DIETI. UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II
DIPARTIMENTO DI INGEGNERIA ELETTRICA E TECNOLOGIE DELL'INFORMAZIONE

Robotics Lab course

Homework 1
Building your robot manipulator

Student:
Daniele Fontana
P38000297

Teacher:
Prof. Mario Selvaggio
Assistants:
Andrea Capuozzo
Claudio Chiariello

Create the description of your robot and visualize it in Rviz

- Download the arm_description package from the repository into your ros2_ws using git commands:

```
$ cd ~/ros2_ws/src
$ git clone https://github.com/RoboticsLab2024/arm_description.git
```

- Within the package create a launch folder containing a launch file named display.launch that loads the URDF as a robot_description ROS param and starts the robot_state_publisher node, the joint_state_publisher node and the rviz2 node. Launch the file using ros2 launch. **Optional**: save a .rviz configuration file, that automatically loads the RobotModel plugin by default, and give it as an argument to your node in the display.launch file.

```
$ cd arm_description
$ mkdir launch
$ cd launch
$ touch display.launch.py
```

In ROS 2, launch files can be created in Python, XML, or YAML formats. Python was chosen due to its clarity and flexibility for writing display.launch.py.

display.launch.py

```
def generate_launch_description():

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

    robot_description_links = {"robot_description": robot_desc}

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

    robot_state_publisher_node = Node(
        package='robot_state_publisher',
        executable='robot_state_publisher',
        output='      both      ',
        parameters=[robot_description_links,
                    {" use_sim_time ":       True }],
        remappings=[('/robot_description', '/robot_description')]
    )

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

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

This Python launch file loads multiple ROS 2 nodes essential for robot visualization and interaction. After parsing the robot's URDF file, it publishes the robot description to the robot_state_publisher node, enabling real-time updates of the robot's state, while joint_state_publisher allows interactive adjustment of joint values. To build correctly the package, one must add the following instructions to the CMakeLists.txt:

CMakeLists.txt

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

After building the package, the RViz2 node is launched. By changhing the Fixed Frame in the lateral bar and adding the RobotModel plugin interface, the manipulator can be visualized.

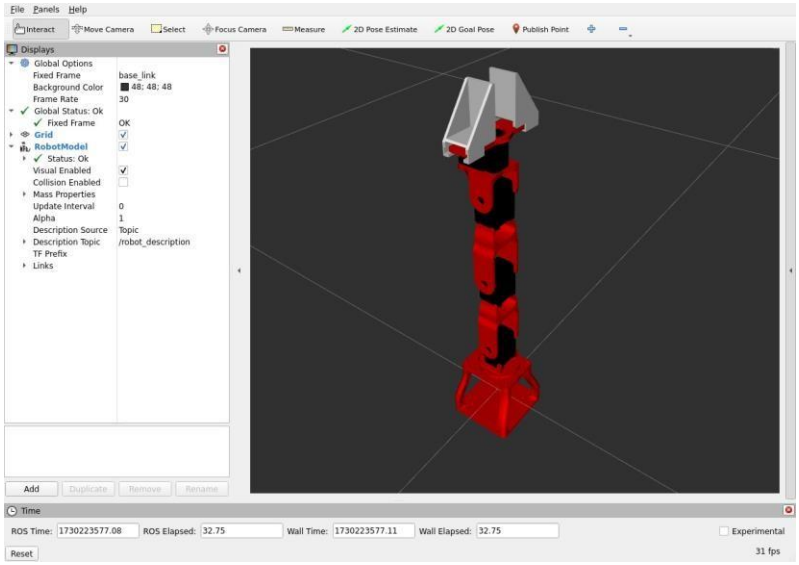


Figure 1: robot arm model in RViz

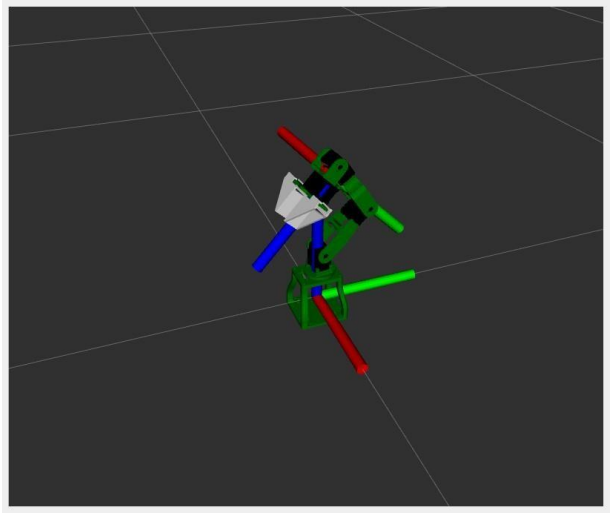


Figure 2: robot arm model in RViz with joint state publisher actuation

A configuration file is saved and loaded by modifying the instruction for launching RViz.

display.launch.py

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

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

- Substitute the collision meshes of your URDF with primitive shapes. Use <box> geometries of reasonable size approximating the links. Hint: Enable collision visualization in RViz (go to the lateral bar > Robot model > Collision Enabled) to adjust the collision meshes size.

  Thanks to a CAD software it has been possible to size accurately every box:

arm.urdf

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

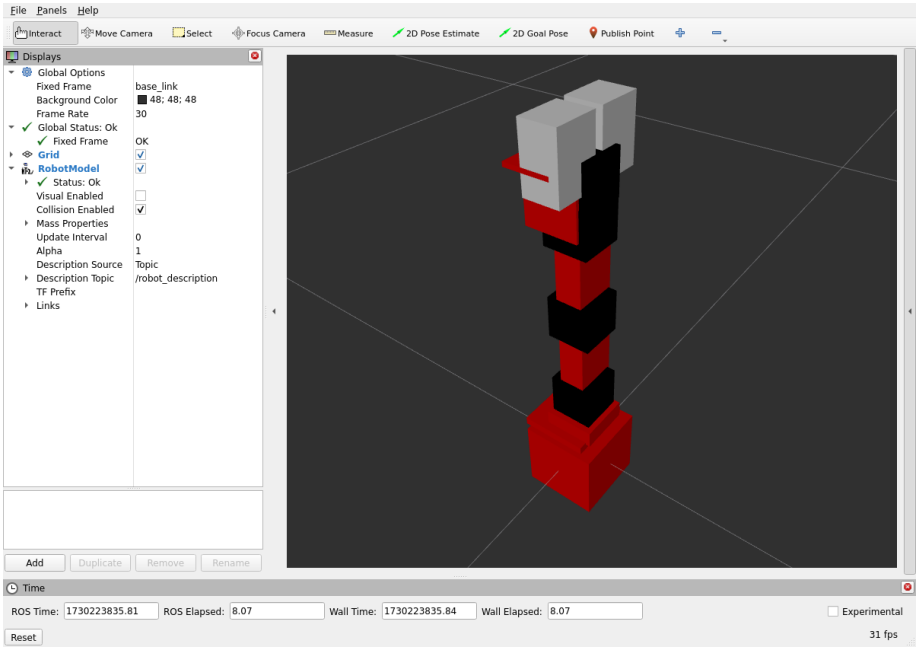


Figure 3: robot arm model in RViz with primitives shapes

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

- Create a package named arm_gazebo.

```
$ cd ~/ros2_ws/src
$ ros2 pkg create --build-type ament_cmake --license Apache-2.0 arm_gazebo
```

- Within this package create a launch folder containing a arm_world.launch file.

```
$ cd arm_gazebo
$ mkdir launch
$ cd launch
$ touch arm_world.launch.py
```

- Fill this launch file with commands that load the URDF into the /robot_description topic and spawn your robot using the create node in the ros_gz_sim package. Launch the arm_world.launch file to visualize the robot in Gazebo.

  In the first part the urdf was automatically generated from arm.urdf.xacro in the launch file. This is convenient because it stays up to date and doesn't use up hard drive space. To run xacro within the launch file, it has been used the command substitution as a parameter to the robot_state_publisher node, that uses the URDF description to broadcast the robot's joint states and transforms in a format other nodes can subscribe to.

arm_world.launch.py

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

# Use xacro to process the file and create the robot description parameter
robot_description_xacro = {"robot_description": Command(['xacro ', xacro])}

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

  Then, it's possible to declare a launch argument, gz_args, which sets default parameters for the Gazebo simulation. These defaults initialize the Gazebo simulator with an empty world, as an sdf file.

arm_world.launch.py

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

  In order to launch Gazebo, the Gazebo ignition launch file (gz_sim.launch.py) has been included with the IncludeLaunchDescription action, based on the provided launch arguments. A node is created to spawn the robot model into the Gazebo simulation. It utilizes the robot's description

(from the robot_description topic) and the specified initial position. The node is named "arm" and allows renaming if there's a naming conflict within the simulation.

arm_world.launch.py

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

position = [0.0, 0.0, 0.045]

# Define a Node to spawn the robot in the Gazebo simulation
gz_spawn_entity = Node(
    package='ros_gz_sim',
    executable='create',
    output='screen',
    arguments=['-topic', 'robot_description',
               '-name', 'arm',
               '-allow_renaming', 'true',
                "-x", str(position[0]),
                "-y", str(position[1]),
                "-z", str(position[2]),],
)
```

Finally, the last part gathers all nodes and arguments needed to start the simulation. By using the launch file, the robot is correctly loaded into the Gazebo physics engine.

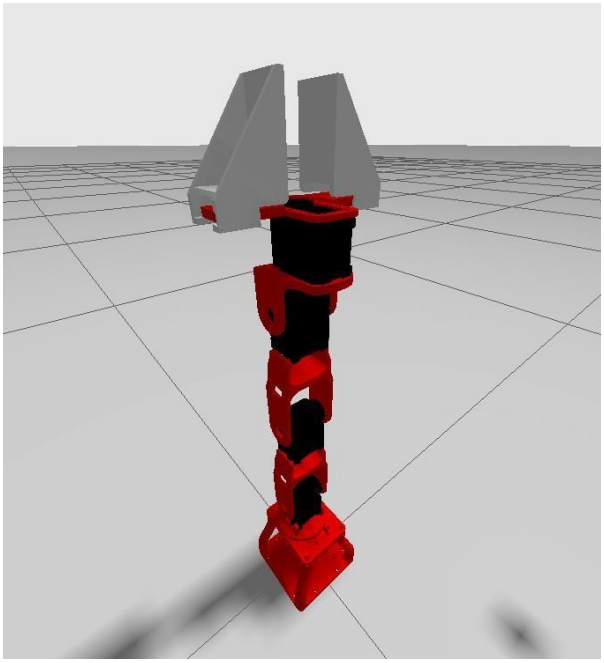


Figure 4: Robot arm in gazebo

- Add a PositionJointInterface as a hardware interface to your robot using ros2_control. Create an arm_hardware_interface.xacro file in the arm_description/urdf folder, containing a macro that defines the hardware interface for the joint, and include it in your main arm.urdf.xacro file using xacro:include. Specifically, define the joint using ros2_control and specify the hardware interface as PositionJointInterface.

arm_hardware_interface.xacro

```
<?xml version="1.0"?>
<!-- macro for creating a joint with a command and feedback interfaces -->
<robot xmlns:xacro="http://ros.org/wiki/xacro">
    <xacro:macro name="position_joint_interface" params="joint_name
    initial_pos">
     <!-- Define a joint element -->
        <joint name="${joint_name}">
            <!-- Command Interface -->
            <command_interface name="position"/>
            <!-- Feedback state Interface (position, velocity, effort)-->
            <state_interface name="position">
                <param name="initial_value">${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>
```

The arm.urdf.xacro file was modified to include arm_hardware_interface.xacro and invoking the macro for each joint needed. In this way each joint is equipped with a command and feedback interfaces for position, velocity, and effort, which are essential for robot joint control and state monitoring.

arm.urdf.xacro

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

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

<!-- Hardware interface is Gazebo Fortress -->
<hardware>
    <plugin>ign_ros2_control/IgnitionSystem</plugin>
</hardware>

<!-- Hardware Interface macro call -->
<xacro:position_joint_interface joint_name="j0" initial_pos="0.0"/>
<xacro:position_joint_interface joint_name="j1" initial_pos="0.0"/>
<xacro:position_joint_interface joint_name="j2" initial_pos="0.0"/>
<xacro:position_joint_interface joint_name="j3" initial_pos="0.0"/>

</ros2_control>
```

The arm_world.launch launch file was executed again and the robot is correctly visualized.

- Add joint position controllers to your robot: create a arm_control package with arm_control.launch file inside its launch folder and a arm_control.yaml file within its config folder.

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

- Fill the arm arm_control.yaml adding a joint_state_bradcaster and a JointPositionController

to all the joints.

arm_control.yaml

```
controller_manager:
  ros__parameters:
    update_rate: 50   # Hz

    joint_state_broadcaster:
      type: joint_state_broadcaster/JointStateBroadcaster

    position_controller:
      type: position_controllers/JointGroupPositionController

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

- Add inside the arm.urdf.xacro the commands to load the joint controller configurations from the .yaml file and spawn the controllers using the controller_manager package. Then, launch the robot simulation in Gazebo and demonstrate how the hardware interface is correctly loaded and connected.

arm.urdf.xacro

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

arm_control.launch

```
def generate_launch_description():

    # Create a node to manage joint state broadcasting
    joint_state_broadcaster = Node(
        package="controller_manager",
        executable="spawner",
        arguments=["joint_state_broadcaster", "--controller-manager", "/
    controller_manager"],
    )

    # Create a node for managing a position controller
    position_controller = Node(
        package="controller_manager",
        executable="spawner",
        arguments=["position_controller", "--controller-manager", "/
    controller_manager"],
    )

    return LaunchDescription([
        joint_state_broadcaster,
        position_controller
    ])
```

- Create an arm_gazebo.launch file into the launch folder of the arm_gazebo package loading the

Gazebo world with arm_world.launch and spawning the controllers within arm_control.launch. Launch the simulation and check if your controllers are correctly loaded.

A launch file has been created to call separate launch files, each containing nodes and parameters. This includes configurations for the Gazebo world and the control setup.

arm_gazebo.launch.py

```
    # Include the arm_world launch file
    arm_world = IncludeLaunchDescription(
      PythonLaunchDescriptionSource([os.path.join(
        get_package_share_directory('arm_gazebo'), 'launch'),
        '/arm_world.launch.py'])
      )

    # Include the arm_control launch file after arm_world
    arm_control = IncludeLaunchDescription(
      PythonLaunchDescriptionSource([os.path.join(
        get_package_share_directory('arm_control'), 'launch'),
        '/arm_control.launch.py'])
      )
```

The simulation can now be launched by using the following command in the terminal:

```
$ ros2 launch arm_gazebo arm_gazebo.launch.py
```

and verify that the controllers are loaded correctly:

```
[ruby $(which ign) gazebo-2] [INFO] [1730233558.484222211] [controller_manager]: Loading controller 'joint_state_broadcaster'
[spawner-4] [INFO] [1730233558.489332483] [spawner_joint_state_broadcaster]: Loaded joint_state_broadcaster
[ruby $(which ign) gazebo-2] [INFO] [1730233558.489564129] [controller_manager]: Loading controller 'position_controller'
[ruby $(which ign) gazebo-2] [INFO] [1730233558.494155071] [controller_manager]: Configuring controller 'joint_state_broadcaster'
[ruby $(which ign) gazebo-2] [INFO] [1730233558.494294096] [joint_state_broadcaster]: 'joints' or 'interfaces' parameter is empty.
All available state interfaces will be published
[spawner-5] [INFO] [1730233558.495134584] [spawner_position_controller]: Loaded position_controller
[ruby $(which ign) gazebo-2] [INFO] [1730233558.497915986] [controller_manager]: Configuring controller 'position_controller'
[ruby $(which ign) gazebo-2] [INFO] [1730233558.498455885] [position_controller]: configure successful
[spawner-4] [INFO] [1730233558.950383097] [spawner_joint_state_broadcaster]: Configured and activated joint_state_broadcaster
[ruby $(which ign) gazebo-2] [INFO] [1730233558.960142309] [position_controller]: activate successful
[spawner-5] [INFO] [1730233558.971512613] [spawner_position_controller]: Configured and activated position_controller
```

In order to test controllers, is possible to publish a std_msgs/msg/Float64MultiArray message on the /position_controller/commands topic. Open a new terminal while the Gazebo simulation is running and use the following command (where j0, j1, j2, j3 are placeholders):

```
ros2 topic pub /position_controller/commands std_msgs/msg/Float64MultiArray "
    data: [j0, j1, j2, j3]"
```

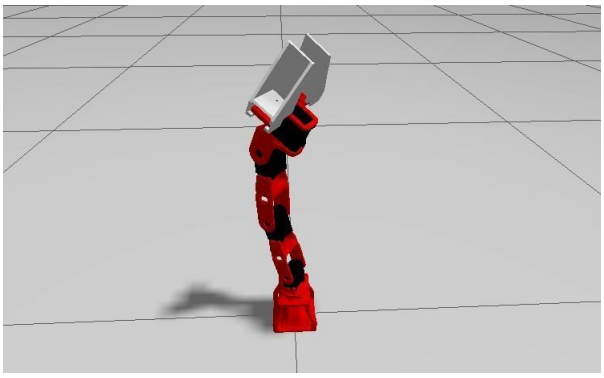


Figure 5: Robot arm in gazebo with user commands

Add a camera sensor to your robot

- Go into your arm.urdf.xacro file and add a camera_link and a fixed camera_joint with the base_link as a parent link. Size and position the camera link opportunely:

  A new link has been added to the robot's representation, connected to the base via a fixed joint. The camera is represented as a gray box, sized appropriately, and the joint is oriented as desired. Particularly, two different camera pose were considered: the first outside the robot frame, so as to capture the robot workspace, while the second on the front of the robot, as by arm design.

arm.urdf.xacro

```
<!-- camera link and camera joint -->
<joint name="camera_joint" type="fixed">
<parent link="base_link"/>
<child link="camera_link"/>
<origin xyz="0.5 -0.3 0.6" rpy="0.0 0.5 -3.66"/>
</joint>

<material name="grey">
<color rgba="0.5 0.5 0.5 1"/>
</material>

<link name="camera_link">
<visual>
    <geometry>
      <box size="0.01 0.03 0.03"/>
    </geometry>
    <material name="grey"/>
</visual>
</link>
```

- Create an arm_camera.xacro file in the arm_gazebo/urdf folder, add the gazebo sensor reference tags and the gz-sim-sensors-system plugin to your xacro.

arm_camera.xacro

```
<?xml version="1.0"?>
<robot xmlns:xacro="http://www.ros.org/wiki/xacro">
    <!-- Camera sensor porperties -->
    <xacro:macro name="arm_camera">
        <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>
```

Then in the urdf the gazebo sensor plugin must be included:

arm_camera.xacro

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

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

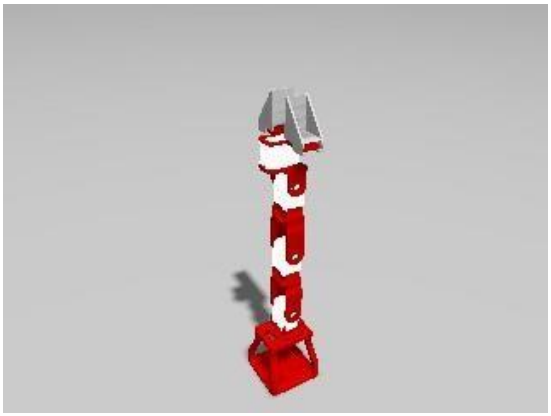
- Launch the Gazebo simulation with using arm_gazebo.launch, and check if the image topic is correctly published using rqt_image_view. **Hint**: remember to add the ros_ign_bridge.

  After adding the ros_ign_bridge to the launch file, the robotic arm simulation in Gazebo can be launched with the following command:

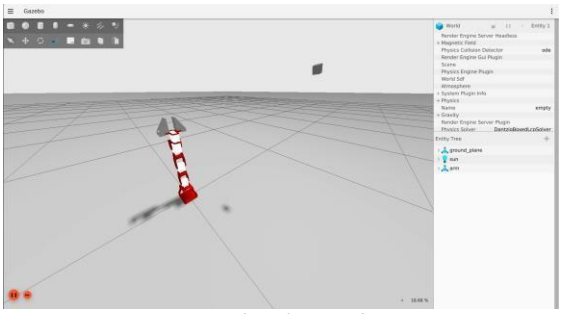
```
$ ros2 launch arm_gazebo arm_gazebo.launch.py
```

  and in another terminal open rqt_image_view and select the /videocamera topic.

```
$ ros2 run rqt_image_view rqt_image_view
```

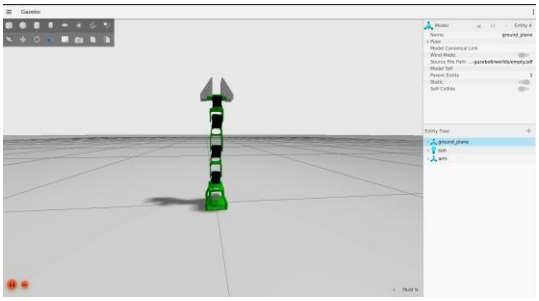


Camera view in rqt_image_view



Scene view in gazebo

Camera view in rqt_image_view



Scene view in gazebo

Figure 6: Frontal camera vs external camera

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

- Inside the arm_controller package create a ROS C++ node named arm_controller_node. The dependencies are rclcpp, sensor_msgs and std_msgs. Modify opportunely the CMakeLists.txt file to compile your node. **Hint**: use add_executable and ament_target_dependencies commands.

```
$ cd ~/ros2_ws/src
$ ros2 pkg create --build-type ament_cmake --license Apache-2.0
    arm_controller
$ cd src
$ touch arm_controller_node.cpp
```

The CMakeLists.txt was modified, in order to add the executable target and specifies dependencies that the target (in this case, the node executable) needs to link with.

CMakeLists.txt

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

- Create a subscriber to the topic joint_states and a callback function that prints the current joint positions. **Note**: the topic contains a sensor_msgs/JointState.

The next lines creates 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.

arm_controller_node.cpp

```
class JointStateSubscriber : public rclcpp::Node
{
  public:
    JointStateSubscriber()
    : Node("joint_state_subscriber")
    {
      subscription_ = this->create_subscription<sensor_msgs::msg::JointState
    >(
      "joint_states", 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_;
};
```

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.

arm_controller_node.cpp

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

```
[INFO] [1730146949.157925539] [joint_state_subscriber]: -----Joint Positions-----
[INFO] [1730146949.158075982] [joint_state_subscriber]: Joint Position 1: '-0.000000'
[INFO] [1730146949.158106114] [joint_state_subscriber]: Joint Position 2: '0.000000'
[INFO] [1730146949.158135415] [joint_state_subscriber]: Joint Position 3: '0.000000'
[INFO] [1730146949.158158066] [joint_state_subscriber]: Joint Position 4: '0.000000'
[INFO] [1730146949.181976789] [joint_state_subscriber]: -----Joint Positions-----
[INFO] [1730146949.182037823] [joint_state_subscriber]: Joint Position 1: '-0.000000'
[INFO] [1730146949.182055544] [joint_state_subscriber]: Joint Position 2: '0.000000'
[INFO] [1730146949.182066816] [joint_state_subscriber]: Joint Position 3: '0.000000'
[INFO] [1730146949.182077374] [joint_state_subscriber]: Joint Position 4: '0.000000'
```

Figure 7: joint states subscriber output

- Create publishers that write commands onto the /position_controller/command topics.

  Initially, the node class PositionControllerPublisher is created, similarly to the joint state subscriber node. The main difference is that the node declares a parameter named joint_positions, which stores the target joint positions as a vector of doubles. This parameter can be modified externally, but a default value of {1.0, 1.0, 1.0, 1.0} is provided. Moreover, the public constructor names the node position_controller_publisher and initializes count_ to 0. Inside the constructor, the publisher is initialized with the Float64MultiArray message type, the topic name /position_controller/command, and the required queue size to limit messages in the event of a backup. Next, timer_ is initialized, which causes the timer_callback function to be executed twice a second. 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. Last is the declaration of the timer, publisher, and counter fields.

arm_controller_node.cpp

```
// Joint states Publisher
class PositionControllerPublisher : public rclcpp::Node
{
  public:
    PositionControllerPublisher()
    : Node("position_controller_publisher"), count_(0)
    {
      // Declare an external parameter with a default value
      this->declare_parameter<std::vector<double>>("joint_positions", {1.0,
    1.0, 1.0, 1.0, 0.0});

      // Retrieve the parameter's value and initialize the publisher and
    timer
      publisher_ = this->create_publisher<std_msgs::msg::Float64MultiArray>("
    /position_controller/commands", 10);
      timer_ = this->create_wall_timer(
      500ms, std::bind(&PositionControllerPublisher::timer_callback, this));
    }

  private:
    void timer_callback()
    {
      // Retrieve the parameter value on each callback
      std::vector<double> joint_positions;
      this->get_parameter("joint_positions", joint_positions);

      auto commands = std_msgs::msg::Float64MultiArray();
      commands.data = joint_positions; // Initialize all values at once
      RCLCPP_INFO(this->get_logger(), "Publishing: '%zu'", count_++);
      RCLCPP_INFO(this->get_logger(), "-----Joint Commands\n");
      for (int i = 0; i < 4; i++)
        RCLCPP_INFO(this->get_logger(), "Joint command %d: '%6f'\n", i+1,
    joint_positions[i]);
      publisher_->publish(commands);
    }
    rclcpp::TimerBase::SharedPtr timer_;
    rclcpp::Publisher<std_msgs::msg::Float64MultiArray>::SharedPtr publisher_
    ;
    size_t count_;
};
```

To run two nodes into the same process the main function has been modified as follow:

arm_controller_node.cpp

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

  // Create executors for the nodes
  auto joint_state_subscriber = std::make_shared<JointStateSubscriber>();
  auto position_controller_publisher = std::make_shared<
    PositionControllerPublisher>();

  // Spin both nodes using a MultiThreadedExecutor
  rclcpp::executors::MultiThreadedExecutor executor;
  executor.add_node(joint_state_subscriber);
  executor.add_node(position_controller_publisher);
  executor.spin();

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

It creates and runs two ROS 2 nodes (JointStateSubscriber and PositionControllerPublisher) in parallel using a multi-threaded executor, which allows each node to operate independently.

```
[INFO] [1730239409.050956443] [position_controller_publisher]: Publishing: '42'
[INFO] [1730239409.051005180] [position_controller_publisher]: -----Joint Commands-----

[INFO] [1730239409.051015232] [position_controller_publisher]: Joint command 1: '1.000000'

[INFO] [1730239409.051026578] [position_controller_publisher]: Joint command 2: '1.000000'

[INFO] [1730239409.051033921] [position_controller_publisher]: Joint command 3: '1.000000'

[INFO] [1730239409.051042703] [position_controller_publisher]: Joint command 4: '1.000000'

[INFO] [1730239409.054348393] [joint_state_subscriber]: -----Joint Positions-----

[INFO] [1730239409.054425506] [joint_state_subscriber]: Joint Position 1: '1.000000'

[INFO] [1730239409.054451147] [joint_state_subscriber]: Joint Position 2: '1.000000'

[INFO] [1730239409.054468587] [joint_state_subscriber]: Joint Position 3: '1.000000'

[INFO] [1730239409.054485552] [joint_state_subscriber]: Joint Position 4: '1.000000'
```

Figure 8: node output

Finally it's time to run the simulation to test the new added custom node. In the first terminal launch the Gazebo simulation. In another terminal use the command:

```
$ ros2 run arm_controller arm_controller_node --ros-args -p joint_positions:=
    "[j0, j1, j2, j3]"
```

where [j0, j1, j2, j3] are placeholders for the joint commands. Various constant commands were tested as well as different controller parameter.

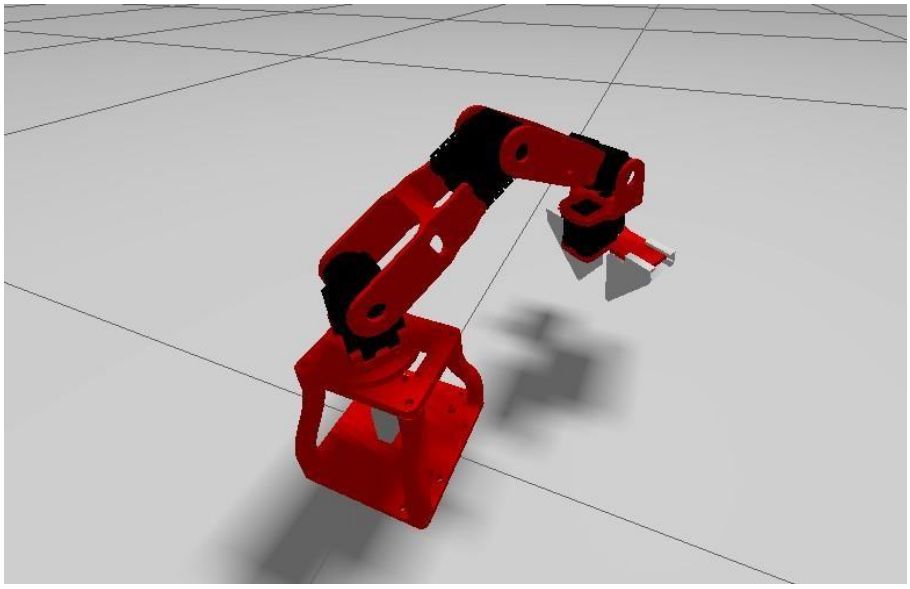


Figure 9: Robot arm in Gazebo following position commands

It is possible to find all the code used in this report at the link: https://github.com/danieffe/4-DOF-robotic-manipulator-arm.git