



## Robot Manipulation

---

### Lab Report

# Robot Manipulation with ROS

---

No.	Full Name	Student ID
1	Moses Chuka Ebere	u1985468
2	Joseph Oloruntoba Adeola	u1988552

Instructor: Sr. Ciurana Ferragutcases, Albert

# 1 Introduction

Robot Operating System (ROS) is a widely adopted open-source software framework for robotics development. It provides a comprehensive suite of tools, libraries, and packages that facilitate the design, development, and control of complex robotic systems. The modular architecture of ROS allows for the creation of sophisticated robotic systems through the integration of individual components and services into a unified system.

ROS supports a broad range of robotic hardware and platforms, including mobile robots, manipulator arms, and unmanned aerial vehicles. The framework includes libraries for robotic algorithms, simulation, visualization, and communication, enabling the creation and control of robots with ease. Additionally, ROS features middleware components that manage the communication between different system components and the flow of data between sensors, actuators, and controllers.

ROS has a thriving community of developers and users, who actively contribute to its growth and improvement. This has established ROS as a valuable resource for those in the field of robotics, including researchers, engineers, and students. The comprehensive and flexible framework provided by ROS makes it an indispensable tool for advancing robotics research and development.

## 1.1 Objective

The objective of this lab is to utilize the Robot Operating System (ROS) framework to design, visualize, and simulate a four-degrees-of-freedom SCARA robot model. The lab requires the creation of a model of the SCARA robot which is an (RRPR manipulator) using the Unified Robot Description Format (URDF), and the visualization of this model in RViz. The urdf model is to be designed following the Denavit-Hartenberg table of the manipulator in fig.x. Additionally, the lab involves simulating the robot's behavior in Gazebo's physics engine, allowing for the demonstration of its capabilities in real-world scenarios. The goal of the lab is to provide a hands-on experience with the tools and libraries available within the ROS ecosystem and learn how they can be applied to robot manipulation.

## 1.2 Definition of Terms

1. **urdf**: The Unified Robot Description Format (URDF) is a standard file format for representing robot models in ROS. It is used to describe a robot's kinematic and dynamic properties, including geometry, joint types, links, and actuators. The URDF model is represented as an XML file that provides information on the visual appearance of the robot, including size, color, and texture. This information is used by tools like RViz to provide a visual representation of the robot, which is helpful for debugging and testing. The URDF format is a crucial component of the ROS framework, used for simulation and visualization of robots.

Some urdf tags include

- **<robot>**: This tag is the top-level tag that specifies the start of the URDF file and contains information about the entire robot.
- **<<link>**: This tag defines a single link in the robot and contains information about its geometry, mass, and visual appearance.
- **<joint>**: This tag defines a single joint in the robot, specifying the type of joint (e.g., revolute, continuous, etc.), its limits, and the links that it connects.
- **<visual>**: This tag defines the visual appearance of a link and can include information about its geometry, color, and texture.

- **<inertial>**: This tag defines the inertial properties of a link, including its mass and moment of inertia.
  - **<origin>**: This tag specifies the position and orientation of a link or joint in the world frame.
  - **<geometry>**: This tag defines the geometry of a link or collision shape and can include information about primitives (e.g., boxes, spheres, cylinders) or meshes.
  - **<material>**: This tag defines the material properties of a link, such as its color and texture.
2. **Xacro**: Xacro (XML Macros) is an XML file format used in ROS to simplify the creation of complex URDF models. Xacro allows the definition of macros, or reusable code blocks, to parameterize properties of robots, links, and joints. It supports inheritance and composition, reducing duplication of code and improving modularity, and provides built-in macros for common robot components. Xacro files are processed into standard URDF files using the xacro command-line tool and can be loaded into RViz or Gazebo for visualization and simulation. It provides a flexible and powerful way to define complex robot models in ROS.
  3. **RViz**: RViz (ROS Visualization) is a 3D visualizer for the Robot Operating System (ROS) that allows for the visualization and interpretation of data from robots and their environments. RViz can display data from various ROS topics, including sensor data, robot geometry, and navigation maps. The data is displayed in an interactive 3D environment, allowing users to inspect and understand the data in a more intuitive manner. RViz also provides tools for visualizing data in different coordinate frames, and can be used to visualize robot models created using the Unified Robot Description Format (URDF). With its powerful visualization capabilities, RViz is an essential tool for debugging and testing robotic systems, and for visualizing the results of simulations and experiments.
  4. **Gazebo**: Gazebo is a 3D simulation engine for robots, environments, and physical systems. It enables users to create virtual environments that closely resemble the real world and test and evaluate robots in a safe and controlled environment. Gazebo is integrated with the Robot Operating System (ROS) and supports physics simulation, including gravity, friction, and collisions. This integration allows for seamless communication between Gazebo and other ROS tools, enabling users to validate their robotic systems without the need for physical hardware.
  5. **Denavit-Hartenberg Table**: The Denavit-Hartenberg (DH) parameters table is a convention for representing the kinematic structure of a robot. The DH table provides a systematic way to describe the relative position and orientation of each link in a robot with respect to the preceding link. The DH parameters table is used in robotics to model the forward and inverse kinematics of robots, as well as to calculate the robot's Jacobian matrix, which is used to control the robot's motion.

Each row in the DH table represents a single joint in the robot, and contains four parameters:

- **$d$** : The distance along the previous link's z-axis to the common normal of the two z-axes.
- **$\alpha$** : The angle between the z-axis of the current link and the z-axis of the next link, measured in the xy-plane.
- **$\alpha$** : The angle between the x-axis of the current link and the x-axis of the next link, measured about the z-axis.
- **$\theta$** : By using these parameters and the concept of homogeneous transforms, the relative position and orientation of each link in the robot can be determined and used to model the robot's kinematic structure.

## 2 Methodology

### 2.1 Systematic urdf definition of the SCARA robot

The SCARA RRPR (Revolute-Revolute-Prismatic-Revolute) robot can be defined systematically in URDF by using its Denavit-Hartenberg (DH) parameters. The DH parameters table includes information on the relative orientation and position of each joint in the robot. This information can be used to determine the transformation matrices between each link in the robot. The transformation matrices can then be used to define the link and joint geometries in the URDF file. According to the DH convention, this transformation is given by:

$${}^{i-1}A_i = R_Z(\theta).T_Z(d).T_X(a).R_Z(\alpha)$$

Fig.1 below shows the scara manipulator.

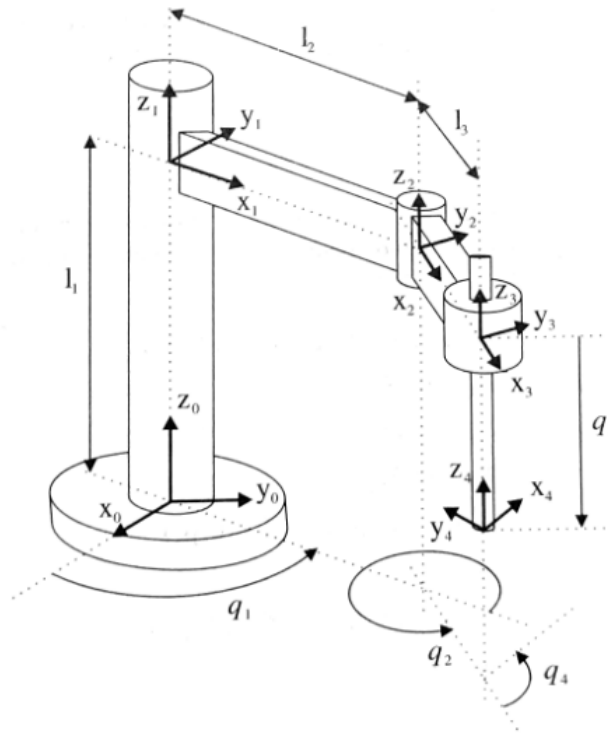


Figure 1: Scara robot

Thus, the first step towards modeling the robot is to first find its DH parameters. It should be noted that the image above provided in the lab guide has not correctly placed the frames of the robot. Thus, we resolved to place the axes as represented in the RViz model which will be shown subsequently (see fig.x). The DH table obtained by placing the axes this way is shown below:

Point	$\theta$	$d$	$a$	$\alpha$
1	$q_1$	$l_1 = 0.35$	$l_2 = 0.2$	0
2	$q_2$	0	$l_3 = 0.2$	$\pi$
3	0	$q_3 = 0.34$	0	0
4	$q_4$	0	0	0

Table 1: Denavit-Hartenberg table.

The values of  $l_1$ ,  $l_2$ ,  $l_3$ ,  $q_3$  are the lengths of the link assigned to us (group 3) in the lab. The next step after obtaining the DH table is to incorporate its parameters into a URDF model by defining them as Xacro properties. This way, the parameters can be treated as variables in the code, making it easier to modify and update the model. The following xml code shows the necessary Xacro properties in XML form, omitting the DH parameters with a value of zero.

```

1 <?xml version="1.0"?>
2 <robot xmlns:xacro="http://www.ros.org/wiki/xacro" name="MJBOT">
3
4     <!-- thickness/radius of model links -->
5     <xacro:property name="thickness" value="0.025"/>
6     <xacro:property name="joint_thickness" value="0.035"/>
7
8
9     <!-- prismatic link thickness -->
10    <xacro:property name="prismatic_link_radius" value="0.012"/>
11
12    <!-- theta values from the DH table -->
13    <xacro:property name="theta1" value="0"/>
14    <xacro:property name="theta2" value="0"/>
15    <xacro:property name="theta4" value="0"/>
16
17    <!-- d parameter values from the DH table -->
18    <xacro:property name="d1" value="0.35"/>
19    <xacro:property name="d2" value="0"/>
20    <xacro:property name="d3" value="0.34"/>
21    <xacro:property name="d4" value="0"/>
22
23    <!-- a parameter values from the DH table -->
24    <xacro:property name="a1" value="0.2"/>
25    <xacro:property name="a2" value="0.2"/>
26
27
28    <!-- alpha parameter values from the DH table -->
29    <xacro:property name="alpha2" value="3.14"/>

```

Next, material, mass, and inertia properties are also defined for global use. To give different colors to links

in the urdf. A macro named "default\_inertia" is defined which sets the mass and moment of inertia for the links. This macro can be called later in the code to specify the inertia of the links.

```

1      <!-- mass for inertia definition -->
2      <xacro:property name="mass1" value="0.1"/>
3
4      <!-- inertia definition using macros -->
5      <xacro:macro name="default_inertia" params="mass">
6          <inertial>
7              <mass value="${mass}"/>
8              <inertia ixx="0.1" ixy="0.0" ixz="0.0" iyy="0.1" iyz="0.0" izz="0.1"/>
9          </inertial>
10     </xacro:macro>
11
12     <!-- material definition -->
13     <material name="dark_grey">
14         <color rgba="0.5 0.5 0.5 1.0"/>
15     </material>
16     <material name="light_grey">
17         <color rgba="0.8 0.8 0.8 1.0"/>
18     </material>

```

The advantage of parameterization is the ease with which we can easily make global changes to the robot model without having to go through the entire urdf file.

After parameterization, we proceeded to construct the robot model by defining a world link for Gazebo and creating the base link of the robot. The code is organized and properly commented on to clearly indicate each component's purpose. It should be noted that some temporary links and fixed joints have been defined while writing the URDF code because it helps to represent the kinematic chain of the robot accurately. Each link in the chain has a specific pose and orientation in space relative to its parent link. By defining temporary links, we can break down the complex structure of the robot into simpler, manageable segments, which makes it easier to describe the overall geometry and kinematics of the robot. The number of revolute and prismatic joints designed was based on the DH table obtained.

```

1      <!-- fixed base joint -->
2      <joint name="fixed1" type="fixed">
3          <parent link="world"/>
4          <child link="base_link"/>
5      </joint>
6
7
8      <!-- First link definition -->
9      <link name="tmp_11">
10         <xacro:default_inertia mass="${mass1}"/>
11         <visual>
12             <geometry>

```

```

13         <cylinder length="{d1}" radius="{thickness}"/>
14     </geometry>
15     <origin rpy="0 0 0" xyz="0 0 {d1/2}"/>
16     <material name="light_grey"/>
17 </visual>
18 </link>
19
20 <joint name="base_joint" type="fixed">
21     <parent link="base_link"/>
22     <child link="tmp_11"/>
23     <origin rpy="0 0 0" xyz="0 0 0"/>
24 </joint>
25
26
27
28
29 <link name="tmp_12">
30     <xacro:default_inertia mass="{mass1}"/>
31     <visual>
32         <geometry>
33             <cylinder length="0.1" radius="{joint_thickness}"/>
34         </geometry>
35         <origin rpy="0 0 0" xyz="0 0 0"/>
36         <material name="dark_grey"/>
37     </visual>
38 </link>
39
40
41 <!-- a fixed joint added to allow us represent the kinematic chain accurately -->
42 <joint name="fixed2" type="fixed">
43     <parent link="tmp_11"/>
44     <child link="tmp_12"/>
45     <origin rpy="0 0 0" xyz="0 0 {d1}"/>
46 </joint>
47
48
49 <!-- first revolute joint -->
50 <joint name="theta1_joint" type="revolute">
51     <parent link="tmp_12"/>
52     <child link="L1"/>
53     <axis xyz="0 0 1"/>
54     <limit effort="1000.0" lower="-1.57" upper="1.57" velocity="0.5"/>
55     <origin rpy="0 0 {theta1}" xyz="0 0 0"/>
56     <dynamics damping="0.7"/>

```

```

57     </joint>
58
59     <link name="L1">
60         <xacro:default_inertia mass="${mass1}" />
61         <visual>
62             <geometry>
63                 <cylinder length="${a1}" radius="${thickness}" />
64             </geometry>
65             <origin rpy="0 ${pi/2} 0" xyz="${a1/2} 0 0" />
66             <material name="light_grey" />
67         </visual>
68     </link>
69
70
71
72
73     <link name="tmp_13">
74         <xacro:default_inertia mass="${mass1}" />
75         <visual>
76             <geometry>
77                 <cylinder length="0.1" radius="0.035" />
78             </geometry>
79             <material name="dark_grey" />
80         </visual>
81     </link>
82
83     <joint name="a1" type="fixed">
84         <parent link="L1" />
85         <child link="tmp_13" />
86         <origin rpy="0 0 0" xyz="${a1} 0 0" />
87     </joint>
88
89
90     <link name="L2">
91         <xacro:default_inertia mass="${mass1}" />
92         <visual>
93             <geometry>
94                 <cylinder radius="${thickness}" length="${a2}" />
95             </geometry>
96             <origin xyz="${a2/2} 0.0 0.0" rpy="0.0 ${pi/2} 0.0" />
97             <material name="light_grey" />
98         </visual>
99     </link>
100

```



```

101
102     <!-- second joint definition/ second revolute joint -->
103     <joint name="theta2_joint" type="revolute">
104         <parent link="tmp_13"/>
105         <child link="L2"/>
106         <axis xyz="0 0 1"/>
107         <limit effort="1000.0" lower="-1.57" upper="1.57" velocity="0.5"/>
108         <origin rpy="0 0 0" xyz="0 0 0"/>
109     </joint>
110
111
112     <link name="L3">
113         <xacro:default_inertia mass="{mass1}"/>
114         <visual>
115             <geometry>
116                 <cylinder length="0.1" radius="0.035"/>
117             </geometry>
118             <material name="dark_grey"/>
119         </visual>
120     </link>
121
122     <joint name="a2" type="fixed">
123         <parent link="L2"/>
124         <child link="L3"/>
125         <origin rpy="0 0 0" xyz="{a2} 0 0"/>
126     </joint>
127
128     <link name="tmp_31">
129         <xacro:default_inertia mass="{mass1}"/>
130         <visual>
131             <geometry>
132                 <cylinder length="{d3}" radius="{prismatic_link_radius}"/>
133             </geometry>
134             <origin rpy="0 0 0" xyz="0 0 {d3/2}"/>
135             <material name="light_grey"/>
136         </visual>
137     </link>
138
139
140     <!-- third joint definition (the only prismatic joint) -->
141     <joint name="d3_joint" type="prismatic">
142         <parent link="L3"/>
143         <child link="tmp_31"/>
144         <axis xyz="0 0 -1"/>

```

```

145     <limit effort="1000.0" lower="0.06" upper="${d3}" velocity="0.5"/>
146     <origin rpy="0 0 0" xyz="0 0 0"/>
147 </joint>
148
149
150 <!-- definition of gripper -->
151 <link name="gripper">
152     <xacro:default_inertia mass="${mass1}"/>
153     <visual>
154         <geometry>
155             <box size="0.05 0.05 0.01"/>
156         </geometry>
157         <material name="dark_grey"/>
158         <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0"/>
159     </visual>
160 </link>
161
162 <!-- gripper first finger -->
163 <link name="gripper_finger1">
164     <xacro:default_inertia mass="${mass1}"/>
165     <visual>
166         <geometry>
167             <box size="0.01 0.05 0.05"/>
168         </geometry>
169         <material name="dark_grey"/>
170         <origin xyz="-0.03 0.0 -0.015" rpy="0.0 0.0 0.0"/>
171     </visual>
172 </link>
173
174
175 <!-- gripper second finger -->
176 <link name="gripper_finger2">
177     <xacro:default_inertia mass="${mass1}"/>
178     <visual>
179         <geometry>
180             <box size="0.01 0.05 0.05"/>
181         </geometry>
182         <material name="dark_grey"/>
183         <origin xyz="0.015 0.0 -0.015" rpy="0.0 0.0 0.0"/>
184     </visual>
185 </link>
186
187
188 <!-- fourth joint/ third revolute joint. -->

```

```

189 <joint name="theta4_joint" type="revolute">
190   <parent link="tmp_31"/>
191   <child link="gripper"/>
192   <axis xyz="0 0 1"/>
193   <limit effort="1000.0" lower="-3.1416" upper="3.1416" velocity="0.5"/>
194   <origin rpy="0 0 ${theta4}" xyz="0 0 0"/>
195 </joint>
196
197 <!-- fixed joints to attach the fingers of the gripper -->
198 <joint name="grip_1" type="fixed">
199   <parent link="gripper"/>
200   <child link="gripper_finger1"/>
201   <origin rpy="0 0 0" xyz="0 0 0"/>
202 </joint>
203 <joint name="grip_2" type="fixed">
204   <parent link="gripper"/>
205   <child link="gripper_finger2"/>
206   <origin rpy="0 0 0" xyz="0.01 0 0"/>
207 </joint>

```

On completing the robot visual model, specific gazebo-related xml tags were added to the urdf file to allow the model to be simulated in the Gazebo simulation environment. These tags include the material type color for each link, joint transmission tags, and gazebo plugin tags. It should be noted that transmission in Gazebo is only added to moveable joints. The codes in the following listing show the gazebo xml tags added to the urdf file.

```

1 <!-- Gazebo materials definition -->
2 <gazebo reference= "L1">
3   <material>Gazebo/Reflection</material>
4 </gazebo>
5 <gazebo reference= "L2">
6   <material>Gazebo/Reflection</material>
7 </gazebo>
8 <gazebo reference= "L3">
9   <material>Gazebo/Black</material>
10 </gazebo>
11 <gazebo reference= "base_link">
12   <material>Gazebo/Black</material>
13 </gazebo>
14 <gazebo reference= "tmp_11">
15   <material>Gazebo/Reflection</material>
16 </gazebo>
17 <gazebo reference= "tmp_12">
18   <material>Gazebo/Black</material>
19 </gazebo>

```

```

20 <gazebo reference= "tmp_13">
21   <material>Gazebo/Black</material>
22 </gazebo>
23 <gazebo reference= "tmp_31">
24   <material>Gazebo/Reflection</material>
25 </gazebo>
26 <gazebo reference= "gripper">
27   <material>Gazebo/Cyan</material>
28 </gazebo>
29 <gazebo reference= "gripper_finger1">
30   <material>Gazebo/Cyan</material>
31 </gazebo>
32 <gazebo reference= "gripper_finger2">
33   <material>Gazebo/Cyan</material>
34 </gazebo>
35
36
37 <!-- Gazebo transmission definition -->
38 <transmission name="theta1_trans">
39   <type>transmission_interface/SimpleTransmission</type>
40   <joint name="theta1_joint">
41     <hardwareInterface>hardware_interface/PositionJointInterface</hardwareInterface>
42   </joint>
43   <actuator name="theta1_motor">
44     <mechanicalReduction>1.0</mechanicalReduction>
45     <hardwareInterface>hardware_interface/PositionJointInterface</hardwareInterface>
46   </actuator>
47 </transmission>
48
49 <transmission name="theta2_trans">
50   <type>transmission_interface/SimpleTransmission</type>
51   <joint name="theta2_joint">
52     <hardwareInterface>hardware_interface/PositionJointInterface</hardwareInterface>
53   </joint>
54   <actuator name="theta2_motor">
55     <mechanicalReduction>1.0</mechanicalReduction>
56     <hardwareInterface>hardware_interface/PositionJointInterface</hardwareInterface>
57   </actuator>
58 </transmission>
59
60
61 <transmission name="d3_trans">
62   <type>transmission_interface/SimpleTransmission</type>
63   <joint name="d3_joint">

```

```

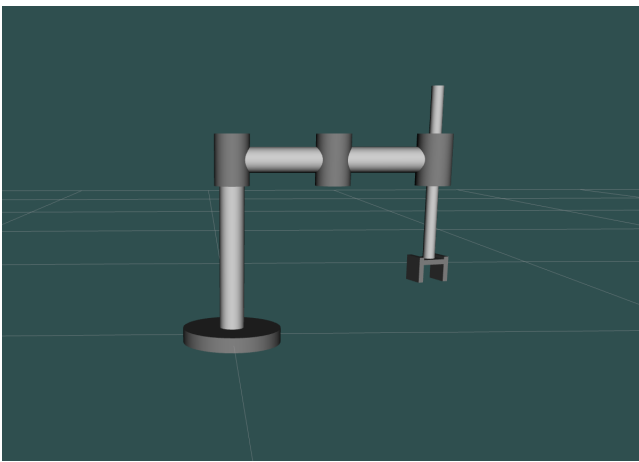
64     <hardwareInterface>hardware_interface/PositionJointInterface</hardwareInterface>
65 </joint>
66 <actuator name="d3_motor">
67     <mechanicalReduction>1.0</mechanicalReduction>
68     <hardwareInterface>hardware_interface/PositionJointInterface</hardwareInterface>
69 </actuator>
70 </transmission>
71
72
73
74 <transmission name="theta4_trans">
75     <type>transmission_interface/SimpleTransmission</type>
76     <joint name="theta4_joint">
77         <hardwareInterface>hardware_interface/PositionJointInterface</hardwareInterface>
78     </joint>
79     <actuator name="theta4_motor">
80         <mechanicalReduction>1.0</mechanicalReduction>
81         <hardwareInterface>hardware_interface/PositionJointInterface</hardwareInterface>
82     </actuator>
83 </transmission>
84
85
86 <!-- Gazebo plugin for ROS Control -->
87 <gazebo>
88     <plugin name="gazebo_ros_control" filename="libgazebo_ros_control.so">
89         <robotNamespace>MJBot</robotNamespace>
90     </plugin>
91 </gazebo>
92 </robot>

```

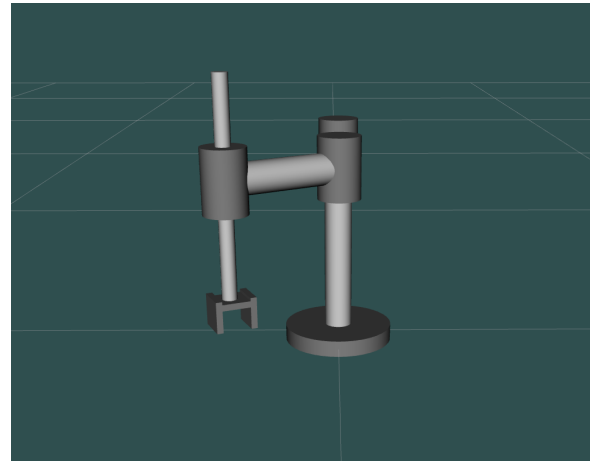
- **Transmission:** The transmission component in Gazebo defines the relationship between joint effort (torque or force) and joint velocity in a robot's limb. It determines the conversion between the actuator's output and the joint's motion.
- **Hardware Interface:** The hardware interface component in the code acts as a bridge between the physical hardware and the simulation environment. It allows a robot's real hardware to be controlled and monitored within the Gazebo environment.
- **Mechanical Reduction:** Mechanical reduction refers to the gear reduction between the actuator and the joint in a robot's limb. It determines the ratio between the actuator's output and the joint's motion. The mechanical reduction can be specified in Gazebo using the transmission component.
- **Actuator Tag:** The actuator tag in defines the type of actuator that is being used to drive a joint. It can be set to either "position" or "velocity" control and it specifies the control mode for the joint. The actuator tag can also define the maximum effort (torque or force) that can be applied to the joint.

### 3 Robot model Visualization

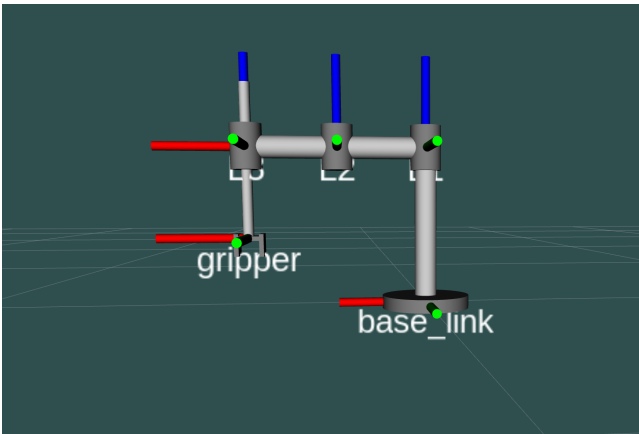
To visualize the robot description using rViz, we parsed and load the URDF to the rosparm database and published a value for each non-fixed joint. We also made sure to publish the robot state of all tf transformations between links and ran the visualization interface. The robot model can be visualized in rViz by running the command `roslaunch scara_robot robot_rviz.launch`. To enhance the visual representation of the robot description in rViz, we modified the background color and alpha parameters in the `urdf_view.rviz` file provided in the lab. The following images in fig... show the Scara robot model in rViz environment.



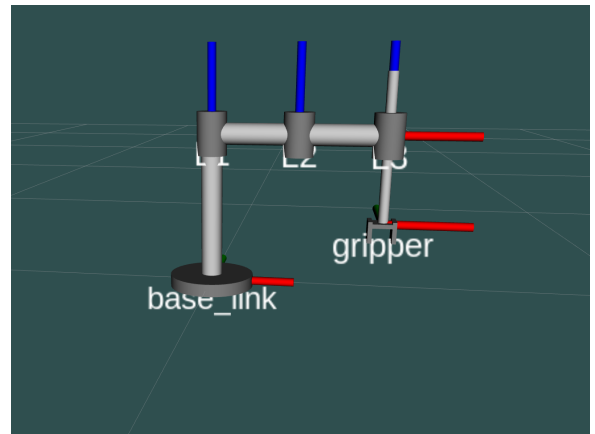
(a) Scara robot right side view in rViz



(b) Scara robot angular view in rViz

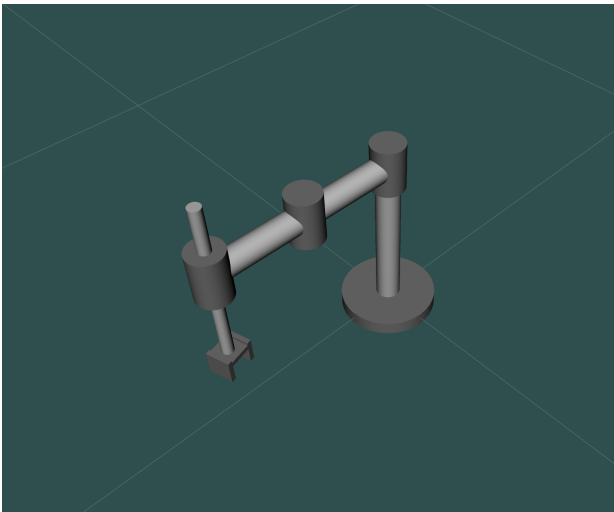


(c) Scara robot side view with axes in rViz

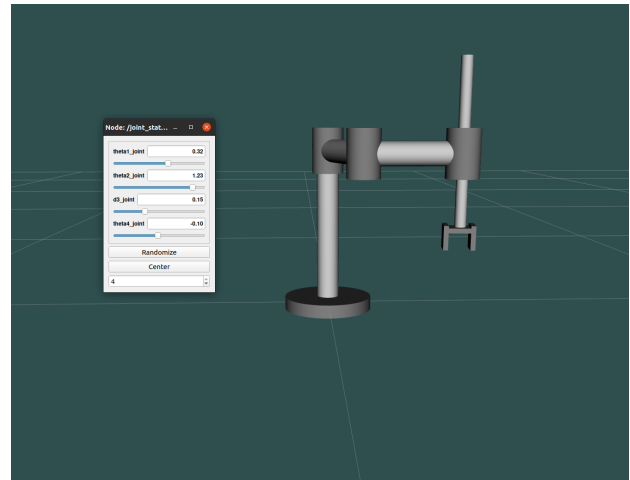


(d) Scara robot side view with axes in rViz

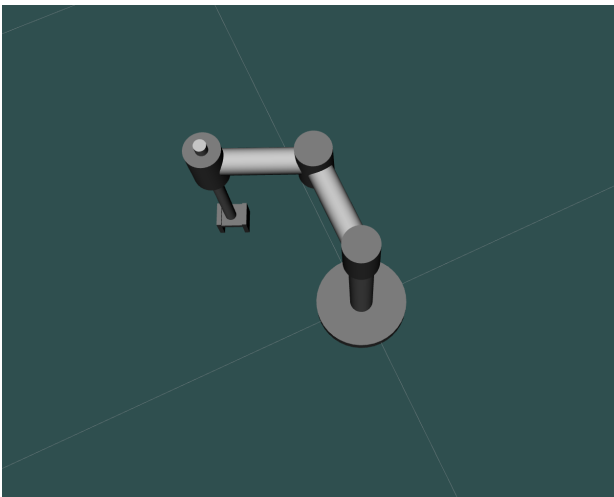
Figure 2: Scara robot model in rViz



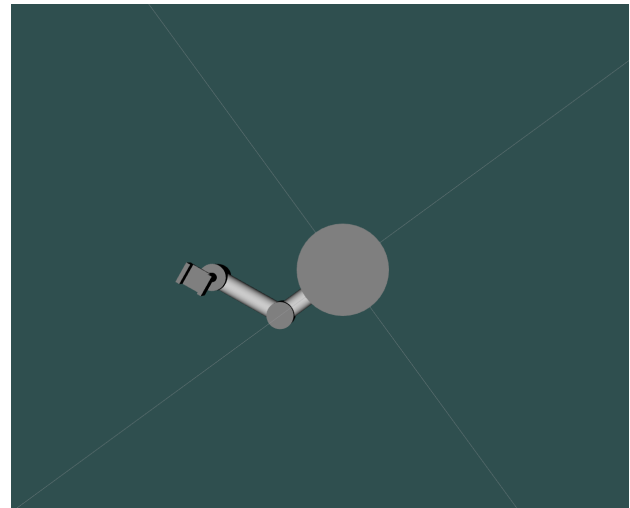
(a) Side view



(b) Angular side view with GUI



(c) Top view



(d) Bottom view

Figure 3: Scara robot view in rViz

### 3.1 State Publishers

The joint state publisher node reads the robot model description, identifies all joints, and publishes their values to the non-fixed joints via a GUI. The GUI displays the state of the robot's joints that were set through the interface, as shown in Fig.3(b) while the Robot State Publisher takes the current joint states of the robot and publishes the 3D position of each link through the URDF-generated kinematics tree. The robot's 3D pose is published as a ROS tf that conveys the relationship between the robot's coordinate frames.

```

1 <launch>
2   <arg name="gui" default="true" />
3
4   <!-- Parse URDF -->
5   <param name="robot_description" command="$(find xacro)/xacro $(find
      scara_robot)/urdf/MJBot.urdf"/>

```

```

6
7  <!-- Joint State Publisher -->
8  <node name="joint_state_publisher_gui" pkg="joint_state_publisher_gui"
9  type="joint_state_publisher_gui" />
10 <node unless="$(arg gui)" name="joint_state_publisher" pkg="joint_state_publisher"
11 type="joint_state_publisher" />
12
13 <!-- Robot State Publisher -->
14 <node name="robot_state_publisher" pkg="robot_state_publisher"
15 type="robot_state_publisher" />
16
17 <!-- rViz visualization tool -->
18 <node name="rviz" pkg="rviz" type="rviz" args="-d $(find
19   scara_robot)/config/urdf_view.rviz" required="true" />
20 </launch>

```

$\theta_{1_{joint}}$	$\theta_{2_{joint}}$	d3	$\theta_{4_{joint}}$
$\frac{\pi}{2}$	$-\frac{\pi}{3}$	0.1	$\pi$

Table 2: Given joint parameters

By setting the values of the robot's joint through the GUI in rViz to the given values in Table.2, and checking the value of the transformation obtained from the base link to the gripper through the command **roslaunch tf\_echo base\_link gripper**, the following values were obtained:

```

Cjoseph@joseph-ROG-Strix-G513RM-G513RM:~/catkin_ws$ roslaunch tf_echo base_link gripper
t time 1675184661.085
Translation: [0.173, 0.301, 0.247]
Rotation: in Quaternion [0.000, 0.000, 0.965, -0.261]
          in RPY (radian) [0.000, -0.000, -2.614]
          in RPY (degree) [0.000, -0.000, -149.775]

```

Figure 4: roslaunch tf\_echo base\_link gripper

Solving the  ${}^0A_4$  symbolic matrix manually using the parameters in the DH table and the equation  ${}^{i-1}A_i$

$${}^{i-1}A_i = \begin{bmatrix} c\theta_i & -c\alpha_i s\theta_i & s\alpha_i s\theta_i & a_i c\theta_i \\ s\theta_i & c\alpha_i c\theta_i & b_3 & a_i s\theta_i \\ 0 & s\alpha_i & c\alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

We obtain

$${}^0A_4 = \begin{bmatrix} -0.8660254 & -0.5 & 0 & 0.17320508 \\ -0.5 & 0.8660254 & 0 & 0.3 \\ 0 & 0 & -1 & 0.25 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Comparing the matrix  ${}^0A_4$  to the model's results, we observe that the translation (last column of  ${}^0A_4$ ) is similar,



with a small deviation due to the challenges of accurately setting the robot's joint values through the GUI. To confirm the angle match, we must convert the rotation portion of  ${}^0A_4$  to RPY angles.

$$R = \begin{bmatrix} -0.8660254 & -0.5 & 0 \\ -0.5 & 0.8660254 & 0 \\ 0 & 0 & -1 \end{bmatrix}$$

This conversion can be done by using the following formula:

$$\begin{aligned} roll &= \text{atan2} \left( \frac{r_{32}}{r_{33}} \right) \\ pitch &= \text{atan2} \left( \frac{r_{31}}{\sqrt{r_{32}^2 + r_{33}^2}} \right) \\ yaw &= \text{atan2} \left( \frac{r_{21}}{r_{11}} \right) \end{aligned}$$

Thus, by substituting the values from the rotation matrix into the formula, we obtain the RPY in radians:

$$\begin{aligned} roll &= 0 \\ pitch &= 0 \\ yaw &= -2.618 \end{aligned}$$

Comparing this result with the result from rViz in fig.4, we observe they are approximately equal. Again, the slight discrepancy is due to the difficulty in setting the robot's joint values through the GUI to the exact value we were asked to.

## 4 Simulation in Gazebo

A simulator is a crucial tool for evaluating and testing various aspects of robots before they are built and deployed in real-world scenarios. This helps to reduce the risk of failure, improve the design, and optimize performance. Simulation can also be used for training and testing algorithms, and to evaluate the effects of different conditions and parameters on the robot's behavior.

### 4.1 Spawning the Model

After visualizing a SCARA robot in rViz, simulating it in Gazebo can provide a more in-depth evaluation of its functionality and performance. With the properties required for simulating the model already defined in the urdf model, i.e material type, transmission, and a fixed joint from the world frame which serves as the Gazebo reference frame, the next task was to spawn the model. Spawning a model refers to the process of adding and instantiating objects within the simulation environment. To spawn the model in Gazebo, we specified the model's information such as its geometry, joint properties, and physical properties in a configuration file. We spawned the model in gazebo using the **gazebo.launch** file provided in the lab to spawn the model.

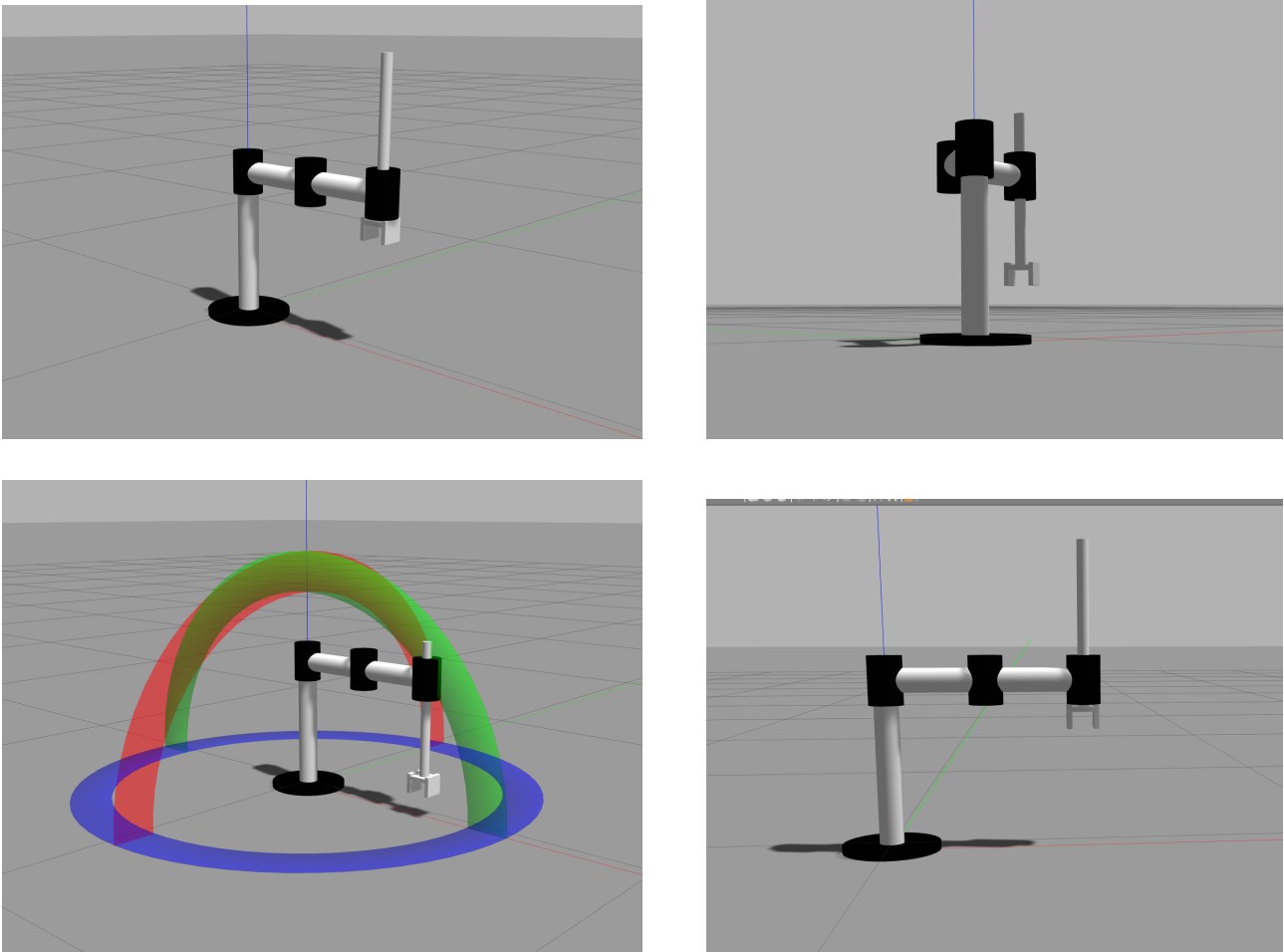


Figure 5: Scara robot model in Gazebo simulation environment

The **gazebo.launch** code is shown below:

```

1 <launch>
2 <!-- these are the arguments you can pass this launch file, for example
3 paused:=true -->
4 <arg name="paused" default="false"/>
5 <arg name="use_sim_time" default="true"/>
6 <arg name="gui" default="true"/>
7 <arg name="headless" default="false"/>
8 <arg name="debug" default="false"/>
9 <arg name="model" default="$(find scara_robot)/urdf/MJBot.urdf"/>
10
11 <!-- We resume the logic in empty_world.launch, changing only the name of the
12 world to be launched -->
13 <include file="$(find gazebo_ros)/launch/empty_world.launch">
14   <arg name="debug" value="$(arg debug)" />
15   <arg name="gui" value="$(arg gui)" />
16   <arg name="paused" value="$(arg paused)" />
17   <arg name="use_sim_time" value="$(arg use_sim_time)" />
18   <arg name="headless" value="$(arg headless)" />
19 </include>
20
21 <param name="robot_description" command="$(find xacro)/xacro $(arg model)" />
22
23 <!-- push robot_description to factory and spawn robot in gazebo -->
24 <node name="urdf_spawner" pkg="gazebo_ros" type="spawn_model" args="-x 0.0 -y 0.0 -z
25 0.0 -unpause -urdf -model robot -param robot_description" respawn="false"
26 output="screen" />
27
28 <node pkg="robot_state_publisher" type="robot_state_publisher"
29   name="robot_state_publisher">
30   <param name="publish_frequency" type="double" value="30.0" />
31 </node>
32 </launch>

```

## 4.2 Controllers

Controllers in Gazebo simulate the behavior of physical systems and control the movement of objects in the simulation environment. They are used to control the movement of robots and other entities in the simulation and provide realistic interactions with the environment. In order to interact with the scara robot in the simulation environment, we set-up a controllers that relates the desired action and the output from the actuators. This controller defines how we control each hardware interface. In this case, we set up two controllers, a joint\_state\_controller to publish the position of each joint and a joint\_group\_position\_controller to control the position of a group of joints. A joint state controller in Gazebo is a type of controller that is used to control the position, velocity, or effort of a joint in a robot model. The joint state controller can be used to control the joint values in real-time based on input commands, to provide a specific joint trajectory over time, or to

achieve a desired behavior for the robot. The joint state controller is implemented as a plugin in Gazebo and can be configured using a plugin file or in the robot model's SDF file while the joint group position controller is a type of robotic control system used to control the movement of joints in a robotic system within the Gazebo simulation environment. The controller takes in a desired joint position or trajectory and uses algorithms to generate control signals that drive the joints toward the desired position. **The two controllers were defined inside a yaml file called config/joints.yaml.**

```

1  MJBot:
2    joint_state_controller:
3      type: joint_state_controller/JointStateController
4      publish_rate: 50
5    joint_group_position_controller:
6      type: position_controllers/JointGroupPositionController
7      joints:
8        - theta1_joint
9        - theta2_joint
10       - d3_joint
11       - theta4_joint
12    theta1_joint:
13      pid: {p: 1, i: 0.01, d: 0, i_clamp: 1}
14    theta2_joint:
15      pid: {p: 1, i: 0.01, d: 0, i_clamp: 1}
16    d3_joint:
17      pid: {p: 1, i: 0.01, d: 0, i_clamp: 1}
18    theta4_joint:
19      pid: {p: 1, i: 0.01, d: 0, i_clamp: 1}

```

The joint controllers are only defined for the moveable joints where we have the actuators. In this case, there are only four moveable joints, three revolute and one prismatic.

After this, we call the spawned model and controllers in a launch file called *joints.launch*.

```

1  <launch>
2    <arg name="model" default="$(find scara_robot)/urdf/MJBot.urdf"/>
3    <arg name="rvizconfig" default="$(find scara_robot)/config/urdf_view.rviz"/>
4    <!-- <arg name="rvizconfig" default="$(find scara_robot)/config/urdf.rviz"/> -->
5
6
7    <include file="$(find scara_robot)/launch/gazebo.launch">
8      <arg name="model" value="$(arg model)" />
9    </include>
10
11    <!-- load joint controller configurations from YAML file to parameter server -->
12    <rosparam command="load" file="$(find scara_robot)/config/joints.yaml"/>
13

```

```

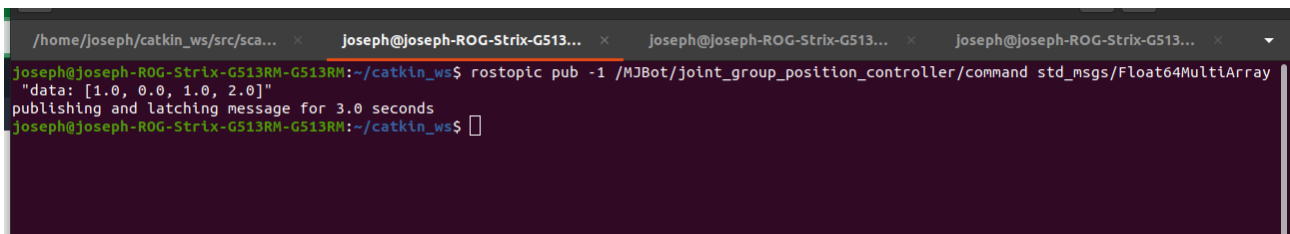
14 <!-- this is the new load controllers -->
15 <node name="MJBot_controller_spawner" pkg="controller_manager" type="spawner"
    respawn="false"
16 output="screen" ns= "MJBot" args="joint_group_position_controller
    joint_state_controller" />
17
18 </launch>

```

The joints.launch file allows us run the simulation environment with the robot. The following images in fig ... show the robot in the gazebo empty world environment.

### 4.3 Moving the robot

To move the robot, we have to publish joint commands to joint\_group\_position\_controller. First, we run **rostopic list** to see the topics we can publish to. In this case, the controller topic is: **/MJBot/joint\_group\_position\_controller/command**. So we publish the joint group command as a float64MultiArray datatype message to the controller. The following images in fig.7 8 show the robot executing the command specified in fig.6

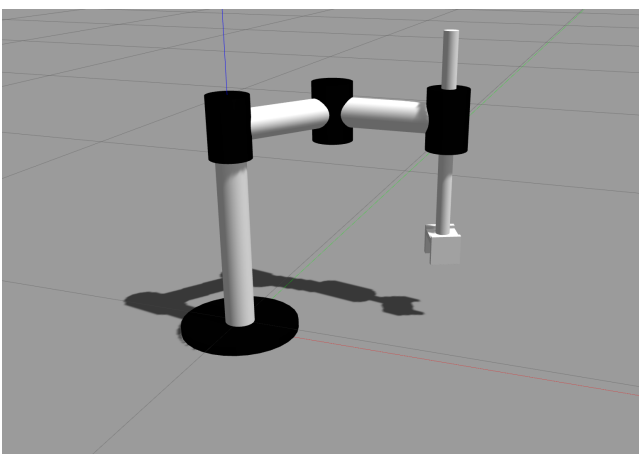


```

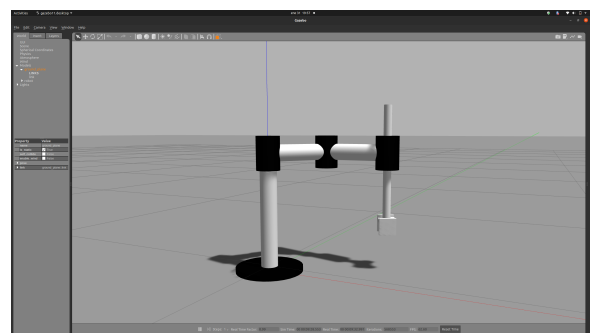
/home/joseph/catkin_ws/src/sca...  joseph@Joseph-ROG-Strix-G513...  joseph@Joseph-ROG-Strix-G513...  joseph@Joseph-ROG-Strix-G513...
joseph@joseph-ROG-Strix-G513RM-G513RM:~/catkin_ws$ rostopic pub -1 /MJBot/joint_group_position_controller/command std_msgs/Float64MultiArray
"data: [1.0, 0.0, 1.0, 2.0]"
publishing and latching message for 3.0 seconds
joseph@joseph-ROG-Strix-G513RM-G513RM:~/catkin_ws$

```

Figure 6: Publishing joint group position command



(a) Robot at the published state (zoomed)



(b) Robot at the published state in Gazebo environment

Figure 7: Robot at the published joint\_group\_position\_command state



Figure 8: Full view of robot at the commanded joint state

We can also retrieve the current value of each of the robot's joints by calling the `joint_state` controller. The following figure shows the command executed to retrieve the joint values of the robot based on its state in fig.8

```

/home/Joseph/catkin_ws/src/scara_robot/laun... x  Joseph@Joseph-ROG-Strix-G513RM: ~/... x  Joseph@Joseph-ROG-St
joseph@joseph-ROG-Strix-G513RM-G513RM:~/catkin_ws$ rostopic echo -n 1 /MJBot/joint_states
header:
  seq: 4302
  stamp:
    secs: 86
    nsecs: 327000000
  frame_id: ''
name:
  - d3_joint
  - theta1_joint
  - theta2_joint
  - theta4_joint
position: [0.2000000000579093, 0.999999999997842, -0.999999999991473, 1.999999999989733]
velocity: [5.790931496308154e-08, -2.1571226984251244e-10, 8.524294785263364e-10, -1.0274842449005146e-09]
effort: [0.0, 0.0, 0.0, 0.0]

```

Figure 9: Retrieving joints state

From fig.9, we can see that retrieved joint states correspond to the joint command earlier sent in fig.6, showing the robot is working as expected.

## 5 Discussion

The scara robot model presented in this report has been carefully designed based on the lab requirement. The following commands can be used to

- **Launch the model in rViz:** `roslaunch scara_robot robot_rviz.launch`
- **Launch the model in Gazebo:** `roslaunch scara_robot joints.launch`
- **Send commands to joint controllers:** `rostopic pub -1 /MJBot/joint_group_position_controller/command std_msgs/Float64MultiArray "data: [1.0, 0.0, 1.0, 2.0]"` (the data specified here is an example)
- **Retrieve joint states:** `rostopic echo -n 1 /MJBot/joint_states`
- **Get the transformation from base-link to gripper in rViz:** `roslaunch tf tf_echo base_link gripper`

### 5.1 Challenges Encountered

The major challenge encountered while building the model is that whenever the model is launched in gazebo, the error in the image below pops up at first but the model eventually works as expected. We couldn't find a way to resolve this because the p gain was actually specified in the yaml file.

```
[ INFO] [1675196259.244169887, 0.141000000]: Loading gazebo_ros_control plugin
[ INFO] [1675196259.244642576, 0.141000000]: Starting gazebo_ros_control plugin in namespace: MJBot
[ INFO] [1675196259.245834635, 0.141000000]: gazebo_ros_control plugin is waiting for model URDF in parameter [/robot_description] on the ROS param server.
[ERROR] [1675196259.382264308, 0.141000000]: No p gain specified for pid. Namespace: /MJBot/gazebo_ros_control/pid_gains/theta1_joint
[ERROR] [1675196259.383779874, 0.141000000]: No p gain specified for pid. Namespace: /MJBot/gazebo_ros_control/pid_gains/theta2_joint
[ERROR] [1675196259.385065520, 0.141000000]: No p gain specified for pid. Namespace: /MJBot/gazebo_ros_control/pid_gains/d3_joint
[ERROR] [1675196259.385878758, 0.141000000]: No p gain specified for pid. Namespace: /MJBot/gazebo_ros_control/pid_gains/theta4_joint
[ INFO] [1675196259.398528212, 0.141000000]: Loaded gazebo_ros_control.
[ INFO] [1675196259.559975, 0.203000]: /clock is published. Proceeding to load the controller(s).
[ INFO] [1675196259.561737, 0.204000]: Controller Spawner: Waiting for service controller_manager/load_controller
[ INFO] [1675196259.564347, 0.207000]: Controller Spawner: Waiting for service controller_manager/switch_controller
[ INFO] [1675196259.567334, 0.210000]: Controller Spawner: Waiting for service controller_manager/unload_controller
[ INFO] [1675196259.570134, 0.212000]: Loading controller: joint_group_position_controller
[ INFO] [1675196259.593667, 0.236000]: Loading controller: joint_state_controller
[ INFO] [1675196259.617213, 0.259000]: Controller Spawner: Loaded controllers: joint_group_position_controller, joint_state_controller
```

Figure 10: gazebo error

Additionally, the lab was very demanding due to the fact that we still struggle to understand some concept in ROS. In conclusion, this lab has successfully demonstrated the utilization of the Robot Operating System (ROS) framework to design, visualize, and simulate a four-degrees-of-freedom SCARA robot model.