Homework 1

Maria Vittoria Cinquegrani

November 2, 2023

GitHub link: https://github.com/MVCinquegrani/ROS_Homework1

1 Create the Description of the Robot and Visualize it in Rviz

1a Download the arm_description Package

To start, I downloaded the `arm_description` package in my `catkin_ws` folder from the provided GitHub repository. This package contains the necessary files to describe the robot's physical characteristics and geometry.

```
$ git clone https://github.com/RoboticsLab2023/arm_description.git
```

This command cloned the `arm_description` package from the GitHub repository to my local ROS workspace.

1b Launch File and Rviz

In order to visualize the robot's description in Rviz, I performed the following steps; Within the `arm_description` package, I created a launch folder and inside it, I added a launch file named display.launch.

```
$ roscd arm_description blue blue#blue blueNavigateblue bluetoblue bluetheblue
    bluepackageblue'bluesblue bluedirectory
$ mkdir launch
$ touch display.launch
```

This launch file is responsible for loading the URDF (Unified Robot Description Format) as a `robot_description` ROS parameter and starting essential nodes, including `robot_state_publisher`, `joint_state_publisher`, and Rviz, hence:

```
<launch>
  <!blue--blue blueLoadblue bluetheblue bluerobotblue bluedescriptionblue blue-->
  <param name="robot_description" textfile="$(find arm_description)/urdf
  /arm_description.urdf"/>

  <!blue--blue blueStartblue bluetheblue bluerobotblue bluestateblue bluepublisher
      blue blue-->
  <node name="robot_state_publisher" pkg="robot_state_publisher"
  type="robot_state_publisher" />

  <!blue--blue blueStartblue bluetheblue bluejointblue bluestateblue bluepublisher
      blue blue-->
  <node name="joint_state_publisher" pkg="joint_state_publisher"
  type="joint_state_publisher" />

  <!blue--blue blueStartblue blueRvizblue bluewithblue blueRobotModelblue blueplugin
      blue blue-->
  <node name="rviz" pkg="rviz" type="rviz" args="-d $(arg rvizconfig)" />
</launch>
```

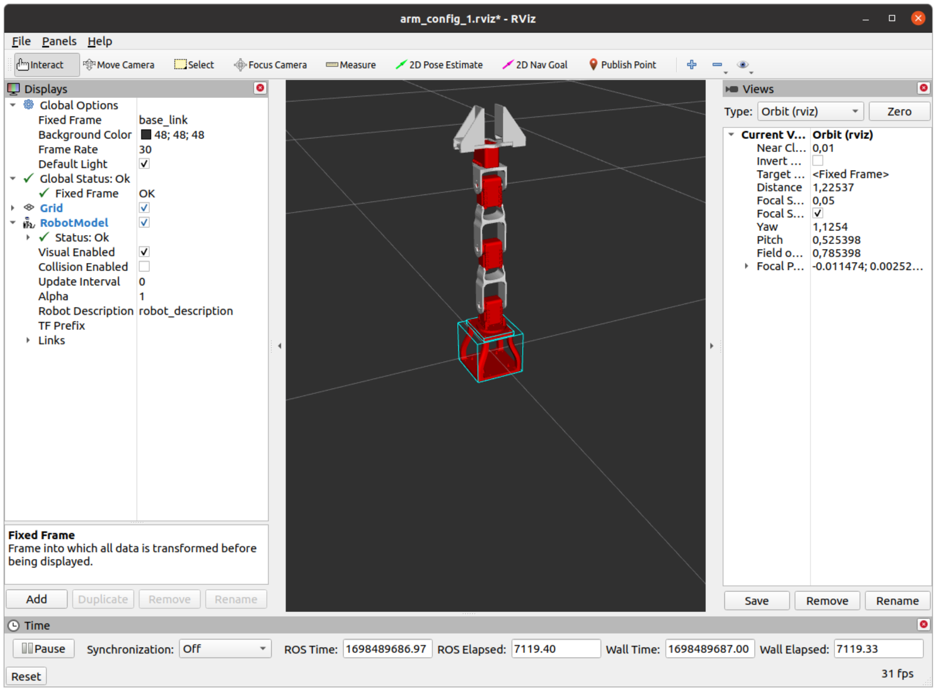

Figure 1:

Than I launched the display.launch file and I ran Rviz using the following command from terminal:

```
$ roslaunch arm_description display.launch
$ rosrun rviz rviz
```

I configured Rviz by setting the "Fixed Frame" to the base link frame and added the "RobotModel" plugin to visualize the robot's geometry; then I saved the Rviz configuration in a desktop folder.

1c Edit URDF for Collision Shapes

I edited the URDF file of the robot (arm.urdf) using a text editor by replacing the existing collision elements with more straightforward primitive shapes such as box. For instance, a representative box collision shape is defined in the URDF file as follows:

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

In the example, a basic box shape with dimensions of 0.1 x 0.1 x 0.1 meters was chosen. The size of the primitive shape is chosen according to the approximate dimensions of the link it represents. With the URDF file updated to include these simplified collision shapes, we can visualize the collision in Rviz as shown in figure 2 by enabling collision visualization in rviz window.

1d Creating and Importing arm.gazebo.xacro in URDF

After creating a file named `arm_gazebo.xacro` within `arm_description/urdf` folder, I defined a xacro:macro within this file. This macro serves as a container for all the <gazebo> tags that are present within the arm.urdf. The structure of the arm.gazebo.xacro file was designed as follows:

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

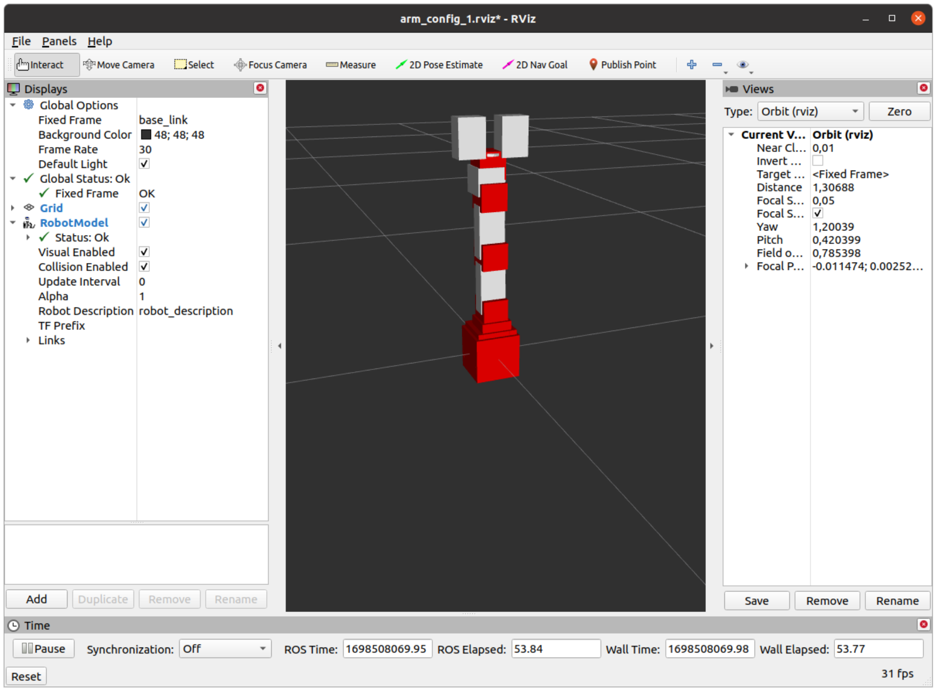

Figure 2:

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

    <gazebo reference="f4">
        <material>Gazebo/Red</material>
    </gazebo>

    <gazebo reference="f5">
        <material>Gazebo/Red</material>
    </gazebo>

    ...

  </xacro:macro>
</robot>
```

To link the URDF file, `arm_urdf.xacro`, with the Gazebo configurations, I renamed the URDF file `arm.urdf` to `arm.urdf.xacro`; then I make modifications to this file by using a text editor and writing the string xmlns:xacro="http://www.ros.org/wiki/xacro" within the `<robot>` tag. Then I used the `xacro:include` directive to include the `arm_gazebo.xacro`. The URDF file structure is the following:

```
<robot name="arm" xmlns:xacro="http://www.ros.org/wiki/xacro">
<!blue--blue blueIncludeblue bluearmblue.bluegazeboblue.bluexacroblue bluefileblue
    blue-->
<xacro:include filename="$(find arm_description)/urdf/arm_gazebo.xacro"/>

  <!blue--blue bluerobot's links and joints definitions go here -->

</robot>
```

With the URDF effectively prepared, the final steps involve loading it into a launch file using the xacro routine, thereby facilitating efficient robot simulation and visualization. Hence I edited the launch file `display.launch` by modifing the already existing `robot_description` parameter and using its "command" attribute to run the "xacro" command:

```
<launch>
  <!blue--blue blueIncludeblue bluexacroblue blueandblue bluedefineblue bluetheblue
      bluexacroblue blueelementblue blue-->
  <param name="robot_description" command="$(find xacro)-
  /xacro '$(find arm_description)/urdf/arm.urdf.xacro'"/>
</launch>
```

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

2a Create a arm_gazebo Package

In the Catkin workspace I used the "`catkin_create_pkg`" command to create a new package named `arm_gazebo` and I build it to make the new package available for use.

```
$ redcd /.../catkin_ws/src
$ catkin_create_pkg arm_gazebo roscpp rospy std_msgs gazebo_ros
$ catkin_build
```

2b Create a Launch folder and file

Within the `arm_gazebo` package I created a launch folder and within it I created a `arm_world.launch` file. These are the commands executed from the terminal:

```
$ redcd /catkin_ws/src/arm_gazebo
$ mkdir launch
$ touch launch/arm_world.launch
```

2c Edit the Launch File for Gazebo

I edited the `arm_world.launch` file by following the `iiwa_world.launch` example from the package `iiwa_stack`. So, I filled in the lunch file as shown as follows:

```
<?redxml redversion="1.0"?>
<launch>

    <!blue--blue blueLoadsblue bluetheblue bluearmblue.blueworldblue blueenvironment
        blue blueinblue blueGazeboblue.blue blue-->

    <arg name="paused" reddefault="false"/>
    <arg name="use_sim_time" reddefault="true"/>
    <arg name="gui" reddefault="true"/>
    <arg name="headless" reddefault="false"/>
    <arg name="hardware_interface" reddefault="PositionJointInterface"/>
    <arg name="debug" reddefault="false"/>
    <arg name="robot_name" reddefault="arm" />

    <include file="$(find gazebo_ros)/launch/empty_world.launch">
        <arg name="debug" value="$(arg debug)" />
        <arg name="gui" value="$(arg gui)" />
        <arg name="paused" value="$(arg paused)"/>
        <arg name="use_sim_time" value="$(arg use_sim_time)"/>
        <arg name="headless" value="$(arg headless)"/>
    </include>

    <!blue--blue blueLoadblue bluetheblue blueURDFblue bluewithblue bluetheblue
        bluegivenblue bluehardwareblue blueinterfaceblue blueintoblue bluetheblue
        blueROSblue blueParameterblue blueServerblue blue-->
```

```
    <include file="$(find arm_description)/launch/$(arg model)_upload.launch">
        <arg name="hardware_interface" value="$(arg hardware_interface)"/>
        <arg name="robot_name" value="$(arg robot_name)" />
    </include>

    <!blue--blue blueRunblue blueablue bluepythonblue bluescriptblue bluetoblue
       bluesendblue blueablue blueserviceblue bluecallblue bluetoblue bluegazebo_ros
       blue bluetoblue bluespawnblue blueablue blueURDFblue bluerobotblue blue-->
    <node name="urdf_spawner" pkg="gazebo_ros" type="spawn_model" respawn="
       false" output="screen"
         args="-urdf -model arm -param robot_description"/>
</launch>
```

Starting with the first line these are the changes I made:

- I replaced the default value of `robot_name` as ”arm”.
- I changed the correct path to the `arm_upload.launch` (that I created after) in the command that includes this other launch file responsible for loading the URDF model of the robot into the ROS Parameter Server.
- I also replace ”arm” insted of ”iiwa” in the last line between -model and -param.

Than I created the `arm_upload.launch` in the `arm_description/launch` folder and I write within it the following commands, always taking as an example the `iiwa7_upload.launch` file.

```
<?redxml redversion="1.0"?>
<launch>

    <!blue--blue blueThisblue bluelauchblue bluefileblue bluejustblue blueloadsblue
       bluetheblue blueURDFblue bluewithblue bluetheblue bluegivenblue bluehardware
       blue blueinterfaceblue blueandblue bluerobotblue bluenameblue blueintoblue
       bluetheblue blueROSblue blueParameterblue blueServerblue blue-->
  <arg name="hardware_interface" reddefault="PositionJointInterface"/>
  <arg name="robot_name" reddefault="arm"/>
  <arg name="origin_xyz" reddefault="'0 0 0'"/>
  <arg name="origin_rpy" reddefault="'0 0 0'"/>

  <param name="robot_description" command="$(find xacro)/xacro '$(find
     arm_description)/urdf/arm.urdf.xacro' hardware_interface:=$(arg
     hardware_interface)"/>
</launch>
```

We can visualize the robot in Gazebo, shown in figure 3, by launching the `arm_world.launch` file:

```
$ catkin build
$ redsource devel/setup.bash
$ roslaunch arm_gazebo arm_world.launch
```

2d Add a PositionJointInterface as Hardware Interface to Your Robot

The goal is to introduce a `PositionJointInterface` as a hardware interface, a fundamental component for joint position control. Hence I created the `arm_transmission.xacro` file within the `arm_description/urdf` folder. Than I defined the `<xacro:include>` in the `arm_transmission.xacro` that encapsulates the hardware interface’s requirements, creating a transmission that employed the `transmission_interface/PositionJointInterface` for joint control. The structure of that file is provided:

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

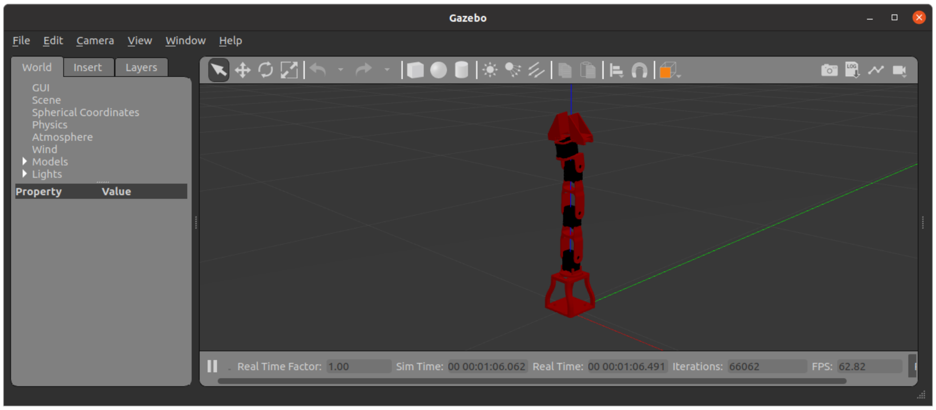

Figure 3:

```
<!blue--blue blueDefineblue blueablue bluemacroblue bluetoblue bluecreateblue
    blueablue bluePositionJointInterfaceblue blueforblue blueeveryblue
    bluejointblue blue-->
<xacro:macro name="arm_transmission">

<xacro:arg name="hardware_interface" reddefault="
    PositionJointInterface"/>


 <transmission name="Tj0">
  <type>transmission_interface/SimpleTransmission</type>
  <joint name="j0">
   <hardwareInterface>hardware_interface/$(arg hardware_interface)</
       hardwareInterface>
 </joint>

 <actuator name="motor_j0">
   <hardwareInterface>hardware_interface/$(arg hardware_interface)</
       hardwareInterface>
   <mechanicalReduction>1.0</mechanicalReduction>
 </actuator>
</transmission>

<transmission name="Tj1">
  <type>transmission_interface/SimpleTransmission</type>
  <joint name="j1">
   <hardwareInterface>hardware_interface/$(arg hardware_interface)</
       hardwareInterface>
 </joint>
 <actuator name="motor_j1">
   <hardwareInterface>hardware_interface/$(arg hardware_interface)</
       hardwareInterface>
   <mechanicalReduction>1.0</mechanicalReduction>
 </actuator>
</transmission>

<transmission name="Tj2">
  <type>transmission_interface/SimpleTransmission</type>
  <joint name="j2">
   <hardwareInterface>hardware_interface/$(arg hardware_interface)</
       hardwareInterface>
 </joint>
 <actuator name="motor_j2">
```

```
            <hardwareInterface>hardware_interface/$(arg hardware_interface)</
                hardwareInterface>
            <mechanicalReduction>1.0</mechanicalReduction>
          </actuator>
        </transmission>

        <transmission name="Tj3">
          <type>transmission_interface/SimpleTransmission</type>
          <joint name="j3">
            <hardwareInterface>hardware_interface/$(arg hardware_interface)</
                hardwareInterface>
          </joint>
          <hardwareInterface>hardware_interface/$(arg hardware_interface)</
              hardwareInterface>
          <actuator name="motor_j3">
            <mechanicalReduction>1.0</mechanicalReduction>
          </actuator>
        </transmission>

        <xacro:macro>

</robot>
```

As the code shows there are four transmission elements `Tj0, Tj1, Tj2, Tj3` that connects the four not fixed joint `j0, j1, j2, j3` to four actuators named `motor_j0`, `motor_j1`, `motor_j2` and `motor_j3`. These transmission elements specify that the hardware interface for both the joints and the motors is the `PositionJointInterface`. The `mechanicalReduction` parameter is set to 1.0, which means there is no mechanical reduction between the joints and the actuators.

To integrate this hardware interface into our robot model, the `<xacro:include>` tag was employed within the `arm.urdf.xacro` file, at the end of the file this macro is used. Below are the lines of code added to the `arm.urdf.xacro` file.

```
<robot name="arm" xmlns:xacro="http://www.ros.org/wiki/xacro">
<xacro:include filename="$(find arm_description)/urdf/arm.transmission.xacro"/
    >
...
<xacro:arm_transmission />
</robot>
```

2e Create package for controllers

To configure the workspace correctly before going ahead the following steps are performed:

- I created a new ROS package named `arm_control`

  ```
  catkin_create_pkg arm_control roscpp controller_manager
      controller_manager_msgs
  ```

- Inside the `arm_control` package, I created a launch folder.
- Within the launch folder, I created an `arm_control.launch` file.
- Inside the `arm_control` package, I created a config folder.
- Within the config folder, I created an `arm_control.yaml` file.

2f Fill the `arm_control.launch` file

To fill the `arm_control.launch` file, I took inspiration from the provided `iwa_control.launch` example, which outlined the necessary components and structure for our custom launch configuration. The following code shows the structure of the `arm_control.launch` file.

```
<?redxml redversion="1.0"?>
<launch>
    <!blue--blue blueDefineblue blueargumentsblue blueforblue bluehardwareblue
        blueinterfaceblue,blue bluecontrollersblue,blue bluerobotblue bluenameblue,
        blue blueandblue bluemodelblue blue-->
    <arg name="hardware_interface" reddefault="PositionJointInterface"/>
    <arg name="controllers" reddefault="j0p_controller j1p_controller
        j2p_controller j3p_controller"/>
    <arg name="robot_name" reddefault="arm" />

    <!blue--blue blueLoadsblue bluejointblue bluecontrollerblue blueconfigurations
        blue bluefromblue blueYAMLblue bluefileblue bluetoblue blueparameterblue
        blueserverblue blue-->
    <rosparam file="$(find arm_control)/config/arm_control.yaml" command="load
        " />

    <!blue--blue blueLoadsblue bluetheblue bluecontrollersblue blue-->
    <node name="controller_spawner" pkg="controller_manager" type="spawner"
        respawn="false"
          output="screen" ns="arm" args="$(arg controllers)" />

    <!blue--blue blueConvertsblue bluejointblue bluestatesblue bluetoblue blueTFblue
        bluetransformsblue blueforblue bluervizblue,blue blueetcblue blue-->
    <node name="robot_state_publisher" pkg="robot_state_publisher" type="
        robot_state_publisher"
          respawn="false" output="screen">
        <remap from="joint_states" to="/$(arg robot_name)/joint_states" />
    </node>
</launch>
```

Here's an explanation of the modifications implemented:

- In the ¡arg¿ elements, set the default values for the robot's hardware interface (`PositionJointInterface`) and robot name (`arm`).
- Modify the `<arg>` attribute to include all the joint controllers that I called `j0p_controller, j1p_controller, j2p_controller, j3p_controller`.
- Updated the rosparam line to load the `arm_control.yaml` configuration file.
- Add the namespace `ns="arm"` to the `controller_spawner` node.

2g Fill the `arm_control.yaml` file

Even to fill the `arm_control.yaml` file, I took inspiration from the provided `iwa_control.launch` example. It is important to mention that this file is highly sensitive to indentation. The structure of that file is shown below:

```
arm:

  # Joint State Controller
  joint_state_controller:
    type: joint_state_controller/JointStateController
    publish_rate: 50
  #  joints:
  #    - j0  # List all your joint names

  # Joint Position Controllers
  j0p_controller:
    type: position_controllers/JointPositionController
    joint: j0
    pid: {p: 100.0, i: 0.01, d: 10.0}
```

```
  j1p_controller:
    type: position_controllers/JointPositionController
    joint: j1
    pid: {p: 100.0, i: 0.01, d: 10.0}

  j2p_controller:
    type: position_controllers/JointPositionController
    joint: j2
    pid: {p: 100.0, i: 0.01, d: 10.0}

  j3p_controller:
    type: position_controllers/JointPositionController
    joint: j3
    pid: {p: 100.0, i: 0.01, d: 10.0}
```

As required I provided `joint_state_controller` that is responsible for publishing joint states and I listed the position controller for each of the arm robot's joints. There is also a standard PID gains specified for every position controller.

2h Create the `arm_gazebo.launch` File

In this step, I created an `arm_gazebo.launch file` in the launch folder of the `arm_gazebo` package. This launch file orchestrates the Gazebo simulation and loads the controllers for the robot. I begun by opening a text editor and crafting the `arm_gazebo.launch` file with the following content:

```
<?redxml redversion="1.0"?>
<launch>
    <!blue--blue blueLoadblue blueGazeboblue blueworldblue blue-->
    <include file="$(find arm_gazebo)/launch/arm_world.launch" />

    <!blue--blue blueLoadblue bluecontrollersblue blueforblue bluetheblue bluerobot
       blue blue-->
    <include file="$(find arm_control)/launch/arm_control.launch" />
</launch>
```

This launch file plays a crucial role in the simulation setup by including two essential components:

- Loading the Gazebo World: We include `arm_world.launch` to bring the Gazebo world into the simulation environment.
- Spawning Controllers: We incorporate `arm_control.launch` to spawn the controllers for the robot, ensuring that it operates as intended.

In the `arm_description` package and I add the `gazebo_ros_control` plugin within the `<robot>` section of the `arm_gazebo.xacro` file.

```
<!blue--blue blueIncludeblue bluegazebo_ros_controlblue bluepluginblue blue-->
<gazebo>
  <plugin name="gazebo_ros_control" filename="libgazebo_ros_control.so">
    <robotNamespace>/arm</robotNamespace>
  </plugin>
</gazebo>
```

I executed the following command to launch the Gazebo simulation:

```
$ roslaunch arm_gazebo arm_gazebo.launch
```

.

In the figure we can see on the left side the presence of the `gazebo_ros_control` plugin.

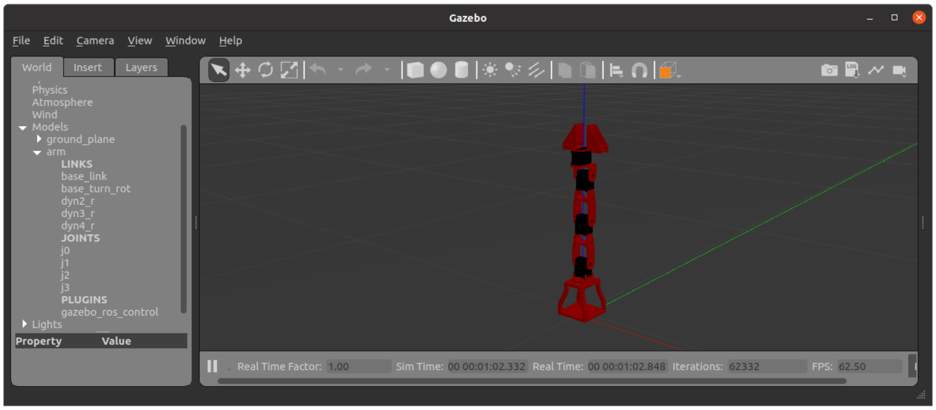

Figure 4:

3 Add a Camera Sensor to Your Robot

3a Modification of arm.urdf.xacro

Within the arm.urdf.xacro file, an essential modification was made to introduce the camera link and establish a fixed camera joint. The code snippet for this section of the URDF file is as follows:

```
<!blue--blue blueCamerablue blueJointblue blueandblue bluelinkblue blue-->
<joint name="camera_joint" type="fixed">
  <parent link="base_link"/>
  <child link="camera_link"/>
  <origin xyz="0.75 0 0.5" rpy="0 0.3 -3.14"/>
</joint>

<link name="camera_link">
  <visual>
    <geometry>
      <box size="0.05 0.05 0.05"/>
    </geometry>
  </visual>
</link>
```

This addition establishes the camera link as a child of the base link, determining the camera's relative position and orientation in the robot's structure. I chose the position and orientation of the camera so that it looks at the robotic arm.

3b Creation of camera.xacro File and Gazebo Sensor Configuration

A separate `camera.xacro` file was created, which included Gazebo sensor reference tags and the essential `libgazebo_ros_camera` plugin. It also references the `camera_link` as the location of the camera sensor. The structure of the `camera.xacro` file shown below is taken from the slides 74-75.

```
<?redxml redversion="1.0"?>

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

  <xacro:macro name="arm_camera">

    <gazebo reference="camera_link">
      <sensor type="camera" name="camera1">
        <update_rate>30.0</update_rate>
        <camera name="head">
          <horizontal_fov>1.3962634</horizontal_fov>
```

```
          <image>
            <width>800</width> <height>800</height> <format>R8G8B8</format>
          </image>
          <clip>
            <near>0.02</near> <far>300</far>
          </clip>
          <noise>
            <type>gaussian</type> <mean>0.0</mean> <stddev>0.007</stddev>
          </noise>
        </camera>
        <plugin name="camera_controller" filename="libgazebo_ros_camera.so">
          <alwaysOn>true</alwaysOn>
          <updateRate>0.0</updateRate>
          <cameraName>camera</cameraName>
          <imageTopicName>image_raw</imageTopicName>
          <cameraInfoTopicName>camera_info</cameraInfoTopicName>
          <frameName>camera_link_optical</frameName>
          <hackBaseline>0.0</hackBaseline>
          <distortionK1>0.0</distortionK1>
          <distortionK2>0.0</distortionK2>
          <distortionK3>0.0</distortionK3>
          <distortionT1>0.0</distortionT1>
          <distortionT2>0.0</distortionT2>
          <CxPrime>0</CxPrime>
          <Cx>0.0</Cx>
          <Cy>0.0</Cy>
          <focalLength>0.0</focalLength>
        </plugin>
      </sensor>
    </gazebo>

  </xacro:macro>

</robot>
```

3c Validation through Gazebo Simulation and `rqt_image_view`

To ensure the proper functioning of the added camera sensor, the Gazebo simulation was launched using `arm_gazebo.launch`.

```
$ roslaunch arm_gazebo arm_gazebo.launch
```

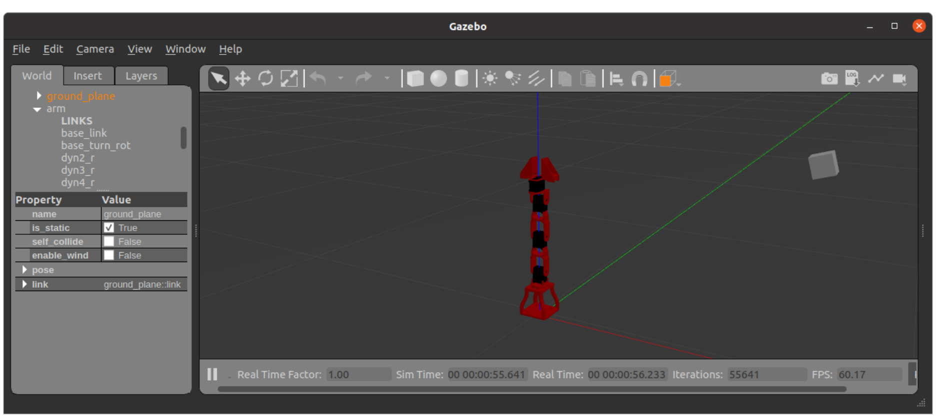

Figure 5:

Subsequently, the presence of the image topic was verified using `rqt_image_view` command from another terminal as shown in figure 6, confirming that the camera sensor was correctly integrated into the robot's simulation environment.

```
$ rqt_image_view
```

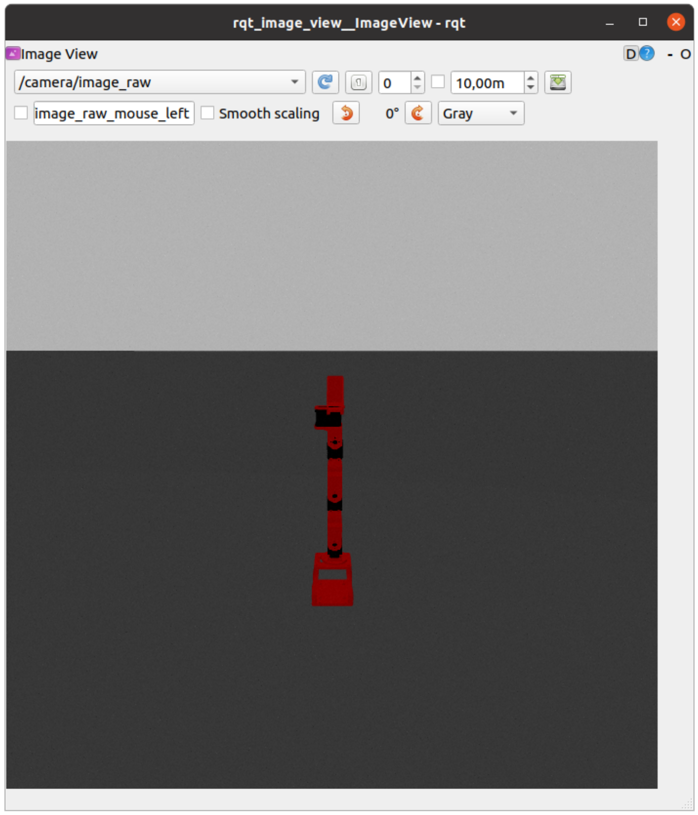

Figure 6: command: `rqt_image_view`

4 Create a ROS Publisher Node

4a Create an `arm_controller` package with a ROS C++ node named `arm_controller` and modifie CMakeLists.txt

I started by creating a ROS package named `arm_controller` using the following command:

```
$ catkin_create_pkg arm_controller roscpp sensor_msgs std_msgs
```

This command generates the necessary package structure with dependencies on `roscpp, sensor_msgs`, and `std_msgs`. Then I created the `arm_controller.cpp` node inside it.

In the CMakeLists.txt file of the `arm_controller` package, I made the following modifications:

- Uncommented the `add_executable` line to specify the name of our executable node and its path, which is `arm_controller src/arm_controller.cpp`.
- Uncommented the `target_link_libraries` line to link the necessary libraries, and wrote inside this function the name of the controller node.

4b Creating a Subscriber in the cpp file

I created a subscriber that listens to the `joint_states` topic, which contains data of type `sensor_msgs/JointState`. This topic provides information about the current joint positions of the robot. Then I defined a callback function named `jointStatesCallback` that is called whenever new data is received on the `joint_states` topic. Inside this function, we can access the current joint positions and perform any necessary operations. Below is the .cpp file:

```
red#redinclude <ros/ros.h>
red#redinclude <sensor_msgs/JointState.h>
red#redinclude <std_msgs/Float64.h>

redvoid jointStateCallback(redconst sensor_msgs::JointState::ConstPtr&
    joint_state) {
   ROS_INFO("\nReceived joint positions:");
    redfor (size_t i = 0; i < joint_state->position.size(); i++) {
        ROS_INFO("Joint %zu: %f", i, joint_state->position[i]);
    }
}

 redint main(redint argc, redchar** argv) {
    ros::init(argc, argv, "arm_controller");
    ros::NodeHandle nh;
    ros::Rate loop_rate(10);


blue    blue//blue blueCreateblue blueablue bluesubscriberblue bluetoblue bluereceive
        blue bluejointblue bluestateblue blueinformation
    ros::Subscriber joint_state_sub = nh.subscribe("joint_states", 10,
        jointStateCallback);

blue    blue//blue blueImplementblue blueanyblue bluenecessaryblue bluecontrolblue
        bluelogic
    redwhile (ros::ok())
  {
    ...
    loop_rate.sleep();
  }

    ros::spin();
    redreturn 0;
}
```

4c Creating Publishers

In the same cpp file I created publishers that write desidered joint position commands onto the controllers' `/command` topics. These commands are of type `std_msgs/Float64`. Each publisher is associated with a specific joint controller. Below is the .cpp file concerning the publisher part.

```
blue//blue blueCreateblue blueablue bluepublisherblue bluetoblue bluesendblue
    bluejointblue bluepositionblue bluecommandsblue blueforblue blueeachblue bluejoint
    blue.
    ros::Publisher joint0_pub = nh.advertise<std_msgs::Float64>("/arm/
        j0p_controller/command", 1);
    ros::Publisher joint1_pub = nh.advertise<std_msgs::Float64>("/arm/
        j1p_controller/command", 1);
    ros::Publisher joint2_pub = nh.advertise<std_msgs::Float64>("/arm/
        j2p_controller/command", 1);
    ros::Publisher joint3_pub = nh.advertise<std_msgs::Float64>("/arm/
        j3p_controller/command", 1);
```

```
blue    blue//blue blueImplementblue blueanyblue bluenecessaryblue bluecontrolblue
        bluelogic
    redwhile (ros::ok())
  {

blue   blue//blue blueGenerateblue blueablue bluecommandblue blueforblue blueeachblue
       bluejoint
    std_msgs::Float64 joint0_command;
    std_msgs::Float64 joint1_command;
    std_msgs::Float64 joint2_command;
    std_msgs::Float64 joint3_command;

    joint0_command.data =-5.0; blue     blue//blue bluerotateblue bluetheblue
        bluearmblue blueclockwise
    joint1_command.data = 0.5;
    joint2_command.data = -1.2;
    joint3_command.data = -1;

blue    blue//blue bluepublishblue bluetheblue bluecommand
    joint0_pub.publish(joint0_command);
    joint1_pub.publish(joint1_command);
    joint2_pub.publish(joint2_command);
    joint3_pub.publish(joint3_command);

blue    blue//blue blueWaitblue blueasblue bluelongblue blueasblue bluenecessaryblue
        bluetoblue bluemaintainblue bluetheblue bluedesiredblue bluefrequency

    loop_rate.sleep();
  }
```

Upon running the `arm_controller` node by writing from terminal ”`rosrun arm_controller arm_controller` and observing the Gazebo simulation, it is possible to confirm the effective operation of our setup. The robot's arm moved as expected, responding to the joint position commands we published; the final position of the robotic arm is shown in figure 6.

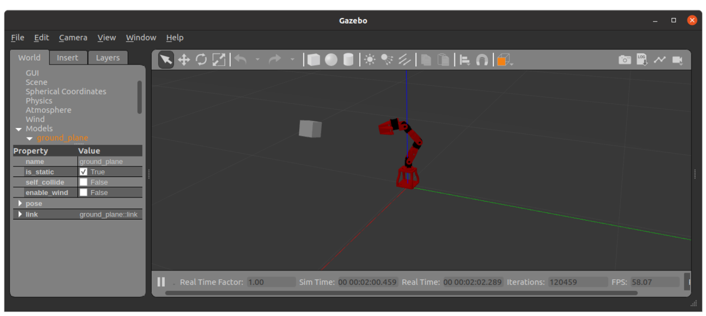

Figure 7: final arm position

To visualize Ros's graph structure, the following command can be executed from another terminal, the output is shown in figure 8.

```
$ rqt_graph
```

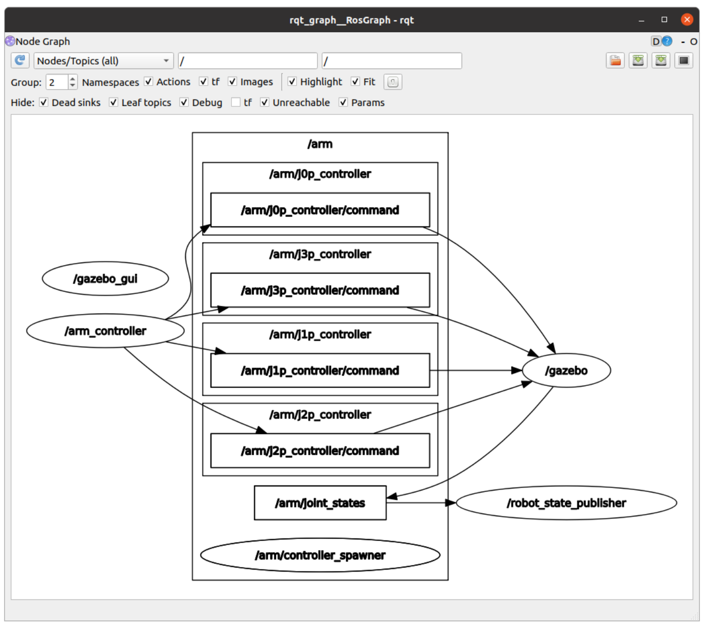

Figure 8: output command: `rqt_graph`