Università degli Studi di Napoli Federico II

Scuola Politecnica e delle Scienze di Base

Dipartimento di Ingegneria Elettrica e Tecnologie dell'Informazione

Robotics Lab

Homework 1: Building your robot manipulator

Components
William Notaro
Chiara Panagrosso
Salvatore Piccolo
Roberto Rocco

Anno Accademico 2024–2025

Contents

This assignment's goal is to create and set up ROS packages that use the Gazebo simulation environment to mimic a four-degree-of-freedom robotic manipulator. In order to model, control, and view the robotic arm and create a working simulation setup for testing and experimentation, this entails building and combining a number of components.

At the beginning of each chapter, a brief explanation of its purpose will be provided, outlining the steps that will be followed to complete the task

Chapter 1

Robot Overview and Rviz Visualization

This chapter's goal is to describe the robotic manipulator in detail and use Rviz to visualize it. The `arm_description package`, which provides the foundational files for the robot's model, is first downloaded into the ROS workspace. To load the URDF as a robot_description parameter and launch the required nodes (robot_state_publisher, joint_state_publisher, and rviz2) for visualization, a display.launch file is generated within this package. Rviz parameters, such as the Fixed Frame and RobotModel plugin, are set up to correctly display the robot after this file is launched. Furthermore, box geometries that approach link sizes are used in place of the robot's collision meshes for efficient collision modeling. This allows for a clear display of collision boundaries in Rviz by using the Robot model, with Collision Enabled option. The output of this chapter is a package, named arm_description, which will contain all the information about the manipulator.

1.1 `arm_description` package download

The first step is to download the `arm_description` package, which contains the necessary files for outlining the physical characteristics of the robot.

Inside the `ros2_ws` directory we used:

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

1.2 Configuring the Launch File and Rviz

We created the requested folder with the `mkdir launch` command, and the requested file with `touch launch/display.launch.py`.

Then, the requested nodes (`robot_state_publisher`, `joint_state_publisher`, and `rviz2`) have been created through the `display.launch.py` launch file.

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

def generate_launch_description():
    package_share_directory = get_package_share_directory('arm_description')
    urdf_file = os.path.join(package_share_directory, 'urdf', 'arm.urdf.xacro')
    rviz_config_file = os.path.join(package_share_directory, 'rviz', 'my_config.rviz')

    robot_state_publisher_node = Node(
            package='robot_state_publisher',
            executable='robot_state_publisher',
            name='robot_state_publisher',
            output='screen',
            parameters=[{'robot_description': Command(['xacro ', urdf_file])}]
        )

    rviz_node = Node(
            package='rviz2',
            executable='rviz2',
            name='rviz2',
            output='screen',
            arguments=['-d', rviz_config_file]
        )

    joint_state_publisher_node = Node(
            package='joint_state_publisher_gui',
            executable='joint_state_publisher_gui',
            name='joint_state_publisher',
            output='screen'
        )

    nodes_to_start = [
        robot_state_publisher_node,
        joint_state_publisher_node,
        rviz_node
        ]

    return LaunchDescription(nodes_to_start)
```

Listing 1.1: display.launch.py

Using the `robot_description` argument, this file loads the robot's URDF (Unified Robot Description Format) file.

1.3 Collision Meshes

In order to substitute the predefine collision meshes with box ones, we first delete (or comment) the `mesh` line inside the `collision` tag, e.g.

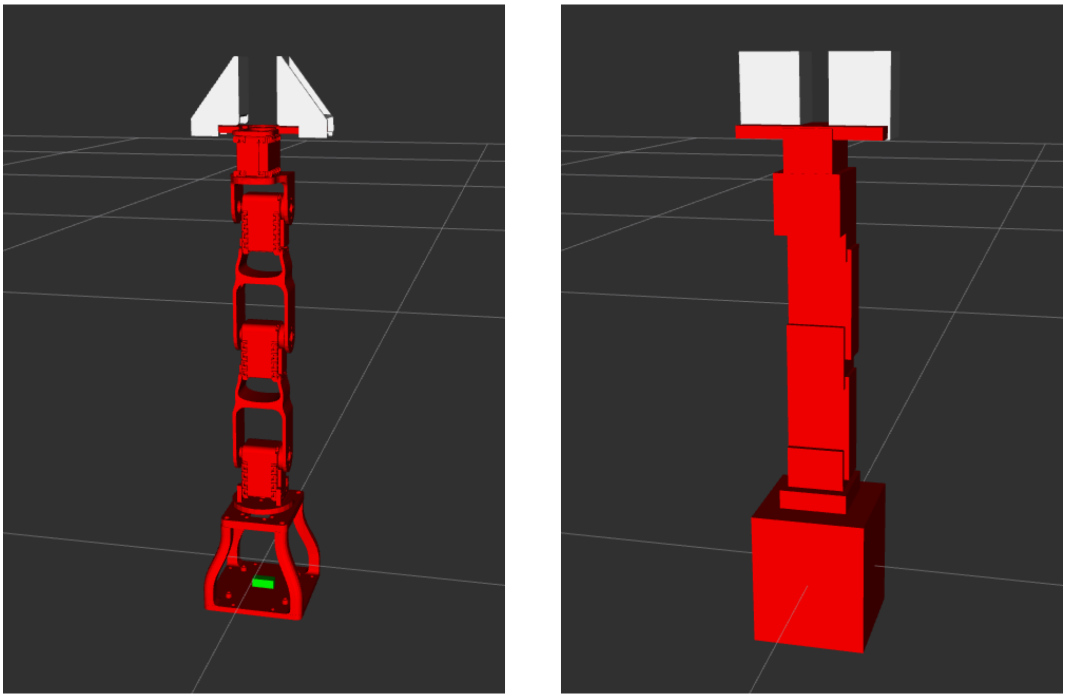
```
<mesh filename="package://arm_description/meshes/base_link.stl" scale="0.001 0.001 0.001"/>
```

Then we create Box geometries for every manipulator's link.

```xml
<collision>
    <geometry>
        <box size="0.1 0.1 0.12"/>
    </geometry>
    <origin rpy="0 0 0" xyz="0 0 0"/>
</collision>
```

Listing 1.2: An example of a box collision geometry

The dimension is chosen via trial and error based on a visual procedure done in Rviz. At the end, the collision space of the robot appears as follows:



(a) Robot original collision mesh (b) Robot approximated collision space

Figure 1.1: Comparison between the exact and approximated robot collision space

Chapter 2

Installing Controllers and Sensors and Gazebo Spawning

The aim of this chapter is to integrate sensors and controls into the robot and simulating its function in Gazebo. Gazebo is a simulation tool that allows developers to create realistic 3D environments for testing robots. It provides features for physics simulation, sensor integration, and visualization, enabling users to simulate robot behavior and interactions with the environment before deploying them in real-world scenarios.

More specifically, we will create an `arm_gazebo` package with a launch file in order to spawn the robot in the Gazebo environment and load its URDF into the `/robot_description` topic. To precisely handle joint positions, we will additionally use `ros2_control` to construct a PositionJointInterface, describe this hardware interface in an `arm_hardware_interface.xacro` file, and set joint controllers using a `.yaml` file.

2.1 Package and Launch File

To create the required package - useful to manage the whole Gazebo environment and spawn the robot inside of it - the following command has been executed:

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

Then, analogously to the previous chapter, a `launch` folder has been created along with the `arm_world.launch` launch file.

The launch file loads the robot's URDF into the `/robot_description` topic and spawns the robot in Gazebo using the node `create` command from the `ros_gz_sim` package.

```python
import os
from launch import LaunchDescription
from launch.substitutions import Command, LaunchConfiguration,
    PathJoinSubstitution
from launch_ros.actions import Node
from launch.actions import DeclareLaunchArgument
from launch.launch_description_sources import
    PythonLaunchDescriptionSource
from launch.actions import IncludeLaunchDescription
from ament_index_python.packages import get_package_share_directory

def generate_launch_description():
    # Get path to URDF file from the arm_description package
    urdf_file = os.path.join(get_package_share_directory('arm_description'
    ), 'urdf', 'arm.urdf.xacro')

    # Argument declaration for using simulation time
```

```python
    use_sim_time = LaunchConfiguration('use_sim_time', default='true')

    # Node for loading robot description
    robot_state_publisher_node = Node(
        package='robot_state_publisher',
        executable='robot_state_publisher',
        output='screen',
        parameters=[{'robot_description': Command(['xacro ', urdf_file])},
                    {"use_sim_time": use_sim_time}]
    )

    # Declare arguments
    declared_arguments = [
        DeclareLaunchArgument(
            'gz_args',
            default_value='-r -v 1 empty.sdf',
            description='Arguments for gz_sim'
        )
    ]
    # Node for starting Gazebo Ignition (gz sim)
    gazebo_ignition = IncludeLaunchDescription(
            PythonLaunchDescriptionSource(
                [PathJoinSubstitution([get_package_share_directory('ros_gz_sim'),
                                    'launch',
                                    'gz_sim.launch.py'])]),
            launch_arguments={'gz_args': LaunchConfiguration('gz_args')}.items()
    )

    # Node for spawning the robot in Gazebo Ignition
    spawn_robot_node = Node(
        package='ros_gz_sim',
        executable='create',
        name='spawn_robot',
        output='screen',
        arguments=[
            '-topic', '/robot_description',
            '-entity', 'robot'
        ]
    )

    # Node for bridging the camera topics from Gazebo Ignition to ROS2
    bridge_camera_node = 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'
    )


    # List of nodes to start
    nodes_to_start = [
```

```
        robot_state_publisher_node,
        gazebo_ignition,
        spawn_robot_node,
        bridge_camera_node
    ]

    return LaunchDescription(declared_arguments + nodes_to_start)
```

Listing 2.1: arm_world.launch.py

The purpose of the launch file is to launch the nodes responsible for managing the robot and its state (`robot_state_publisher`), for creating the Gazebo world (the version used is Gazebo Ignition), and, as we will see in the following points, for all the components responsible for the controllers and the control_manager.

After compiling the package through the `colcon build` command, the launch file is called using:

`ros2 launch arm_gazebo arm_world.launch.py`

It is to be noticed, however, that such command could give arise to the following list of errors:

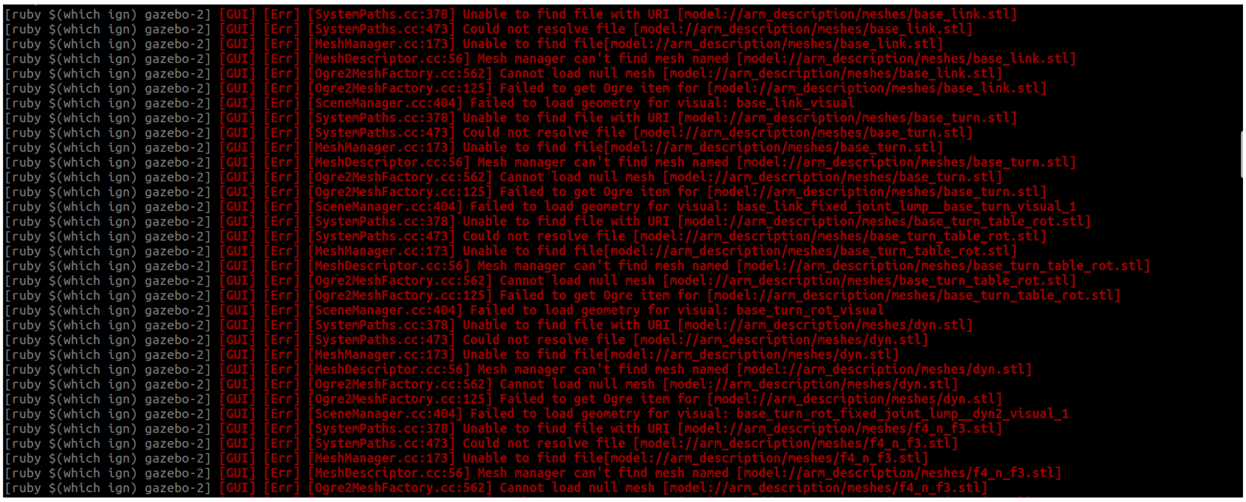


Figure 2.1: Possible list of errors when launching the Gazebo environment

This is due to the fact that the `gazebo` tag inside the `package.xml` file of the `arm_gazebo` package does not update the `GZ_SIM_RESOURCE_PATH` as expected:

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

Listing 2.2: An extract from the `arm_gazebo package.xml`

2.2 Hardware Interface

The hardware interface is defined inside the `arm_hardware_interface.xacro` file in the `arm_description/urdf` folder. In particular, this file contains contains a macro called `PositionJointInterface` that defines the hardware interface for each joint. Such macro is integrated into the main URDF through a `xacro:include` directive and a call inside the `ros2_control` environment.

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

<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>
```

Listing 2.3: arm_hardware_interface.xacro

`ros2_control` is a framework designed to provide a standardized way to control robot hardware. It enables the development of robot control systems by defining a clear interface for controlling various hardware components, such as motors and sensors.

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

...

<ros2_control name="IgnitionSystem" type="system">
  <hardware>
    <plugin>ign_ros2_control/IgnitionSystem</plugin>
  </hardware>

  <xacro:PositionJointInterface name="j0" initial_pos="1.0"/>
  <xacro:PositionJointInterface name="j1" initial_pos="2.0"/>
  <xacro:PositionJointInterface name="j2" initial_pos="-1.0"/>
  <xacro:PositionJointInterface name="j3" initial_pos="-2.0"/>

</ros2_control>
```

Listing 2.4: An extract of the main URDF which includes and calls the hardware interface macro

2.3 Control

The `controller_manager` package is a core component for ROS2 control architecture: it handles the loading, unloading, starting, stopping, and configuration of controllers dynamically at runtime. This allows for flexible control strategies and easy adjustments to the robot's behavior.

The `controller manager` node can be started in two ways: either by a launch file (via command line) or via plugin. Since the whole homework project moves around a simulation environment, we will start it in the latter manner, i.e. through the

`ign_ros2_control::IgnitionROS2ControlPlugin` plugin, called in the URDF file.

Note: a plugin is a shared library, a collection of pre-compiled code and data that can be used by multiple programs simultaneously.

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

Listing 2.5: An extract of the main URDF which calls the plugin that starts the control framework

In order for the controllers to be configured through the `.yaml` files and activated, the respective node of the `controller_manager` must have been declared beforehand. The declarations of the controllers and their configuration is stated in the `controllers.yaml` file, inside the `config` folder of the `arm_control` package.

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

    joint_state_broadcaster:
      type: joint_state_broadcaster/JointStateBroadcaster

    joint_trajectory_controller:
      type: joint_trajectory_controller/JointTrajectoryController

    position_controller:
      type: position_controllers/JointGroupPositionController

position_controller:
  ros__parameters:
    joints:
      - j0
      - j1
      - j2
      - j3
    command_interfaces:
      - position
    state_publish_rate: 100.0       # Hz for state publication rate
    action_monitor_rate: 20.0       # Hz for action monitoring
    allow_partial_joints_goal: true # Allows sending position goals for a
    subset of joints
```

```
    open_loop_control: false         # Set true for open-loop control
```

Listing 2.6: `controller.yaml`

Additionally, the launch of the controllers can only occur once the robot has been spawned in the Gazebo world; otherwise, errors will occur. This is done by another launch file, called `arm_gazebo.launch.py` that will make sure to run sequentially first the `arm_world.launch.py` launch file - responsible for generating the world and spawning the robot in Gazebo - and then the `arm_control.launch.py` - responsible for the creation of the controller nodes.

```python
# arm_gazebo.launch.py

from launch import LaunchDescription
from launch_ros.substitutions import FindPackageShare
from launch.substitutions import PathJoinSubstitution
from launch.launch_description_sources import PythonLaunchDescriptionSource
from launch.actions import IncludeLaunchDescription

def generate_launch_description():

    return LaunchDescription([

        # Include the arm_world.launch.py
        IncludeLaunchDescription(
            PythonLaunchDescriptionSource(
                PathJoinSubstitution([FindPackageShare('arm_gazebo'), 'launch', 'arm_world.launch.py'])
            ),
            launch_arguments={'use_sim_time': 'true'}.items()
        ),

        # Include the arm_control.launch.py to spawn controllers
        IncludeLaunchDescription(
            PythonLaunchDescriptionSource(
                PathJoinSubstitution([FindPackageShare('arm_control'), 'launch', 'arm_control.launch.py'])
            )
        ),

    ])
```

Listing 2.7: arm_gazebo.launch.py

To check that everything is correctly set, we execute the `arm_gazebo.launch.py` launch file. Then, in another console we type two commands for checking that the hardware interfaces and the controllers are correctly loaded.

For the hardware interfaces we type:

```
ros2 control list_hardware_interfaces
```

and we obtain the following output:

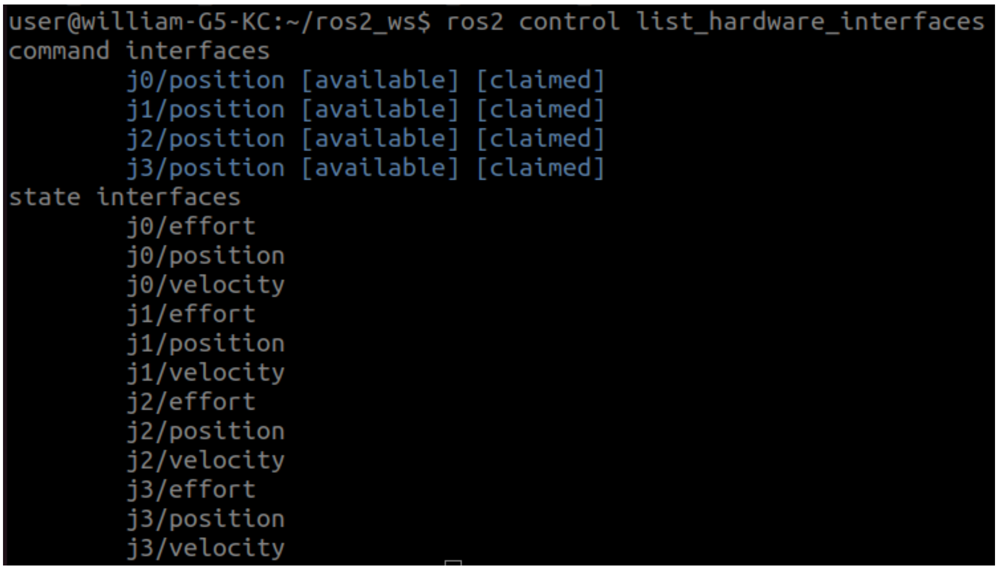


Figure 2.2: Check of the hardware interfaces

For the controllers we type:

```
ros2 control list_controllers
```

and we obtain the following output:

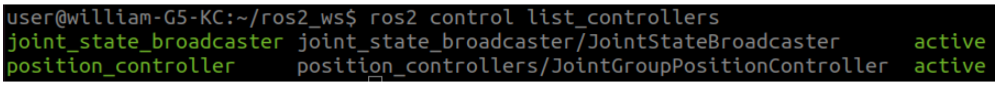


Figure 2.3: Check of controllers

In addition, the Gazebo environment starts showing the world along with the arm manipulator.

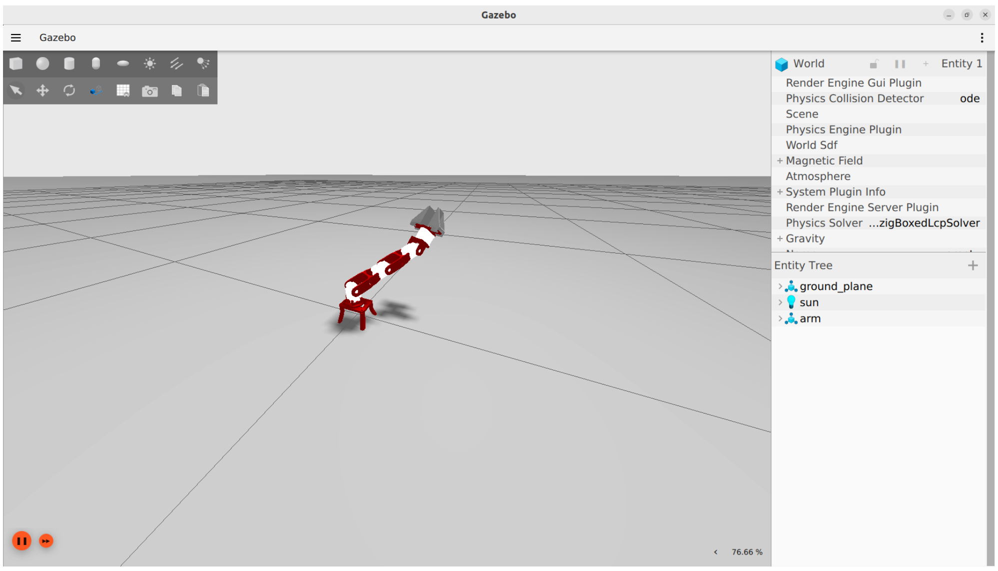


Figure 2.4: The Gazebo simulator

Chapter 3

Camera Sensor Integration

For improved efficiency in the Gazebo simulation environment, incorporate a camera sensor into your robot model. This entails adding a camera_link and a fixed camera_joint that are attached to the robot's base_link by altering the `arm.urdf.xacro` file. You must also make an `arm_camera.xacro` file, in which the Gazebo sensor plugin and pertinent camera parameters are specified. To make sure the camera stream is functioning correctly in the simulated environment, the camera configuration should be set up to publish image data, which can then be confirmed using tools like `rqt_image_view`.

3.1 Add camera to robot configuration

Let us begin by modifying the URDF file that describes the robot's configuration; within it, we need to add the camera sensor inside an appropriate <gazebo> tag.

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

<link name="camera_link">
  <visual>
    <geometry>
      <box size="0.02 0.008 0.008"/>
    </geometry>
    <origin rpy="0 0 0" xyz="0 0 -0.022"/>
    <material name="green"/>
  </visual>
</link>
```

Listing 3.1: The special camera link and joint added to the main URDF file

This sensor will be treated as a special link of the robot, with properties added to designate it as a sensor. The size and positioning of this sensor can be visualized in the Rviz environment, allowing a trial-and-error approach to assign consistent values to its frame origin and scale parameters.

Furthermore, it will be necessary to set specific fields unique to the camera sensor, such as the image capture resolution, the update rate, and the field of view (FOV).

```
<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>
```

Listing 3.2: The Gazebo directive to give the property of being a camera to the `camera_link`

3.2 Camera Plugin and Launch

Finally, we create a `xacro:macro` responsible for invoking the camera plugin, which will be called within the previously defined URDF file.

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

  <!-- Definizione della macro per la telecamera -->
  <xacro:macro name="arm_camera_plugin" params="">
    <gazebo>
      <plugin filename="gz-sim-sensors-system" name="
  gz::sim::systems::Sensors">
        <render_engine>ogre2</render_engine>
      </plugin>
    </gazebo>
  </xacro:macro>

</robot>
```

Listing 3.3: arm_camera.xacro

At this point, the camera is correctly set up and viewable on the robot, allowing the simulation to run in Gazebo. To view the data the camera is collecting (for example, exploiting Rviz functionality), it is necessary to establish communication between Gazebo and ROS using the ros_gz_bridge, which provides a network bridge to enable message exchange between ROS 2 and Gazebo.

To properly use `ros_gz_bridge`, it is essential first to identify which specific Gazebo topic you intend to connect with and then determine the type of message it supports.

This can be accomplished using the `ign topic` and `interface show` commands from the command window, which help reveal both the available topics and their corresponding message types.

By typing `ign topic -l` and then `ign topic -i --topic /camera` we obtain:

While by typing `ros2 interface list --only-msgs` we obtain:

```
user@william-G5-KC:~/ros2_ws$ ign topic -l
/camera
/camera_info
/clock
/gazebo/resource_paths
/gui/camera/pose
/sensors/marker
/stats
/world/empty/clock
/world/empty/dynamic_pose/info
/world/empty/pose/info
/world/empty/scene/deletion
/world/empty/scene/info
/world/empty/state
/world/empty/stats
user@william-G5-KC:~/ros2_ws$ ign topic -i --topic /camera
Publishers [Address, Message Type]:
  tcp://172.17.0.1:39299, ignition.msgs.Image
  tcp://172.17.0.1:37999, ignition.msgs.Image
user@william-G5-KC:~/ros2_ws$
```

Figure 3.1: Gazebo topics and types of message

```
sensor_msgs/msg/CameraInfo
sensor_msgs/msg/ChannelFloat32
sensor_msgs/msg/CompressedImage
sensor_msgs/msg/FluidPressure
sensor_msgs/msg/Illuminance
sensor_msgs/msg/Image
```

Figure 3.2: ROS2 Camera and Image types of message

Once this information is gathered, you can create the bridge connection. The effect of the bridge is the creation of a topic with the same name within ROS, effectively allowing ROS nodes to communicate seamlessly with Gazebo topics.

To prove the existence and visualize the content transmitted to the `/videocamera` topic, we will make use of the `rqt_image_view` tool. Adding an object to the Gazebo simulation environment, it will be seen also in the camera topic.

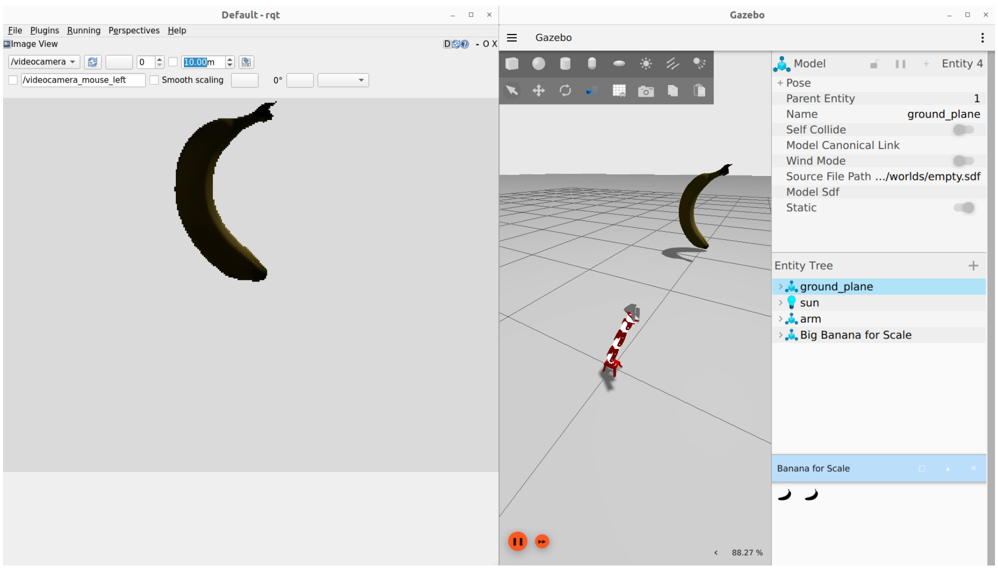


Figure 3.3: Publishing of the image topic using `rqt_image_view`

Chapter 4

Constructing a Joint Control ROS Publisher Node

The aim of the last chapter is to develop a ROS C++ publisher node within the `arm_controller` package that reads the robot's joint state data and sends joint position commands to control the robot. This involves creating a node, configuring its dependencies in `CMakeLists.txt`, and setting up a subscriber to the `joint_states` topic to monitor current joint positions. The node will also publish commands to `/position_controller/command` topics, thereby facilitating joint position control for the robot manipulator.

4.1 ROS arm_controller_node

Now that the Gazebo environment is set up, the robot is spawned, and the position controllers for each joint are managed by the `controller_manager`, we can implement control. This is handled by a C++ node that functions as a publisher on the `position_controller` topic, where the controllers listen for control input, and as a subscriber on the `joint_states` topic, which publishes the robot's current state in terms of joint variables, enabling closed-loop control.

```cpp
#include <chrono>
#include <functional>
#include <memory>
#include <string>

#include <rclcpp/rclcpp.hpp>
#include <sensor_msgs/msg/joint_state.hpp>
#include <std_msgs/msg/float64_multi_array.hpp>

using namespace std::chrono_literals;

class ArmControllerNode : public rclcpp::Node
{
public:
    ArmControllerNode() : Node("arm_controller_node")
    {
        // Subscriber to joint_states topic
        joint_state_subscriber_ = this->create_subscription<sensor_msgs::
    msg::JointState>(
            "/joint_states", 10,
            std::bind(&ArmControllerNode::jointStateCallback, this, std::
    placeholders::_1));
```

```cpp

        // Publisher to position_controller/command topic
        position_command_publisher_ = this->create_publisher<std_msgs::msg::Float64MultiArray>(
            "/position_controller/commands", 10);
        timer_ = this->create_wall_timer(500ms, std::bind(&ArmControllerNode::timer_callback, this));
    }

private:
    void jointStateCallback(const sensor_msgs::msg::JointState::SharedPtr msg)
    {
        // Print current joint positions
        RCLCPP_INFO(this->get_logger(), "Current Joint Positions:");
        for (size_t i = 0; i < msg->position.size(); ++i)
        {
            RCLCPP_INFO(this->get_logger(), "Joint %zu: %f", i, msg->position[i]);
        }
    }

    void timer_callback()
    {
      // Publish a command to the joints
      std_msgs::msg::Float64MultiArray command_msg;
      command_msg.data = {1.0, 0.5, 0.0, 0.0}; // Desired positions for the joints
      position_command_publisher_->publish(command_msg);
    }

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

int main(int argc, char **argv)
{
    rclcpp::init(argc, argv);
    rclcpp::spin(std::make_shared<ArmControllerNode>());
    rclcpp::shutdown();
    return 0;
}
```

Listing 4.1: `arm_controller_node.cpp`

After creating the node, you can view the information flow by displaying the `rqt_graph` on the screen.

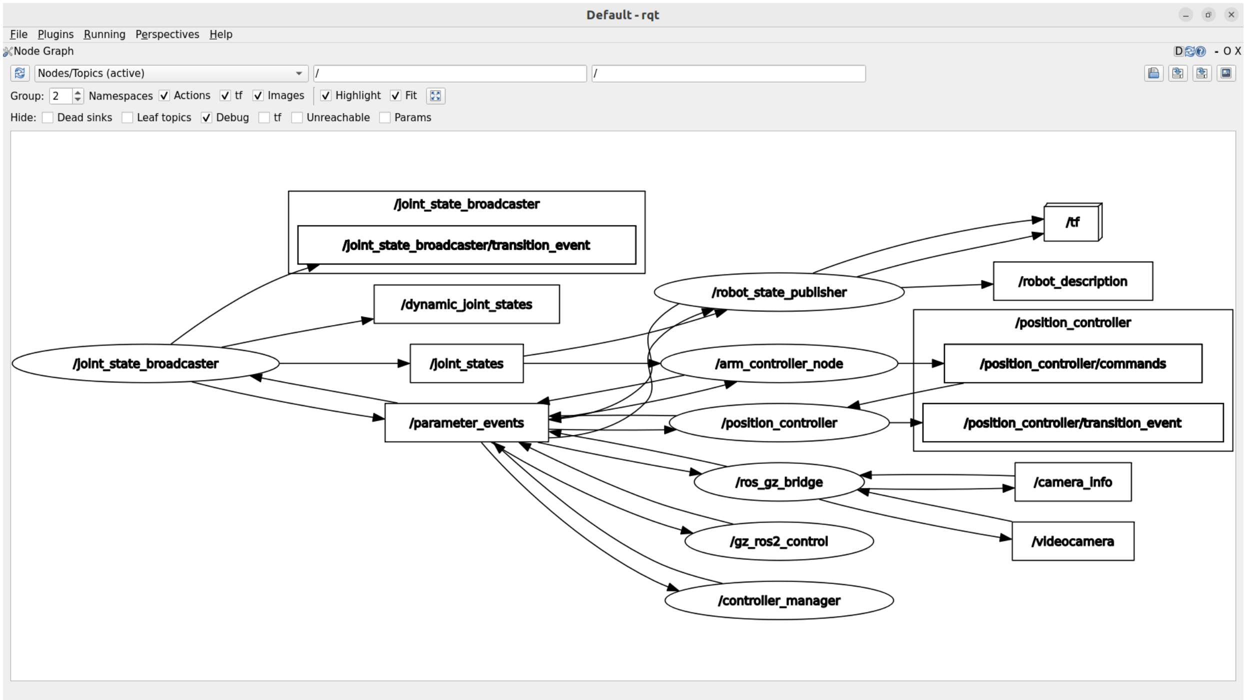


Figure 4.1: RQT flow of the whole project