# RBE 500 Group Assignment #1

**Joshua Gross, Arjan Gupta, Melissa Kelly**

# Problem 1

## Create SCARA Robot in Gazebo

The 3 DOF SCARA robot we have built is shown below.

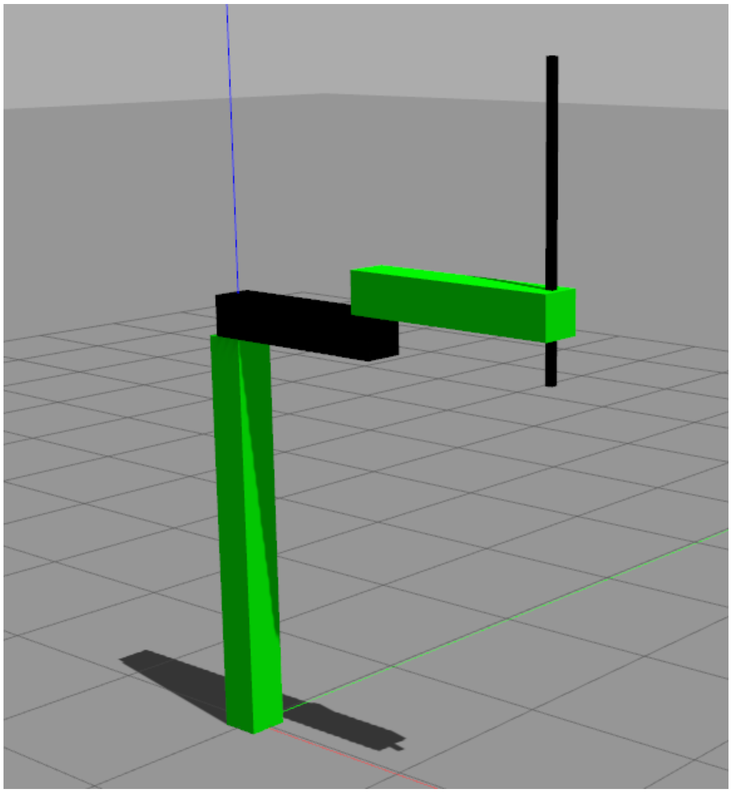


Figure 1: Final SCARA built by our team

We undertook the following steps to create our SCARA robot.

### 1 — Modify joint locations

In the downloaded package, the RRBot robot has its revolute joints on the 'sides' of its links, as shown in the following figure.

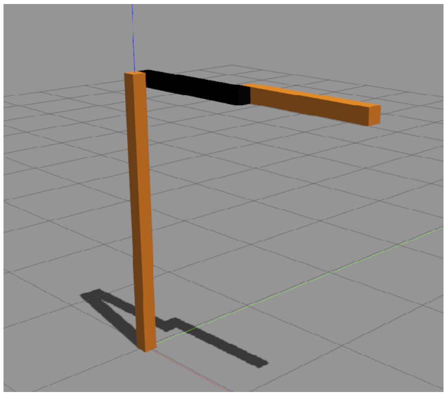


Figure 2: Initial RRBot in Gazebo

Problem 1 continued on next page. . . 

However, for a standard SCARA robot, we want the revolute joints to sweep angles in the XY plane of the world frame, not in the XZ plane.
Hence, we edited the <joint> element blocks in the URDF file rrbot_description.urdf.xacro. For the first joint, we made the following change.

```
    <joint name="${prefix}joint1" type="revolute">
      <parent link="${prefix}base_link"/>
      <child link="${prefix}link1"/>
      <!-- Set limits of revolute joint to -90deg to +90deg, 1000 N effort limit, velocity ...
          of 180 rad/s  -->
      <limit lower="-1.5708" upper="1.5708" effort="1000" velocity="3.14159"/>
      <origin xyz="0 0 ${height1 + axel_offset*2}" rpy="0 1.5708 0"/>
      <axis xyz="-1 0 0"/>
      <dynamics damping="0.7"/>
    </joint>
```

In the above code snippet, we changed the type attribute of the joint element from continuous to revolute. We also added the limit sub-element, and modified the origin and axis sub-elements. We made similar changes for the second joint, for which the code snippet is shown below.

```
      <joint name="${prefix}joint2" type="revolute">
        <parent link="${prefix}link1"/>
        <child link="${prefix}link2"/>
        <!-- Set limits of revolute joint to -90deg to +90deg, 1000 N effort limit, ...
            velocity of 180 rad/s  -->
        <limit lower="-1.5708" upper="1.5708" effort="1000" velocity="3.14159"/>
        <origin xyz="${width * -1} 0 ${height2 - axel_offset*2}" rpy="0 0 0"/>
        <axis xyz="-1 0 0"/>
        <dynamics damping="0.7"/>
      </joint>
```

As a result, our robot now looked like the following image.

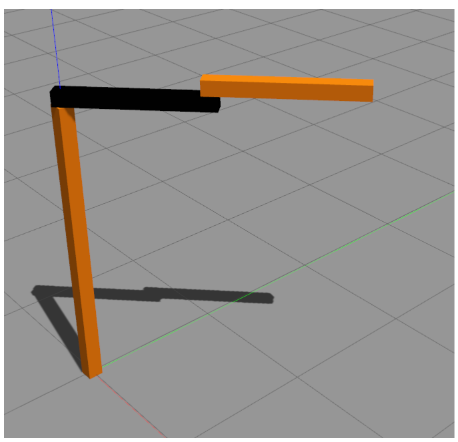


Figure 3: RRBot with top-joints

In order to test our changes, we moved our robot by publishing joint values in the format of the following ROS command in our terminal.

```
        ros2 topic pub --once /forward_position_controller/commands ...
            std_msgs/msg/Float64MultiArray "{data: [0.75 0.82]}"
```

### 2 — Add prismatic joint

Now our revolute joints resemble those of a SCARA robot, but we still need a prismatic joint. In order to do this, we first made the following changes to the rrbot_description.urdf.xacro file.

```
      <!-- Tool Joint -->
      <joint name="${prefix}tool_joint" type="prismatic">
        <parent link="${prefix}link2"/>
        <child link="${prefix}tool_link" />
        <!-- Set limits of prismatic joint -->
        <limit lower="${prismatic_offset * -1}" upper="1.1" effort="1000" velocity="1"/>
        <origin xyz="${prismatic_offset - height4} 0 ${height3 - axel_offset*2}" rpy="0 ...
            1.5708 0" />
        <axis xyz="0 0 1"/>
        <dynamics damping="0.7"/>
      </joint>

      <!-- Tool Link -->
      <link name="${prefix}tool_link">
        <collision>
          <origin xyz="0 0 ${height4/2 - axel_offset}" rpy="0 0 0"/>
          <geometry>
            <box size="${prismatic_width} ${prismatic_width} ${height4}"/>
          </geometry>
        </collision>

        <visual>
          <origin xyz="0 0 ${height4/2 - axel_offset}" rpy="0 0 0"/>
          <geometry>
            <box size="${prismatic_width} ${prismatic_width} ${height4}"/>
          </geometry>
        </visual>

        <inertial>
          <origin xyz="0 0 ${height4/2 - axel_offset}" rpy="0 0 0"/>
          <mass value="${mass}"/>
          <inertia
            ixx="${mass / 12.0 * (prismatic_width*prismatic_width + height4*height4)}" ...
                ixy="0.0" ixz="0.0"
            iyy="${mass / 12.0 * (height4*height4 + prismatic_width*prismatic_width)}" ...
                iyz="0.0"
            izz="${mass / 12.0 * (prismatic_width*prismatic_width + ...
                prismatic_width*prismatic_width)}"/>
          </inertial>
      </link>
```

In the above code snippet, we have changed the tool joint from fixed to prismatic, defined its translation axis, set its limits, and defined its origin. We defined a special `prismatic_width` so that we could make the tool link thinner than the regular links. We also defined a `prismatic_offset` so that the tool link is slightly 'lowered'. This makes it so that the zero position of the tool link is at the same height as our first link, which makes our calculations consistent with our forward-kinematics derivation.

Next, we defined the tool joint in the rrbot.ros2_control.xacro file, so that the command and state interfaces

for ROS2 could be made available for that joint. Finally, we also edited the gazebo_controllers.yaml file to define the tool joint as part of the three controllers — `forward_position_controller`, `forward_velocity_controller`, and `forward_effort_controller`. All these modified files have been included as part of our submission inside the rrbot_description.zip compressed directory.
At this point, our robot looked as follows.

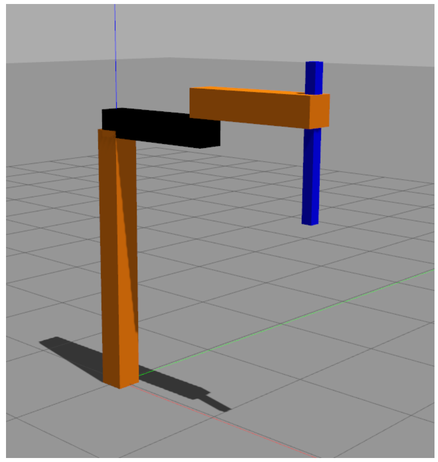


Figure 4: SCARA configuration without finishing touches

After this, we tweaked some offsets, adjusted some link lengths, changed the colors of the robot, and decreased the thickness of the tool link in order to arrive at our finished implementation of the SCARA robot. We also used the same ROS command as earlier (except with 3 data-points this time) to ensure that we were able to move all the joints of our robot.

# Question 2

## Forward Kinematics for SCARA

Before implementing forward kinematics in ROS, we worked out a derivation for the forward kinematics of the SCARA robot model.

### Derivation

The image below shows the frame assignments for the robot.

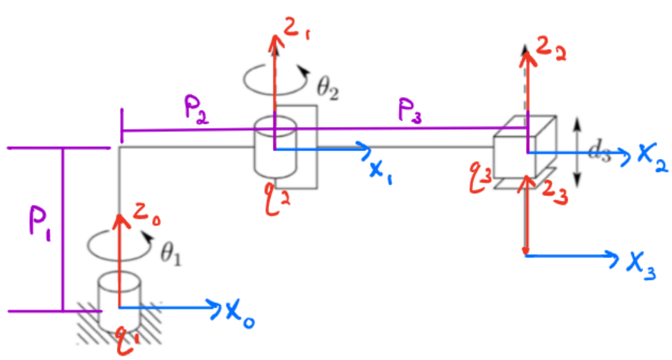



Figure 5: Frame assignments for forward kinematics

Based on this, we can form the DH table as shown below.

| Link | $\alpha_i$ | $a_i$ | $\theta_i$ | $d_i$ |
|---|---|---|---|---|
| 1 | 0 | $p_2$ | $q_1$ | $p_1$ |
| 2 | 0 | $p_3$ | $q_2$ | 0 |
| 3 | 0 | 0 | 0 | $q_3$ |

Next, we create a MATLAB script as shown below. We use the matrix obtained from equation 3.10 of the textbook to write $A_1, A_2, A_3$. By multiplying these matrices, we obtain the $T_3^0$.

```
%% Forward Kinematics
clc
clear

syms P1 P2 P3 q1 q2 q3
a = [P2 P3 0];
theta = [ q1 q2 0];
d = [P1 0 q3];
alpha = [ 0 0 0];

% NOTE: Please enter theta in degrees

i = 1;
A1 = [cosd(theta(i)) (-sind(theta(i))*cosd(alpha(i))) (sind(theta(i))*sind(alpha(i))) ...
     (a(i)*cosd(theta(i)));
     sind(theta(i)) (cosd(theta(i))*cosd(alpha(i))) (-cosd(theta(i))*sind(alpha(i))) ...
         (a(i)*sind(theta(i)));
     0 sind(alpha(i)) cosd(alpha(i)) d(i);
     0 0 0 1];

i = 2;
A2 = [cosd(theta(i)) (-sind(theta(i))*cosd(alpha(i))) (sind(theta(i))*sind(alpha(i))) ...
     (a(i)*cosd(theta(i)));
     sind(theta(i)) (cosd(theta(i))*cosd(alpha(i))) (-cosd(theta(i))*sind(alpha(i))) ...
         (a(i)*sind(theta(i)));
     0 sind(alpha(i)) cosd(alpha(i)) d(i);
     0 0 0 1];

i = 3;
A3 = [cosd(theta(i)) (-sind(theta(i))*cosd(alpha(i))) (sind(theta(i))*sind(alpha(i))) ...
     (a(i)*cosd(theta(i)));
     sind(theta(i)) (cosd(theta(i))*cosd(alpha(i))) (-cosd(theta(i))*sind(alpha(i))) ...
         (a(i)*sind(theta(i)));
     0 sind(alpha(i)) cosd(alpha(i)) d(i);
     0 0 0 1];

T30 = A1*A2*A3;

% Export to latex
latex(T30)
```

This MATLAB script gives us the following $T_3^0$ matrix, which is the homogeneous transformation for the end-effector with respect to the base frame.

Question 2 continued on next page... 

$$\begin{bmatrix} \cos q_1\ \cos q_2 - \sin q_1\ \sin q_2 & -\cos q_1\ \sin q_2 - \cos q_2\ \sin q_1 & 0 & P_2\ \cos q_1 + P_3\ \cos q_1\ \cos q_2 - P_3\ \sin q_1\ \sin q_2 \\ \cos q_1\ \sin q_2 + \cos q_2\ \sin q_1 & \cos q_1\ \cos q_2 - \sin q_1\ \sin q_2 & 0 & P_2\ \sin q_1 + P_3\ \cos q_1\ \sin q_2 + P_3\ \cos q_2\ \sin q_1 \\ 0 & 0 & 1 & P_1 + q_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

**ROS Implementation of Forward Kinematics**

For implementing the forward kinematics in ROS, a new package was created (this package has been submitted as a zip file along with this report). Inside this package, a file was created to house all of the code for the forward kinematics subscriber, calculations, and publisher. In the code, we first create the Subscriber class which houses both the `__init__` and `calculate_forward_kinematics` functions. The `__init__` function initializes certain attributes to be used later. Here, the link lengths, as provided by the Gazebo model, are defined. Next, the subscriber and publisher are each created and defined by their respective message types and topics. Inside of the `calculate_forward_kinematics` function, the message containing each joint value is accepted and split apart into each individual value as a variable. These joint values, along with the link lengths are then plugged into the overall transformation matrix as previously calculated by hand and solved with Matlab. From this 3x4 matrix, the first three rows and columns are extracted as the rotation matrix. Once this matrix is obtained, an open source scipy package is used to calculate the quaternion. Elements of the rotation matrix as well as the quaternion are then extracted and placed into the pose, which will be published. Once this pose message is published, it can be accessed through terminal to obtain the location of the end effector. Below are the commands used with this system to obtain the desired results:

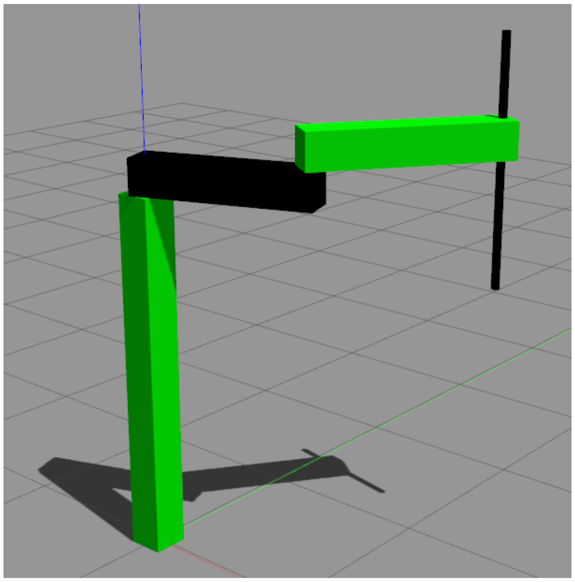
```
        # Moves the Gazebo robot to the desired position
        ros2 topic pub --once /forward_position_controller/commands ...
            std_msgs/msg/Float64MultiArray "{data: [0.349066 0.610865 0.5]}"
        # Kicks off the subscriber so that it can start calculating from the joint values ...
            that are repeatedly published
        ros2 run group_project calculate
```

**Gazebo Screenshots with ROS Output**

<u>Pose 1</u>
First we moved our SCARA robot by changing the joint values to $q_1 = 20°$, $q_2 = 35°$, $q_3 = 0.5$. We did this by executing the following command

```
        ros2 topic pub --once /forward_position_controller/commands ...
            std_msgs/msg/Float64MultiArray "{data: [0.349066 0.610865 0.5]}"
```

Next, we launced our forward kinematics calculation ROS node. The screenshots obtained are shown in Figure 6.



(a) Gazebo robot screenshot


(b) Terminal output screenshot

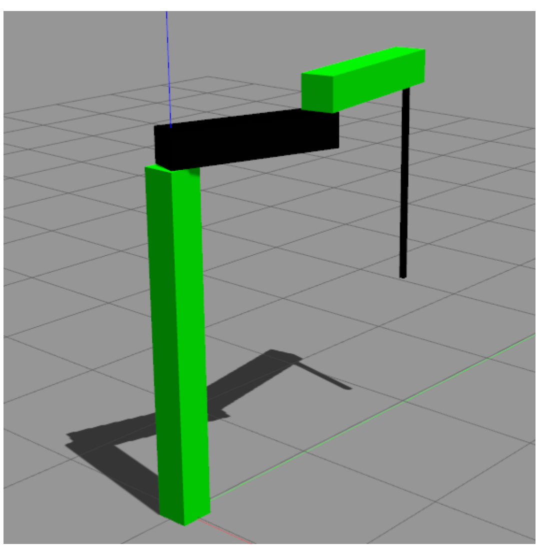
Figure 6: Pose 1 Screenshots

<u>Pose 2</u>

We moved our SCARA robot by setting the joint values to $q_1 = 72°$, $q_2 = 28°$, $q_3 = 1.05$. We did this by executing the following command

```
ros2 topic pub --once /forward_position_controller/commands ...
    std_msgs/msg/Float64MultiArray "{data: [1.25664 0.488692 1.05]}"
```

After this we launched the forward kinematics ROS node as well. The obtained screenshots are shown in Figure 7.

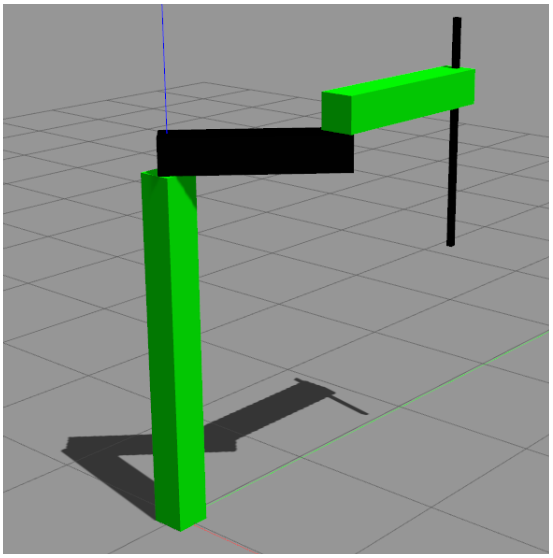


(a) Gazebo robot screenshot


(b) Terminal output screenshot

Figure 7: Pose 2 Screenshots

Pose 3



(a) Gazebo robot screenshot

```
geometry_msgs.msg.Pose(position=geometry_msgs.msg.Point(x=0.70
71123957996255, y=1.7071049903309277, z=2.6640000002453235), o
rientation=geometry_msgs.msg.Quaternion(x=0.0, y=0.0, z=0.7071
053195440795, w=0.7071082428259943))
geometry_msgs.msg.Pose(position=geometry_msgs.msg.Point(x=0.70
71123987703104, y=1.7071049893663792, z=2.664000000243426), or
ientation=geometry_msgs.msg.Quaternion(x=0.0, y=0.0, z=0.70710
53187709831, w=0.7071082435990873))
geometry_msgs.msg.Pose(position=geometry_msgs.msg.Point(x=0.70
71124017376698, y=1.7071049884029097, z=2.6640000002415443), o
rientation=geometry_msgs.msg.Quaternion(x=0.0, y=0.0, z=0.7071
053179987525, w=0.7071082443713148))
geometry_msgs.msg.Pose(position=geometry_msgs.msg.Point(x=0.70
71124047017151, y=1.7071049874405166, z=2.6640000002396746), o
rientation=geometry_msgs.msg.Quaternion(x=0.0, y=0.0, z=0.7071
053172273842, w=0.70710824514268))
geometry_msgs.msg.Pose(position=geometry_msgs.msg.Point(x=0.70
71124076624449, y=1.7071049864791998, z=2.6640000002378184), o
rientation=geometry_msgs.msg.Quaternion(x=0.0, y=0.0, z=0.7071
053164568788, w=0.7071082459131823))
geometry_msgs.msg.Pose(position=geometry_msgs.msg.Point(x=0.70
71124106198661, y=1.7071049855189568, z=2.6640000002359763), o
rientation=geometry_msgs.msg.Quaternion(x=0.0, y=0.0, z=0.7071
053156872342, w=0.7071082466828235))
geometry_msgs.msg.Pose(position=geometry_msgs.msg.Point(x=0.70
71124135739838, y=1.7071049845597863, z=2.6640000002341453), o
rientation=geometry_msgs.msg.Quaternion(x=0.0, y=0.0, z=0.7071
053149184495, w=0.7071082474516049))
geometry_msgs.msg.Pose(position=geometry_msgs.msg.Point(x=0.70
71124165247948, y=1.7071049836016905, z=2.6640000002323276), o
```

(b) Terminal output screenshot

Figure 8: Pose 3 Screenshots

We moved our SCARA robot by setting the joint values to $q_1 = 45°$, $q_2 = 45°$, $q_3 = 0.65$. We did this by executing the following command

```
ros2 topic pub --once /forward_position_controller/commands ...
    std_msgs/msg/Float64MultiArray "{data: [0.785398 0.785398 0.65]}"
```

After this we launched the forward kinematics ROS node as well. The obtained screenshots are shown in Figure 8

# Question 3

## Inverse Kinematics for SCARA

Before implementing forward kinematics in ROS, we worked out a derivation for the inverse kinematics of the SCARA robot model.

### Derviation

For the inverse kinematics, a side view and a top view were used to geometrically solve for the unknown joint values.

Top view

From the figure above, $q_1$ can be expressed as

$$\boxed{q_1 = \arctan\left(\frac{y_3^0}{x_3^0}\right) - \gamma}$$

Similarly, $q_2$ can be expressed as,

$$\boxed{q_2 = 180° - \beta}$$

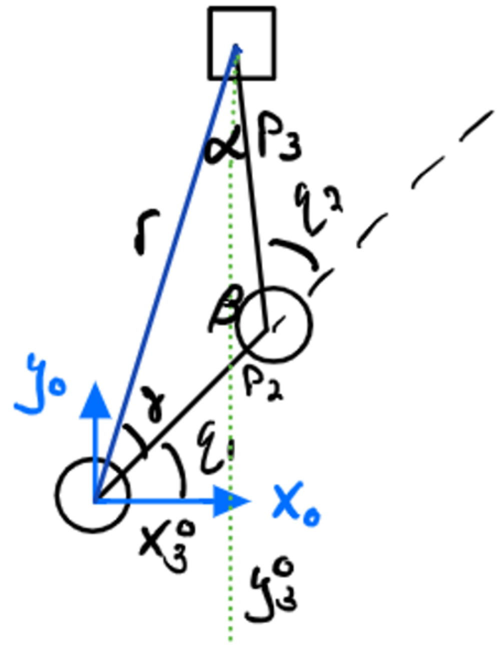



Figure 9: Top view for inverse kinematics

In order to find $\alpha$ and $\gamma$, we can use the law of cosines. For instance, $\alpha$ can be found as follows,

$$
\begin{aligned}
{P_2}^2 &= r^2 + {P_3}^2 - 2rP_3 \cos\alpha \\
2rP_3 \cos\alpha &= r^2 + {P_3}^2 - {P_2}^2 \\
\alpha &= \arccos\left(\frac{r^2 + {P_3}^2 - {P_2}^2}{2rP_3}\right)
\end{aligned}
$$

Similarly, $\gamma$ can be expressed as,

$$\gamma = \arccos\left(\frac{r^2 + {P_2}^2 - {P_3}^2}{2rP_3}\right)$$

Hence, $\beta$ can be written as,

$$\beta = 180^\circ - \alpha - \gamma$$

Also, we can express r in its norm,

$$r = \sqrt{(y_3^0)^2 + (x_3^0)^2}$$

<u>Side view</u>
From the figure, we can find $q_3$ as follows

$$z_3^0 = P_1 - q_3$$

$$\boxed{q_3 = -(P_1 - z_3^0)}$$

Now we have expressed all the joint variables in the form of constants of the system and end-effector position coordinates. These equations for each joint value can now be solved once the end effector location is known. In the following MATLAB script, these equations are written out, and when plugging in the values found in the forward kinematics, the original joint values are found, therefore proving their accuracy.

Question 3 continued on next page. . . 

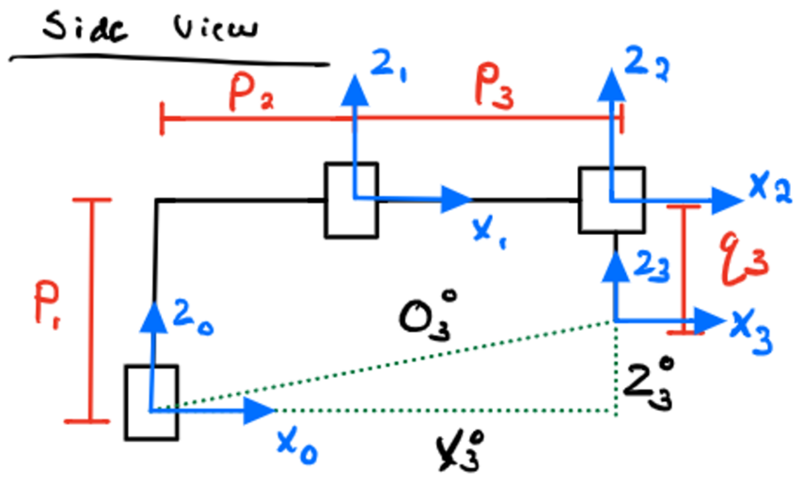



Figure 10: Side view for inverse kinematics

```
% Group Project Part 1 Forward and Inverse Kinematics

clc; clear

% Forward Kinematics

% Link lengths from Gazebo
P1 = 2;
P2 = 1;
P3 = 1;

% This will come from ROS, current values are arbitrary for testing
q1 = 45;
q2 = 45;
q3 = 0.65;

% DH parameters from hand-written work
a = [P2 P3 0];
theta = [ q1 q2 0];
d = [P1 0 q3];
alpha = [ 0 0 0];

% A10, A21, and A32 transformation matrices
i = 1;
A1 = [cosd(theta(i)) (-sind(theta(i))*cosd(alpha(i))) (sind(theta(i))*sind(alpha(i))) ...
     (a(i)*cosd(theta(i)));
    sind(theta(i)) (cosd(theta(i))*cosd(alpha(i))) (-cosd(theta(i))*sind(alpha(i))) ...
        (a(i)*sind(theta(i)));
    0 sind(alpha(i)) cosd(alpha(i)) d(i);
    0 0 0 1];

i = 2;
A2 = [cosd(theta(i)) (-sind(theta(i))*cosd(alpha(i))) (sind(theta(i))*sind(alpha(i))) ...
     (a(i)*cosd(theta(i)));
    sind(theta(i)) (cosd(theta(i))*cosd(alpha(i))) (-cosd(theta(i))*sind(alpha(i))) ...
        (a(i)*sind(theta(i)));
    0 sind(alpha(i)) cosd(alpha(i)) d(i);
    0 0 0 1];

i = 3;
```

```
A3 = [cosd(theta(i)) (-sind(theta(i))*cosd(alpha(i))) (sind(theta(i))*sind(alpha(i))) ...
    (a(i)*cosd(theta(i)));
    sind(theta(i)) (cosd(theta(i))*cosd(alpha(i))) (-cosd(theta(i))*sind(alpha(i))) ...
        (a(i)*sind(theta(i)));
    0 sind(alpha(i)) cosd(alpha(i)) d(i);
    0 0 0 1];

% End effector pose
T30 = A1*A2*A3;
end_effector_pose = T30 % display whole end-effector T matrix
end_effector = T30(1:3,4); % use only position part of T

% Inverse Kinematics

%These will come from the forward kinematics
x30 = end_effector(1);
y30 = end_effector(2);
z30 = end_effector(3);

q3 = z30 - P1;

r = sqrt((y30^2) + (x30^2));

% Needed angles from law of cosines
alpha = acos(((r^2) + (P3^2) - (P2^2))/(2*r*P3));
gamma = acos(((P2^2)+(r^2)-(P3^2))/(2*P2*r));
beta = 180 - alpha - gamma;

q2 = 180 - beta;
q1 = atan(y30/x30)-gamma;

% Calculated joint values
joint_angles = [(q1/(pi/180)); (q2/(pi/180)); q3]
```

Both the forward and inverse kinematics are now solved and ready to be plugged into ROS.

### ROS Implementation of Inverse Kinematics

For implementing the ROS portion of forward kinematics, we lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Purus semper eget duis at tellus at. Ac turpis egestas integer eget aliquet nibh. Aliquam etiam erat velit scelerisque in. Ac turpis egestas sed tempus urna et pharetra pharetra. Faucibus in ornare quam viverra. Libero id faucibus nisl tincidunt eget.

Aenean pharetra magna ac placerat vestibulum lectus mauris. Consequat nisl vel pretium lectus quam id leo. Lorem sed risus ultricies tristique nulla aliquet enim tortor at. Ligula ullamcorper malesuada proin libero nunc.

### Terminal Screenshots of ROS Output

<u>Screenshot 1</u>

Question 3 continued on next page. . .

<u>Screenshot 2</u>

<u>Screenshot 3</u>