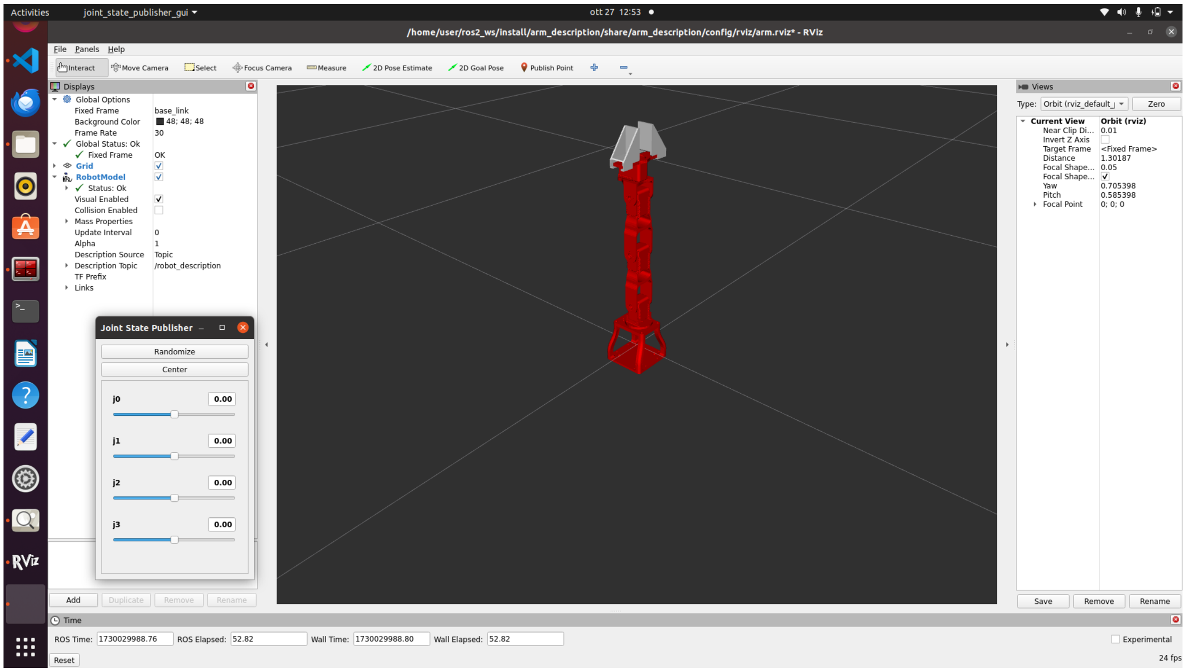
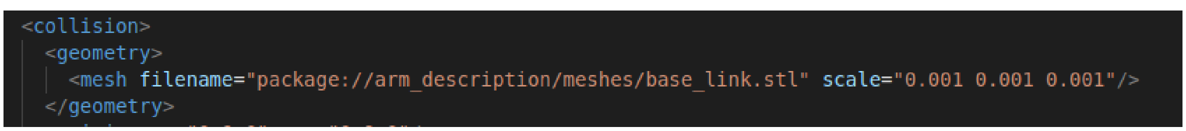


Figure 5: Robot visualization in Rviz

We saved the `arm.rviz` configuration file, that automatically loads the RobotModel plugin by default, inside the `config/rviz` directory.

1.0.3 1.c Substitution of the collision meshes with box

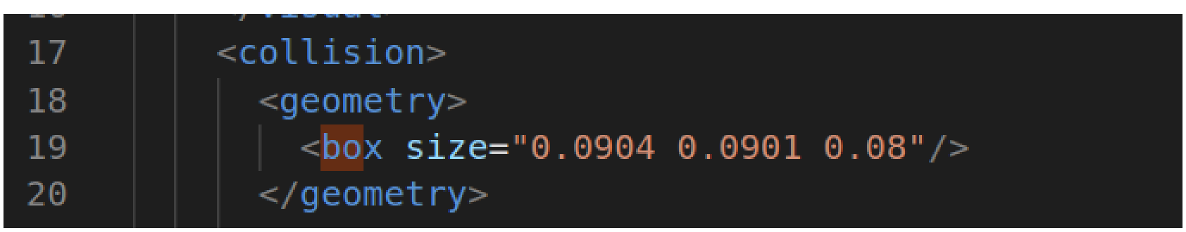
To substitute the collision meshes of our URDF with primitive shapes, we use `<box>` geometries of reasonable size approximating the links.
For example, for the base_link, we had before:



```
<collision>
  <geometry>
    <mesh filename="package://arm_description/meshes/base_link.stl" scale="0.001 0.001 0.001"/>
  </geometry>
```

Figure 6: Collision geometry before edit

After the edit, we have:



```
<collision>
  <geometry>
    <box size="0.0904 0.0901 0.08"/>
  </geometry>
```

Figure 7: Collision geometry after edit

By enabling the collision visualization in Rviz, we see:

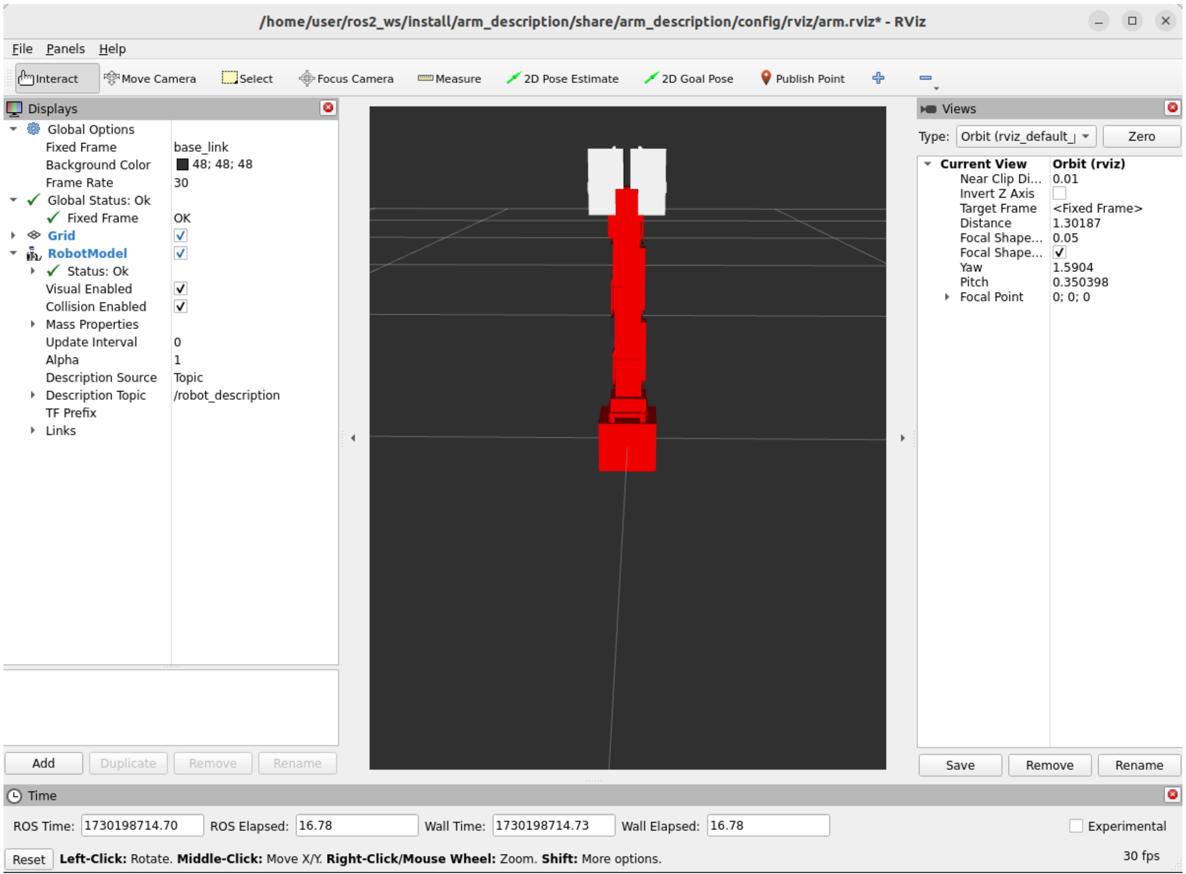


Figure 8: Robot visualization in Rviz with collision geometry

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

In this chapter we will create a ROS 2 package called `arm_gazebo` to simulate a robot in Gazebo, configuring the launch files needed to load the URDF model and manage the joint controllers.

2.0.1 2.a Creation of arm_gazebo package

To create the `arm_gazebo`, we use the following command:

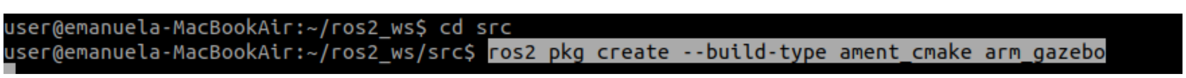


Figure 9: ros2 pkg create

2.0.2 2.b Creation of arm_world.launch file

Within the `arm_gazebo` package, we create a launch folder, containing an `arm_world.launch.py` file, as seen in Fig.10 and Fig.11:

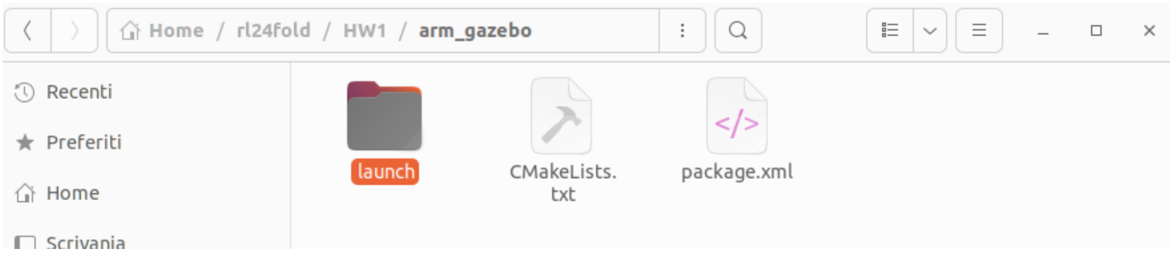


Figure 10: Launch folder

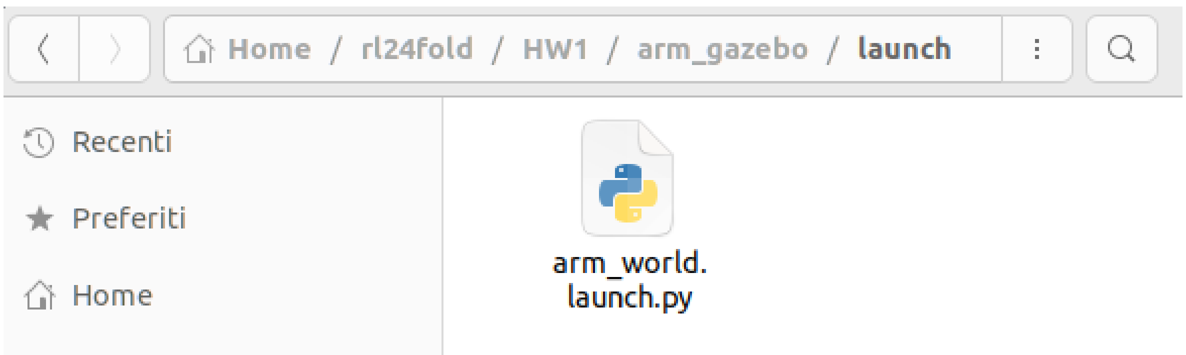


Figure 11: arm_world.launch.py file

2.0.3 2.c Spawn the robot in Gazebo

To load the URDF into the `/robot_description` topic, we proceeded in the following way (Fig.12):

```python
arm_path = get_package_share_directory('arm_description')

arm_urdf = os.path.join(arm_path, "urdf", "arm.urdf")

with open(arm_urdf, 'r') as info:
    arm_desc = info.read()

robot_arm_description = {"robot_description": arm_desc}

joint_state_publisher_node = Node(
    package="joint_state_publisher_gui",
    executable="joint_state_publisher_gui",
)

robot_state_publisher_node_links = Node(
    package="robot_state_publisher", #ros2 run robot_state_publisher robot_state_publisher
    executable="robot_state_publisher",
    output="both",
    parameters=[robot_arm_description],
)
```

Figure 12: Loading the URDF into the /robot_description topic

Then, we spawn the robot using the create node in the `ros_gz_sim` package, as seen in Fig.13:

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

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


gz_spawn_entity = Node(
    package='ros_gz_sim',
    executable='create',
    output='screen',
    arguments=['-topic', 'robot_description',
               '-name', 'arm',
               '-allow_renaming', 'true',],
)

ign = [gazebo_ignition, gz_spawn_entity]


nodes = [
    *ign,
    robot_state_pub_node,
]

return LaunchDescription(declared_arguments + nodes)
```

Figure 13: Spawn the robot model

With the command `ros2 launch arm_gazebo arm_world.launch.py` we launch the `arm_world.launch.py` file to visualize the robot in Gazebo.

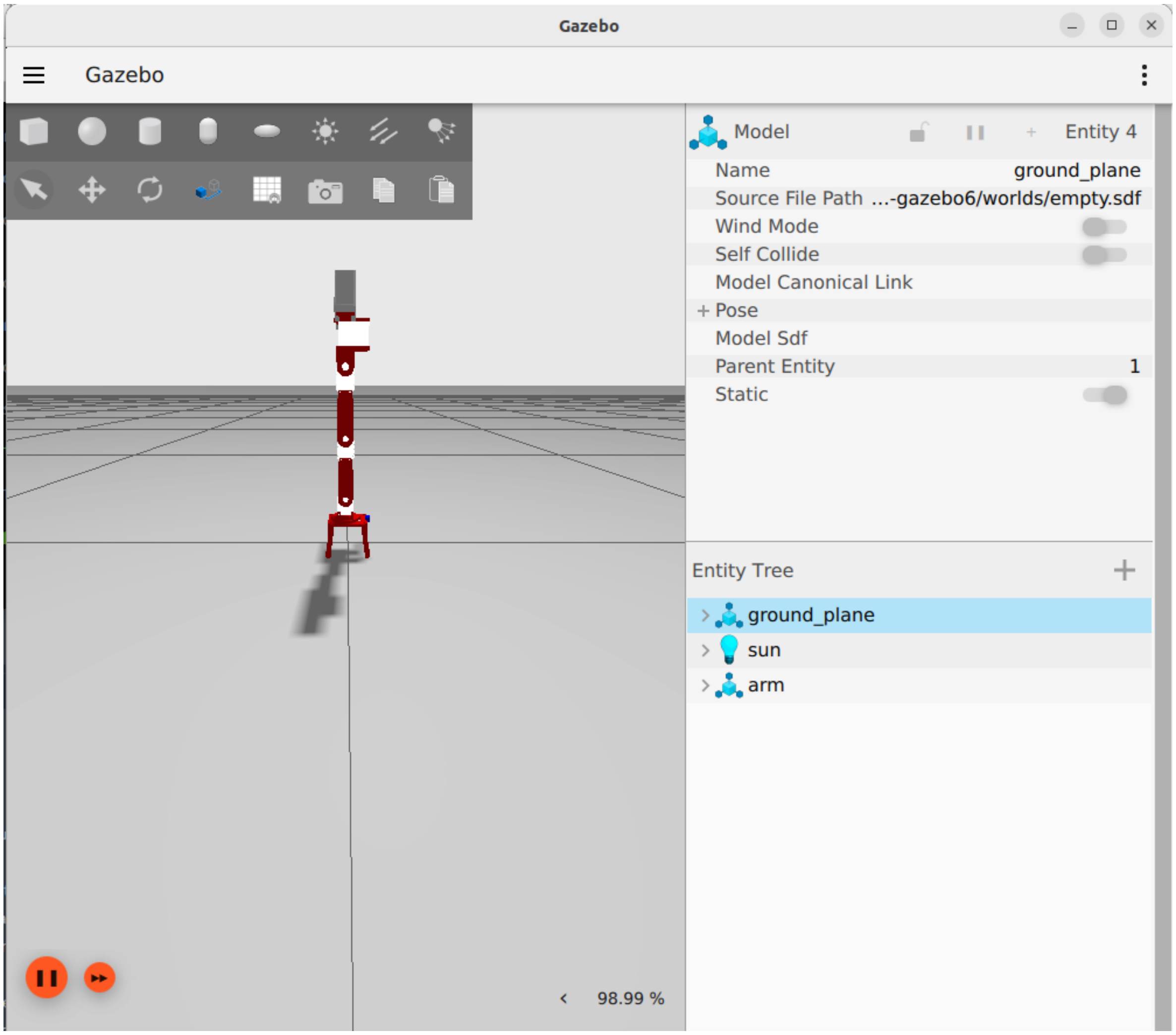


Figure 14: Robot visualization in Gazebo

2.0.4 2.d Add a PositionJointInterface as a hardware interface to the robot using ros2_control

First of all, we rename the URDF file to arm.urdf.xacro, by adding the string:
`xmlns:xacro="http://www.ros.org/wiki/xacro"` within the `<robot>` tag (Fig.15):

```
rl24fold > HW1 > arm_description > urdf > arm.urdf.xacro
<?xml version="1.0"?>
<robot xmlns:xacro="http://www.ros.org/wiki/xacro" name="arm">
```

Figure 15: Renaming the URDF file to arm.urdf.xacro

Then, we load the URDF in our launch file using the xacro routine (Fig.16):

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

Figure 16: Loading the URDF with the xacro routine

In the `arm_description/urdf` folder we create an `arm_hardware_interface.xacro` file, containing a macro that defines the hardware interface for the joints (Fig.17):

rl24fold > HW1 > arm_description > urdf > arm_hardware_interface.xacro

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


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

    <joint name="${name}">
        <command_interface name="position"/>
        <state_interface name="position">
            <param name="initial_value">0.0</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 17: arm_hardware_interface.xacro file

We include the `arm_hardware_interface.xacro` file in the `arm.urdf.xacro` as follows (Fig.18):

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


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

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

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


</ros2_control>
```

Figure 18: including the arm_hardware_interface.xacro file in the arm.urdf.xacro

2.0.5 2.e Load the joint controller configurations and spawn the controllers

First, we create an `arm_control.yaml` file within `arm_control/config` folder (Fig.19) :

```
arm_controllers.yaml M ×
HW_RL_1 > HW1 > arm_control > config > arm_controllers.yaml
controller_manager:
  ros__parameters:
    update_rate: 225  # Hz


    joint_state_broadcaster:
      type: joint_state_broadcaster/JointStateBroadcaster

    position_controller:
      type: position_controllers/JointGroupPositionController


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


```

Figure 19: arm_control.yaml file

Then, we spawn the controllers using the `controller_manager` package (Fig.20 and Fig.21):

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

Figure 20: Spawn the controllers

REMARKS: In the figure Fig.21 `TimerAction()` creates a timer that starts the `joint_state_broadcaster` and `position_controller` nodes every 2 seconds, constantly updating the state of the joints via the `controller_manager`.

```
rl24fold > HW_RL_1 > HW1 > arm_control > launch > arm_control.launch.py > generate_launch_description
from launch.event_handlers import OnProcessExit
from launch.actions import TimerAction


def generate_launch_description():
    declared_arguments = []


    joint_state_broadcaster = TimerAction (
        period=2.0,
        actions=[
        Node(
            package="controller_manager",
            executable="spawner",
            arguments=["joint_state_broadcaster", "--controller-manager", "/controller_manager"],
        )
        ]
    )

    position_controller = TimerAction (
        period=2.0,
        actions=[
        Node(
            package="controller_manager",
            executable="spawner",
            arguments=["position_controller", "--controller-manager", "/controller_manager"],
        )
        ]
    )

    nodes_to_start = [
        joint_state_broadcaster,
        position_controller
    ]

    return LaunchDescription(declared_arguments + nodes_to_start)
```

Figure 21: Spawn the controllers

After launching the robot simulation in Gazebo, the hardware interface appears correctly loaded and connected as seen in (Fig.22):

```
[ruby $(which ign) gazebo-1] [INFO] [1730127666.021914628] [resource_manager]: Initialize hardware 'HardwareInterface_Ignition'
[ruby $(which ign) gazebo-1] [WARN] [1730127666.021934129] [gz_ros2_control]: On init...
[ruby $(which ign) gazebo-1] [INFO] [1730127666.022009681] [resource_manager]: Successful initialization of hardware 'HardwareInterface_Ignition'
[ruby $(which ign) gazebo-1] [INFO] [1730127666.022049698] [resource_manager]: 'configure' hardware 'HardwareInterface_Ignition'
[ruby $(which ign) gazebo-1] [INFO] [1730127666.022055174] [gz_ros2_control]: System Successfully configured!
[ruby $(which ign) gazebo-1] [INFO] [1730127666.022066469] [resource_manager]: Successful 'configure' of hardware 'HardwareInterface_Ignition'
[ruby $(which ign) gazebo-1] [INFO] [1730127666.022072813] [resource_manager]: 'activate' hardware 'HardwareInterface_Ignition'
[ruby $(which ign) gazebo-1] [INFO] [1730127666.022077213] [resource_manager]: Successful 'activate' of hardware 'HardwareInterface_Ignition'
```

Figure 22: Hardware interface loaded

2.0.6 2.f Creation of the arm_control package

We create the `arm_control` package from terminal by using the command `ros2 pkg create arm_control` and we create the subfolders and files as seen in Fig. 23

```
HW_RL_1 / HW1
  arm_control
    config
      arm_controllers.yaml
    launch
      arm_control.launch.py
    CMakeLists.txt
    package.xml
```

Figure 23: arm_control package

2.0.7 2.g Defining the controllers in the arm_controllers.yaml file

We add the `joint_state_broadcaster` and joint position controllers for each joint as seen in Fig. 19.

2.0.8 2.h Spawning the robot and controllers using arm_gazebo.launch

We create a single launch file called `arm_gazebo.launch` that calls both `arm_world.launch` and `arm_control.launch`, as seen in Fig. 24.

```
rl24fold > HW_RL_1 > HW1 > arm_gazebo > launch > arm_gazebo.launch.py > ...
from launch import LaunchDescription
from launch.actions import IncludeLaunchDescription
from launch.launch_description_sources import PythonLaunchDescriptionSource
from launch.substitutions import PathJoinSubstitution
from launch_ros.substitutions import FindPackageShare




def generate_launch_description():


    other_launch_file_arm_world = PathJoinSubstitution(
        [FindPackageShare("arm_gazebo"), "launch", "arm_world.launch.py"]
    )


    include_other_launch_arm_world= IncludeLaunchDescription(
        PythonLaunchDescriptionSource(other_launch_file_arm_world)
    )

    other_launch_file_arm_control= PathJoinSubstitution(
        [FindPackageShare("arm_control"), "launch", "arm_control.launch.py"]
    )


    include_other_launch_arm_control = IncludeLaunchDescription(
        PythonLaunchDescriptionSource(other_launch_file_arm_control)
    )


    return LaunchDescription([
        include_other_launch_arm_world,
        include_other_launch_arm_control

    ])
```

Figure 24: arm_gazebo.launch.py file

The attached video ("video_robot_punto2.webm") shows that the robot and controllers are correctly loaded. We publish the target joint position $data : [0.7, 0.0, 0.4, 0.67]$ onto the `/position_controllers/commands` topic and we show that the joints reach the target by echoing the `/joint_states` topic.

3 Add a camera sensor to the robot

3.0.1 3.a camera_link and camera_joint

Into the arm.urdf.xacro file we add a camera_link and a fixed camera_joint with base_link as a parent link (Fig. 25):

```
<joint name="camera_joint" type="fixed">
  <parent link="base_link"/>
  <child link="camera_link"/>
  <origin xyz="0.001 -0.04 0.055" rpy="0.0 0.0 -1.57"/>
</joint>

<link name="camera_link">
  <visual>
    <geometry>
      <box size="0.01 0.01 0.01"/>
    </geometry>
      <material name="blue">
        <color rgba="0 0 1 1"/>
      </material>
  </visual>
</link>
```

Figure 25: camera_link and camera_joint

The camera is positioned with an offset of $xyz = "0.001 - 0.040.055"$, placing it slightly above and to the left of the joint's origin. The orientation is defined by $rpy = "0.00.0 - 1.57"$, indicating that the camera is tilted at -1.57 radians (or -90 degrees). The camera is designed to point forward relative to the robot, allowing it to have a clear view of the workspace and the objects to be manipulated. This configuration is ideal for controlling the robot's interaction with the surrounding environment.

3.0.2 3.b arm_camera.xacro

We create an `arm_camera.xacro` file in the `arm_gazebo/urdf` folder, containing the gazebo sensor reference tags and the `gz-sim-sensors-system` plugin (Fig. 26):

rl24fold > HW_RL_1 > HW1 > arm_description > urdf > arm_camera.xacro

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

<xacro:macro name="my_camera" >

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

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

</xacro:macro>

</robot>
```

Figure 26: arm_camera.xacro file

Then, we import the `arm_camera.xacro` file in `arm.urdf.xacro` using `xacro:include`, as follows (Fig. 27):

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

Figure 27: xacro:include

3.0.3 3.c Launching Gazebo simulation and verifying image topic publishing in rqt_image_view

We add the `ros_ign_bridge` in the `arm_world.launch.py` (Fig. 28).This step is essential because `ros_ign_bridge` provides a network bridge which enables the exchange of messages between ROS 2 and Gazebo.

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

nodes = [
    *ign,
    robot_state_pub_node,
    bridge_camera

]

return LaunchDescription(declared_arguments + nodes)
```

Figure 28: ros_ign_bridge

We start the Gazebo simulation using `ros2 launch arm_gazebo arm_gazebo.launch.py` command. With `rqt_image_view` we can notice that the image topic has been published correctly (Fig. 29):

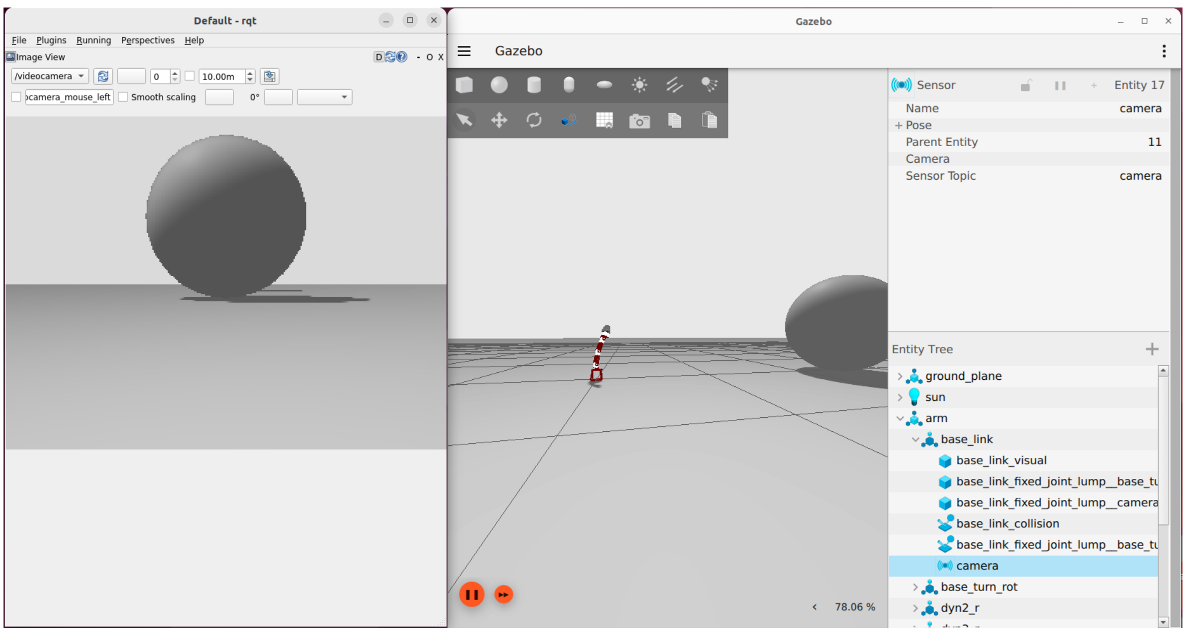


Figure 29: /camera

With 'ros2 topic list' command we see that, among all the topics currently active in the system, there are /camera_info and /videocamera (Fig. 30):

```
user@stebosco-Aspire-A515-56G:~/ros2_ws$ ros2 topic list
/camera_info
/clock
/joint_states
/parameter_events
/robot_description
/rosout
/tf
/tf_static
/videocamera
user@stebosco-Aspire-A515-56G:~/ros2_ws$ ros2 topic list -t
/camera_info [sensor_msgs/msg/CameraInfo]
/clock [rosgraph_msgs/msg/Clock]
/joint_states [sensor_msgs/msg/JointState]
/parameter_events [rcl_interfaces/msg/ParameterEvent]
/robot_description [std_msgs/msg/String]
/rosout [rcl_interfaces/msg/Log]
/tf [tf2_msgs/msg/TFMessage]
/tf_static [tf2_msgs/msg/TFMessage]
/videocamera [sensor_msgs/msg/Image]
```

Figure 30: ros2 topic list

In Fig.30, '/videocamera' is the name of the topic from which we want copy data and 'sensor_msgs/msg/Image' is the message type that will be published on the Ros topic.

The 'rqt_grap' command opens a graphical tool that shows a visual representation of the network of nodes and topics active in the system.In the rqt_graph (Fig. 31), the ros_gz_bridge node is connected to the /videocamera and /camera_info topics by bidirectional arrows.The bidirectional arrows indicate that the ros_gz_bridge node is both publishing and subscribing to the /videocamera and /camera_info topics.This means that ros_gz_bridge receives messages from Gazebo related to the camera and publishes them in ROS, and viceversa.

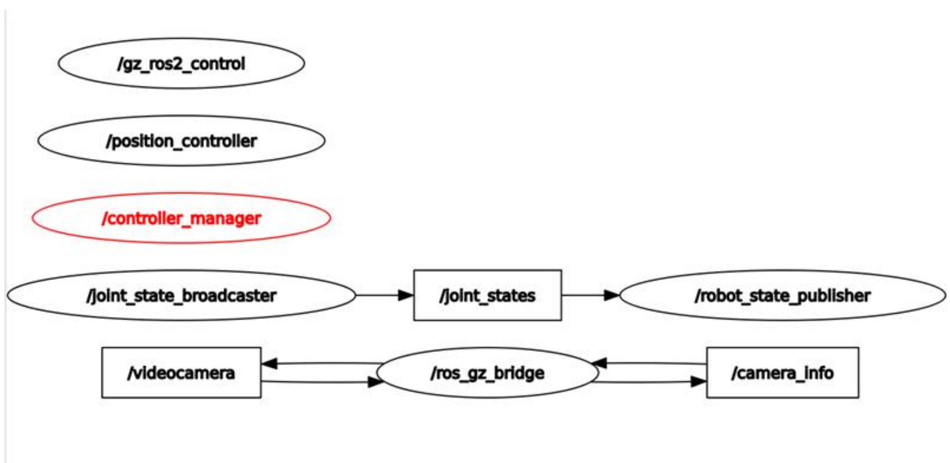


Figure 31: rqt_graph

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

4.0.1 4.a Creating the node with its dependencies specified in CMakeLists

In order to compile the arm_controller_node, we need to add the libraries it depends on and create an executable, by modifying the CMakeLists.txt of its package. The edits are seen in Fig.32.

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

# find_package(<dependency> REQUIRED)
add_executable(arm_controller_node src/arm_controller_node.cpp)

ament_target_dependencies(arm_controller_node rclcpp std_msgs sensor_msgs)
```

(a)

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

(b)

Figure 32: CMakeLists.txt in arm_controller

4.0.2 4.b Writing the subscriber

We define the subscriber by calling the function `create_subscription()`. The node subscribes to the topic */joint_states* and we specify the type of message `<sensor_msgs::msg::JointState>`. The callback function `topic_callback()` is shown in Fig.34 and it prints the current joint states as info in the terminal.

4.0.3 4.c Writing the publisher

We define the publisher by calling the function `create_publisher()`. It publishes a `<std_msgs::msg::Float64MultiArray>` type message onto the topic */position_controllers/commands*. The message is included as an argument and stored in the variable *joint_positions_*.The callback function of the publisher is `publish_command()`.
In the attached video ("video_robot_punto4.webm") we show that the node publishes the desired command "$joint\_positions := [0.5, 0.2, 0.0, -0.5]$" onto the topic and the manipulator reaches the target position, as confirmed by the joint states printed in the terminal.

```cpp
class Arm_controller_node : public rclcpp::Node
{
public:
  Arm_controller_node()
  : Node("arm_controller_node")
  {
    this->declare_parameter<std::vector<double>>("joint_positions", std::vector<double>{0.0, 0.0, 0.0, 0.0});

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

    timer_ = this->create_wall_timer(
      500ms, std::bind(&Arm_controller_node::publish_command, this));

    // Inizializza il sottoscrittore per il topic "joint_states"
    subscription_ = this->create_subscription<sensor_msgs::msg::JointState>(
      "joint_states", 10,
      std::bind(&Arm_controller_node::topic_callback, this, std::placeholders::_1));
  }
```

Figure 33: Constructor of Arm_controller_node

```cpp
private:

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

  void publish_command()
  {
    auto message = std_msgs::msg::Float64MultiArray();

    this->get_parameter("joint_positions", joint_positions_);

    message.data = joint_positions_;

    RCLCPP_INFO(this->get_logger(), "Publishing command: [%f, %f, %f, %f]",
                message.data[0], message.data[1], message.data[2], message.data[3]);

    publisher_->publish(message);
  }

  rclcpp::Publisher<std_msgs::msg::Float64MultiArray>::SharedPtr publisher_;
  rclcpp::TimerBase::SharedPtr timer_;
  rclcpp::Subscription<sensor_msgs::msg::JointState>::SharedPtr subscription_;
  std::vector<double> joint_positions_;
};
```

Figure 34: Function callbacks for the publisher and subscriber

5 GitHub repositories links:

Students

Annese Antonio	https://github.com/antann1/HomeworkRL_1.git
Bosco Stefano	https://github.com/SteBosco/HMW1_RL_2024.git
Ercolanese Luciana	https://github.com/LErcolanese/RL2024_HW1.git
Varone Emanuela	https://github.com/Emanuela-var/Homework1_RL.git