ALPHABET WRITING MANIPULATOR

Submitted By

TEAM -11

M KARTHIKEYAN (CB.EN.U4AIE20029)

RISHEKESAN SV (CB.EN.U4AIE20058)

……..

For the Completion of

21AIE213 – ROBOTIC OPERATING SYSTEMS AND ROBOT SIMULATION

CSE - AI

12th July 2022

INTRODUCTION

One of the primary functions of ROS is to facilitate communication between the user, the computer's operating system, and equipment that is not connected to the computer. These tools may include robots, cameras, and sensors. The advantage of ROS, as with any operating system, is the hardware abstraction and its capacity to control a robot without requiring the user to be familiar with all of the robot's specifics. This provides an additional benefit when designing our robots.

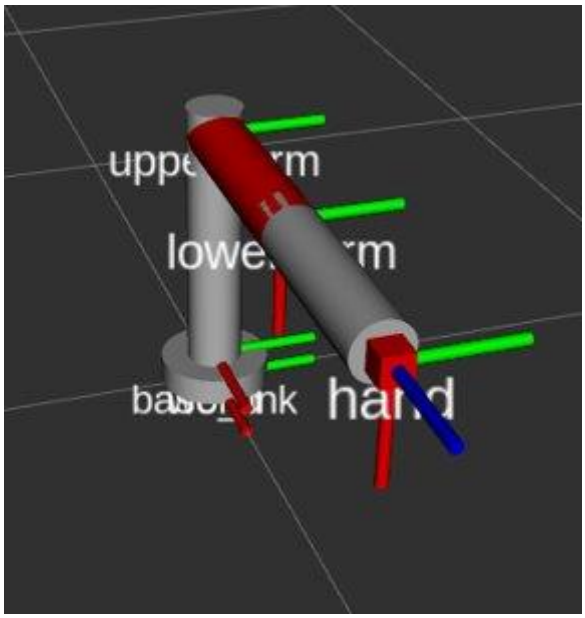
In the GAZEBO environment, our goal is to simulate a robot that draws particular alphabets with its corresponding coordinates. The letters A, T, H, E, L, and F are written using a planar robotic manipulator that we develop. The system receives an alphabet to write and outputs the alphabet that our robot drew. We simulate various alphabets in our project based on user input.

AIM

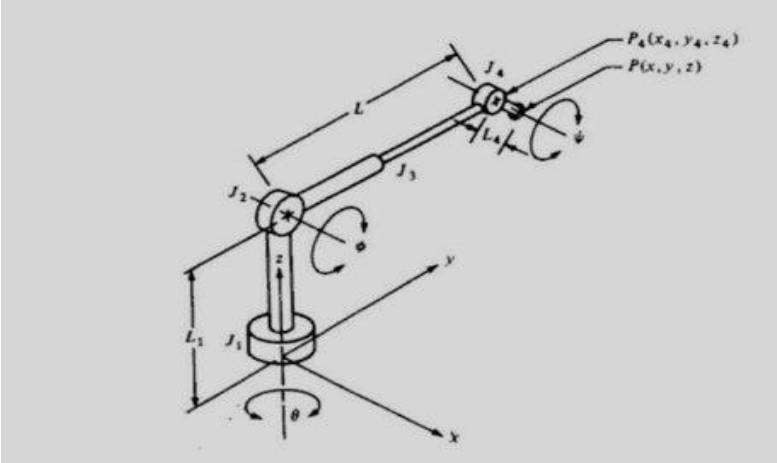
Simulating a robot that draws specific alphabets with having its respective co-ordinates and simulate it in the GAZEBO environment. We mainly create a planar robotic manipulator to draw the letters A, T, H, E, L and F based on user input.

THEORY

The manipulator that we use is 4 DOF 3R planar manipulator having 4 joints namely wrist , hip, shoulder, elbow joints namely.



The degree of freedom is 4 because three for the different links connected, and one joint for the end effector. Based on the transformations we make we can draw the alphabet based on user input.



The above picture represents the sample model of 4 DOF 3R manipulator design.

- Joint 1 (type T joint) allows rotation about the axis;
- Joint 2 (type R) allows rotation about an axis that is perpendicular to the z axis,
- Joint 3 is a linear joint which is capable of sliding over a certain range
- Joint 4 is a type R joint which allows rotation about an axis that is parallel to the joint 2 axis.

Let us define the angle of rotation of joint 1 to be the base rotation θ; the angle of rotation of joint 2 will be called the elevation angle φ; the length of linear joint 3 will be called the extension L (L represents a combination of links 2 and 3); and the angle that joint 4 makes ψ with the x - y plane will be called the pitch angle.

The position of the end of the wrist, P, defined in the world coordinate system for the robot, is given by

$x = \cos\theta\ (L\cos\varphi + L_4\cos\psi)$

$y = \sin\theta\ (L\cos\varphi + L_4\cos\psi)$

$z = L_1 + L\sin\varphi + L_4\sin\psi$

Given the specification of point P (x, y, z) and pitch angle ψ , we can find any of the joint positions relative to the world coordinate system. Using PA (x_4, y_4, z_4), which is the position of joint 4, as an example,

$x_4 = x - \cos\theta\ (L_4\cos\psi)$

$y_4 = y\text{-}\sin\theta\ (L_4 \cos\psi)$

$z_4 = z\text{-}L_4 \sin\psi$

The values of L, φ, and θ can next be computed:

$L= [x_4\text{^}2+y_4\text{^}2+(z_4 - L_1)^2\ ]^{-1}$

$\text{Sin}\ \varphi = z_4 - L_1\ / L$

$\text{Cos}\ \theta = y_4 / L$

- Each robot link is given a coordinate frame that extends from the base to the end-effector. Other nearby objects (like a camera mounted on a wall, a ball it must pick up, the end-goal-pose, effector's, etc.) are given coordinate frames in addition to the robot's links.
- These frames will give us information about the orientation and placement of the frame in relation to other frames. The coordinates of these frames are found using the Transformation matrices.
- These frame coordinates must be updated whenever the robot or other objects move. The precise control of joints is necessary for accurate manipulator control. The forces and inertias acting on joints determine how they move.
- Robot links and joints can be modelled very easily using DH parameters, which can be applied to any robot configuration. This method is now widely used to represent robots and simulate their motions. This is crucial to our analysis of model control.

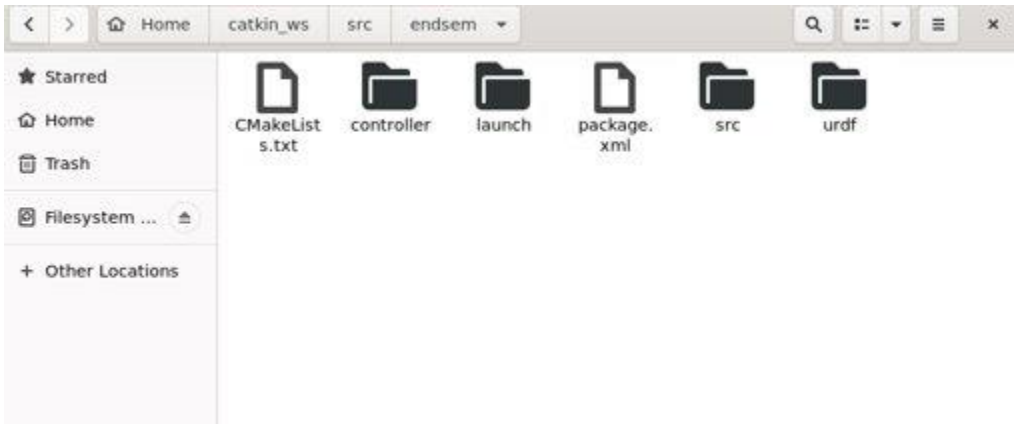
IMPLEMENTATION

PACKAGE CREATION:

Create a package with a list of dependencies which creates different files which include package.xml, CmakeLists.txt and src folder.

```
rishe@DESKTOP-SFQD8MJ:~/catkin_ws/src$ catkin_create_pkg TEAM_11_endsem std_msgs rospy roscpp
WARNING: Package name "TEAM_11_endsem" does not follow the naming conventions. It should start with a lower case letter
and only contain lower case letters, digits, underscores, and dashes.
Created file TEAM_11_endsem/package.xml
Created file TEAM_11_endsem/CMakeLists.txt
Created folder TEAM_11_endsem/include/TEAM_11_endsem
Created folder TEAM_11_endsem/src
Successfully created files in /home/rishe/catkin_ws/src/TEAM_11_endsem. Please adjust the values in package.xml.
```

Next, we proceed with the creation of different folders for the urdf files, launch files as well as the scripts files. In this case we first create the respective folders using the mkdir folder name command.



UPDATING CMAKELIST AND PACKAGE.XML

For building actions, include actionlib_msgs among the dependencies:

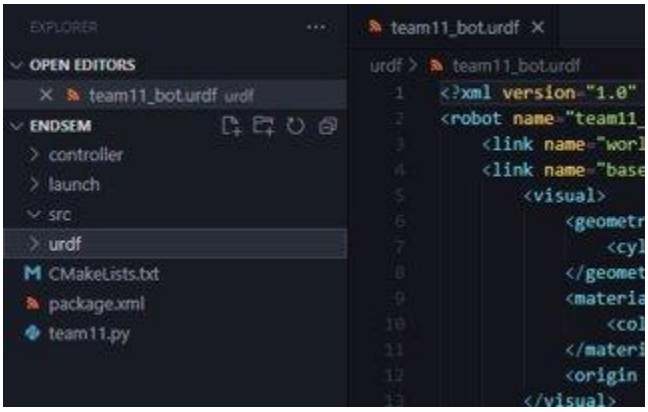
```
find_package(catkin REQUIRED COMPONENTS
  actionlib_msgs
  message_generation
  message_runtime
  rospy
  std_msgs
)
```

- actionlib_msgs defines the common messages to interact with an action server and an action client.
- Rospy is a ROS client library written entirely in Python. Python programmers can use the rospy client API to quickly interact with ROS Topics, Services, and Parameters.

We used visual –code IDE for creating the urdf and respective py scripts

```
rishe@DESKTOP-SFQD8MJ:~/catkin_ws/src/endsem$ code .
```

URDF FILE CREATION:



URDF is an XML format specifically defined to represent robot models down to their component level. These URDF files can become long and cumbersome on complex robot systems.

A C++ parser for the Unified Robot Description Format (URDF), an XML format for representing a robot model, is included in this package. The parser's code API has been reviewed and will remain backwards compatible in future releases.

urdf > team11_bot.urdf

```xml
<?xml version="1.0" ?>
<robot name="team11_bot">
    <link name="world" />
    <link name="base_link">
        <visual>
            <geometry>
                <cylinder length="0.05" radius="0.1" />
            </geometry>
            <material name="silver">
                <color rgba="0.75 0.75 0.75 1" />
            </material>
            <origin rpy="0 0 0" xyz="0 0 0.025" />
        </visual>
        <collision>
            <geometry>
                <cylinder length="0.05" radius="0.1" />
            </geometry>
            <origin rpy="0 0 0" xyz="0 0 0.025" />
        </collision>
```

- Defining the first link of radius 0.1 and length 0.05
- All we need to do to add dimensions to our connections is specify the offset from a link to the joint of its children. To do so, we will add the origin field to each of the joints.
- To simulate the robot in a program like Gazebo, we must define a collision tag. Next, we will add the <collision> properties to each of our <link> elements. Even though we have defined the visual properties of the elements, Gazebo's collision detection engine uses the collision property to identify the boundaries of the object. If an object has complex visual properties (such as a mesh), a simplified collision property should be defined in order to improve the collision detection performance.

```xml
<inertial>
            <mass value="0.1" />
            <inertia ixx="0.03" iyy="0.03" izz="0.03" ixy="0.0" ixz="0.0" iyz="0.0" />
        </inertial>

    </link>
    <joint name="fixed" type="fixed">
        <parent link="world" />
        <child link="base_link" />
    </joint>
    <link name="torso">
        <visual>
            <geometry>
                <cylinder length="0.5" radius="0.05" />
            </geometry>
            <material name="silver">
                <color rgba="0.75 0.75 0.75 1" />
            </material>
            <origin rpy="0 0 0" xyz="0 0 0.25" />
        </visual>
        <collision>
            <geometry>
                <cylinder length="0.5" radius="0.05" />
            </geometry>
            <origin rpy="0 0 0" xyz="0 0 0.25" />
        </collision>

<inertial>
```

Inside the inertial tag the 3x3 rotational inertia matrix is specified with the inertia element. Since this is symmetrical, it can be represented by only 6 elements, as such.With the additional physical properties of mass and inertia, our robot will be ready to be launched in the Gazebo simulator. These properties are needed by Gazebo's physics engine. We should not use inertia elements of zero (or almost zero) because real-time controllers can cause the robot model to collapse without warning, and all links will appear with their origins coinciding with the world origin.

```
<transmission name="trans_hip">
    <type>transmission_interface/SimpleTransmission</type>
    <joint name="hip">
        <hardwareInterface>hardware_interface/PositionJointInterface</hardwareInterface>
    </joint>
    <actuator name="hip_motor">
        <hardwareInterface>hardware_interface/PositionJointInterface</hardwareInterface>
        <mechanicalReduction>1</mechanicalReduction>
    </actuator>
</transmission>
<transmission name="trans_shoulder">
    <type>transmission_interface/SimpleTransmission</type>
    <joint name="shoulder">
        <hardwareInterface>hardware_interface/PositionJointInterface</hardwareInterface>
    </joint>
    <actuator name="shoulder_motor">
        <hardwareInterface>hardware_interface/PositionJointInterface</hardwareInterface>
        <mechanicalReduction>1</mechanicalReduction>
    </actuator>
</transmission>
<transmission name="trans_elbow">
    <type>transmission_interface/SimpleTransmission</type>
    <joint name="elbow">
        <hardwareInterface>hardware_interface/PositionJointInterface</hardwareInterface>
    </joint>
    <actuator name="elbow_motor">
        <hardwareInterface>hardware_interface/PositionJointInterface</hardwareInterface>
        <mechanicalReduction>1</mechanicalReduction>
    </actuator>
</transmission>
<transmission name="trans_wrist">
    <type>transmission_interface/SimpleTransmission</type>
    <joint name="wrist">
        <hardwareInterface>hardware_interface/PositionJointInterface</hardwareInterface>
    </joint>
    <actuator name="wrist_motor">
        <hardwareInterface>hardware_interface/PositionJointInterface</hardwareInterface>
        <mechanicalReduction>1</mechanicalReduction>
    </actuator>
</transmission>
```

The <transmission> element is used to define the relationship between the robot joint and the actuator. These elements are supposed to encapsulate the details of the mechanical coupling, with specific gear ratios and parallel linkages defined. Hardware_interface package, this package wraps existing raw data. Connecting and linking all the four links (Wrist,Hip,Shoulder,Elbow).Specific elements must be added to the URDF/SDF in order for a model to be controlled in the Gazebo simulation environment.

FOLLOWED BY THE URDF FILE CREATION NEXT, WE WILL BE CREATING THE PYTHON SCRIPT

Our code is structured in publisher, subscriber model phase where we will be we be using trajectory as tool in order to perform transformations which would be helpful in tracing the required alphabet based on a user input-based approach. Inserting the respective letter and its co-ordinates is the key phase in our code and also the order of execution.

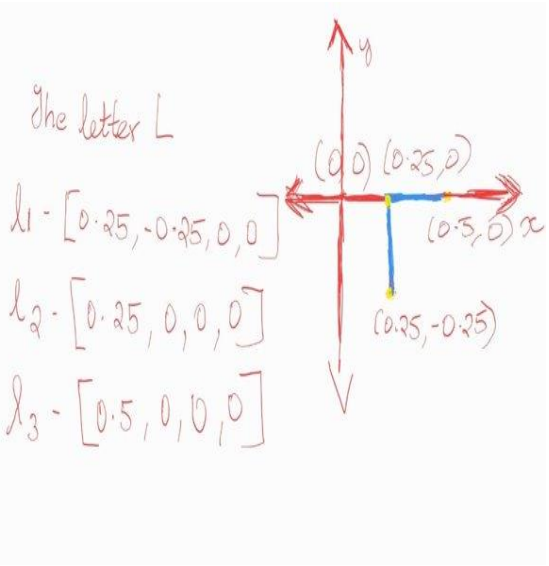
Messages for defining robot trajectories are defined in this package. These messages are also the foundation for the majority of the control msgs actions.

For Example: Letter(A) the co-ordinates are a1, a2, a3, a4, a5 are the transformation steps required for drawing alphabet - A

```python
#!/usr/bin/env python3

import rospy
from trajectory_msgs.msg import JointTrajectory
from trajectory_msgs.msg._JointTrajectoryPoint import JointTrajectoryPoint
import time

def move():
    arm_pub = rospy.Publisher('/arm_controller/command', JointTrajectory, queue_size=1, latch=True)

    rospy.init_node('arm_move', anonymous=True)
    rate = rospy.Rate(10)


    arm_cmd = JointTrajectory()

    arm_point = JointTrajectoryPoint()

    arm_cmd.joint_names = ["hip", "shoulder", "elbow", "wrist"]

    initial = [0.0, 0.0, 0.0, 0.0]

    a1 = [1.0, 0.0, 0.0, 0.0]
    a2 = [1.5, -1.0, 0.0, 0.0]
    a3 = [2.0, 0.0, 0.0, 0.0]
    a4 = [1.25, -0.5, 0.0, 0.0]
    a5 = [1.75, -0.5, 0.0, 0.0]

    t1 = [0.5, 0.0, 0.0, 0.0]
    t2 = [0.75, -0.5, 0.0, 0.0]
    t3 = [0.25, -0.5, 0.0, 0.0]

    h1 = [0.25, 0.0, 0.0, 0.0]
    h2 = [0.25, -0.5, 0.0, 0.0]
    h3 = [0.75, 0.0, 0.0, 0.0]
    h4 = [0.75, -0.5, 0.0, 0.0]
    h5 = [0.75, -0.25, 0.0, 0.0]
```

This below image represents the transformations required for tracing an alphabet. That is the initially we are at [0,0,0,0]. Based on the co-ordinates for each stage we first plot the points and then trace their part. This is very helpful in drawing alphabets that involve only vertical lines excluding curved letters.

```python
if (alphabet == 'L'):
    arm_point.positions = l1
    arm_cmd.points = [arm_point]
    arm_pub.publish(arm_cmd)
    time.sleep(3)

    arm_point.positions = l2
    arm_cmd.points = [arm_point]
    arm_pub.publish(arm_cmd)
    time.sleep(3)

    arm_point.positions = l3
    arm_cmd.points = [arm_point]
    arm_pub.publish(arm_cmd)
    time.sleep(3)

    print("L is published")

    arm_point.positions = initial
    arm_cmd.points = [arm_point]
    arm_pub.publish(arm_cmd)
    time.sleep(3)
```



In a similar fashion for different alphabets, we create transformations at each stage, and based on our letter we plot the points and trace the corresponding path.

```
        arm_cmd.points = [arm_point]
        arm_pub.publish(arm_cmd)
        time.sleep(3)

        arm_point.positions = t3
        arm_cmd.points = [arm_point]
        arm_pub.publish(arm_cmd)
        time.sleep(3)

        print("T is published")

        arm_point.positions = initial
        arm_cmd.points = [arm_point]
        arm_pub.publish(arm_cmd)
        time.sleep(3)


    if (alphabet == 'L'):
        arm_point.positions = l1
        arm_cmd.points = [arm_point]
        arm_pub.publish(arm_cmd)
        time.sleep(3)

        arm_point.positions = l2
        arm_cmd.points = [arm_point]
```

```
        time.sleep(3)

        arm_point.positions = f5
        arm_cmd.points = [arm_point]
        arm_pub.publish(arm_cmd)
        time.sleep(3)


        print("F is published")

        arm_point.positions = initial
        arm_cmd.points = [arm_point]
        arm_pub.publish(arm_cmd)
        time.sleep(3)




if __name__ == '__main__':
    try:
        move()
    except rospy.ROSInterruptException:
        pass
```

The python code can be downloaded at : [team11.py]

CREATING LAUNCH FILE:

Roslaunch is a ROS tool that makes it easy to launch multiple ROS nodes as well as set parameters on the ROS Parameter Server. Roslaunch configuration files are written in XML and typically end in a .launch extension. In a distributed environment, the. launch files also indicate the processor the nodes should run on.

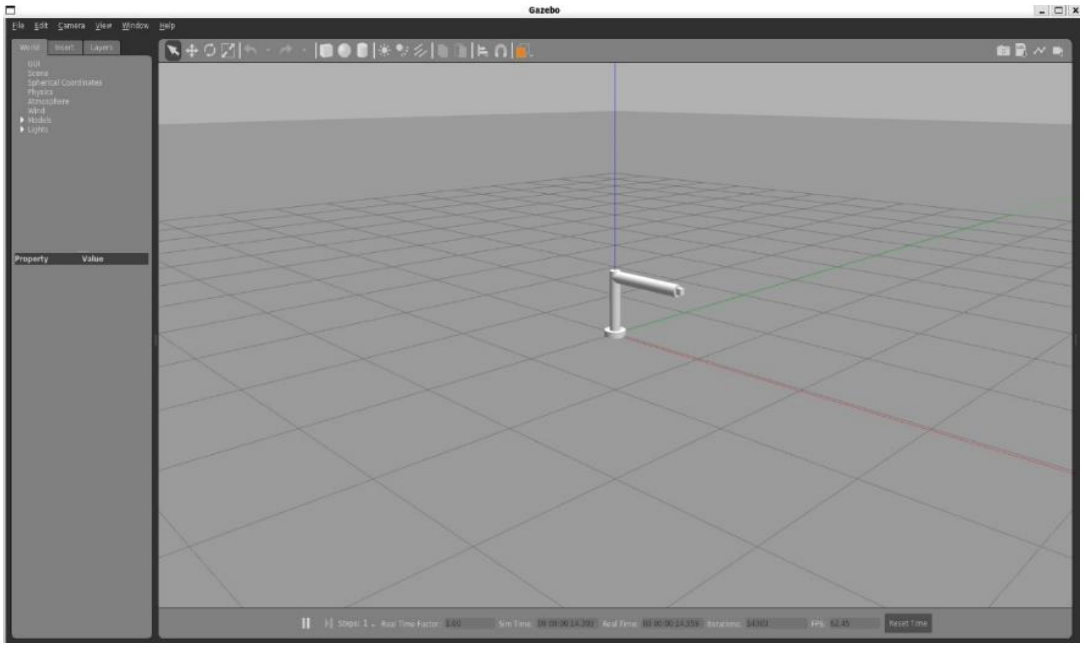
launch > endsem.launch

```
<launch>
<!-- Load the TortoiseBot URDF model into the parameter server -->
<param name="robot_description" textfile="$(find endsem)/urdf/team11_bot.urdf" />
<!-- Start Gazebo with an empty world -->
<include file="$(find gazebo_ros)/launch/empty_world.launch"/>
<!-- Spawn a TortoiseBot in Gazebo, taking the description from the
parameter server -->
<node name="spawn_urdf" pkg="gazebo_ros" type="spawn_model"
args="-param robot_description -urdf -model endsem" />

<rosparam file="$(find endsem)/controller/controllers.yaml" command="load"/>

<node name="controller_spawner" pkg="controller_manager" type="spawner" args="arm_controller"/>

<node name="robot_state_publisher" pkg="robot_state_publisher" type="robot_state_publisher"/>

</launch>
```

To use roslaunch for our URDF file, you will need to create a launch directory under our package and create the launch file using XML code. A pair of launch tags must separate the contents of a launch file. They offer an easy approach to initialize numerous nodes, a master, and other startup needs like specifying parameters.

OUTPUT:

First, we will be launching our launch file in terminal 1 using the roslaunch endsem endsem. launch command. This loads our manipulator in the gazebo environment. In order to perform some action in the manipulator we have to use a python script in another terminal which acts as a control operation.



Terminal 1

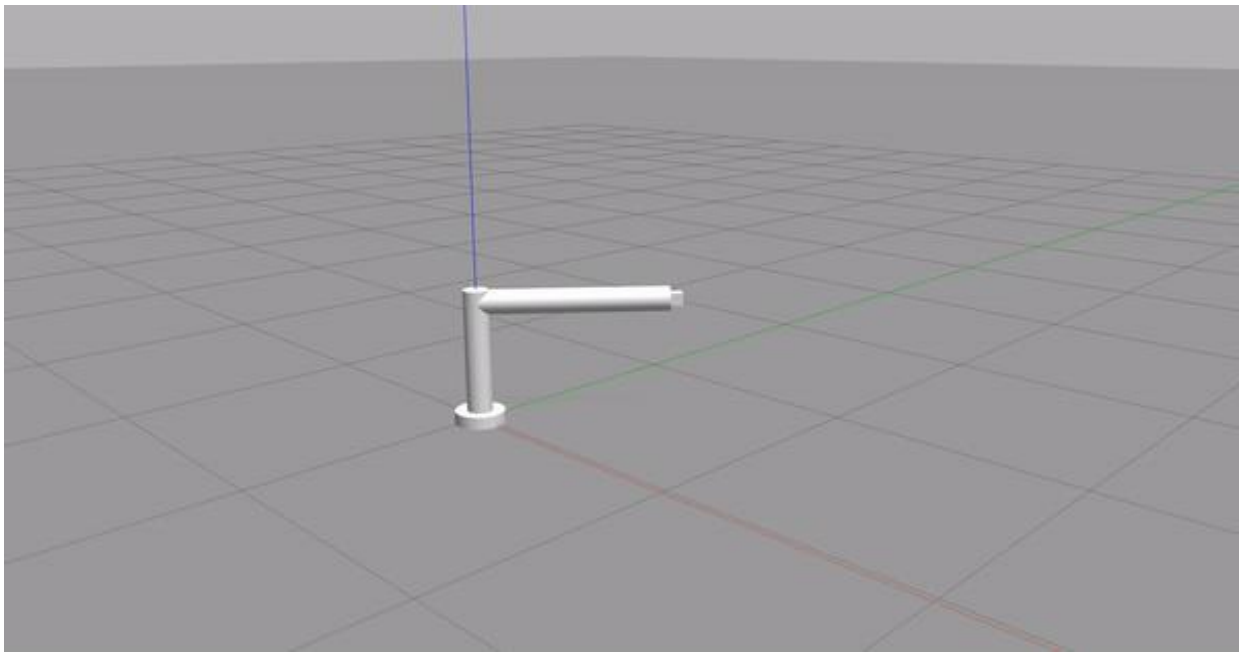
```
rrishe@DESKTOP-SFQD8MJ:~$ roslaunch endsem endsem.launch
WARNING: Package name "Assembly_Toby" does not follow the n
nd only contain lower case letters, digits, underscores, an
... logging to /home/rishe/.ros/log/4658e41e-0042-11ed-9991
Checking log directory for disk usage. This may take a whil
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

WARNING: Package name "Assembly_Toby" does not follow the n
nd only contain lower case letters, digits, underscores, an
started roslaunch server http://DESKTOP-SFQD8MJ:40877/

SUMMARY
========

PARAMETERS
 * /arm_controller/joints: ['hip', 'shoulder...
 * /arm_controller/type: position_controll...
 * /gazebo/enable_ros_network: True
 * /robot_description: <?xml version="1....
 * /rosdistro: noetic
 * /rosversion: 1.15.14
 * /use_sim_time: True
```

Terminal 2

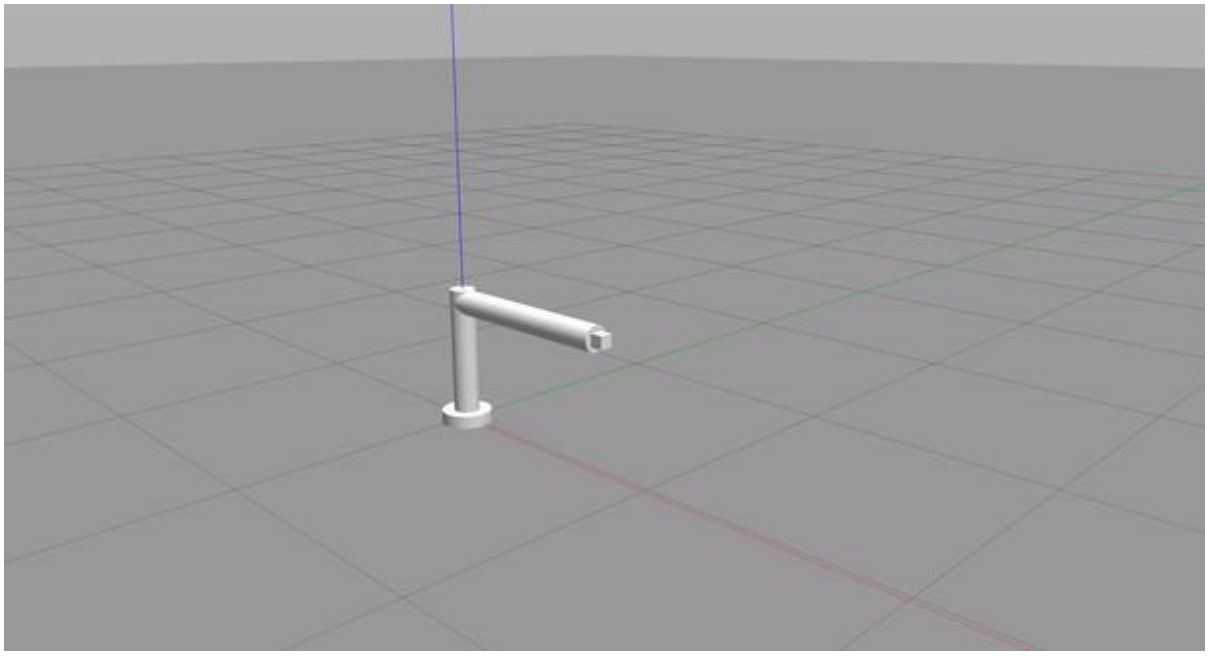
```
rishe@DESKTOP-SFQD8MJ:~$ rosrun endsem team11.py
Enter an alphabet in A T H L E F :
T
T is published
rishe@DESKTOP-SFQD8MJ:~$ rosrun endsem team11.py
Enter an alphabet in A T H L E F :
E
E is published
rishe@DESKTOP-SFQD8MJ:~$ rosrun endsem team11.py
Enter an alphabet in A T H L E F :
E
E is published
rishe@DESKTOP-SFQD8MJ:~$ rosrun endsem team11.py
Enter an alphabet in A T H L E F :
A
A is published
rishe@DESKTOP-SFQD8MJ:~$ rosrun endsem team11.py
Enter an alphabet in A T H L E F :
A
A is published
rishe@DESKTOP-SFQD8MJ:~$ rosrun endsem team11.py
Enter an alphabet in A T H L E F :
L
L is published
rishe@DESKTOP-SFQD8MJ:~$ rosrun endsem team11.py
Enter an alphabet in A T H L E F :
H
H is published
```

RESULTS:

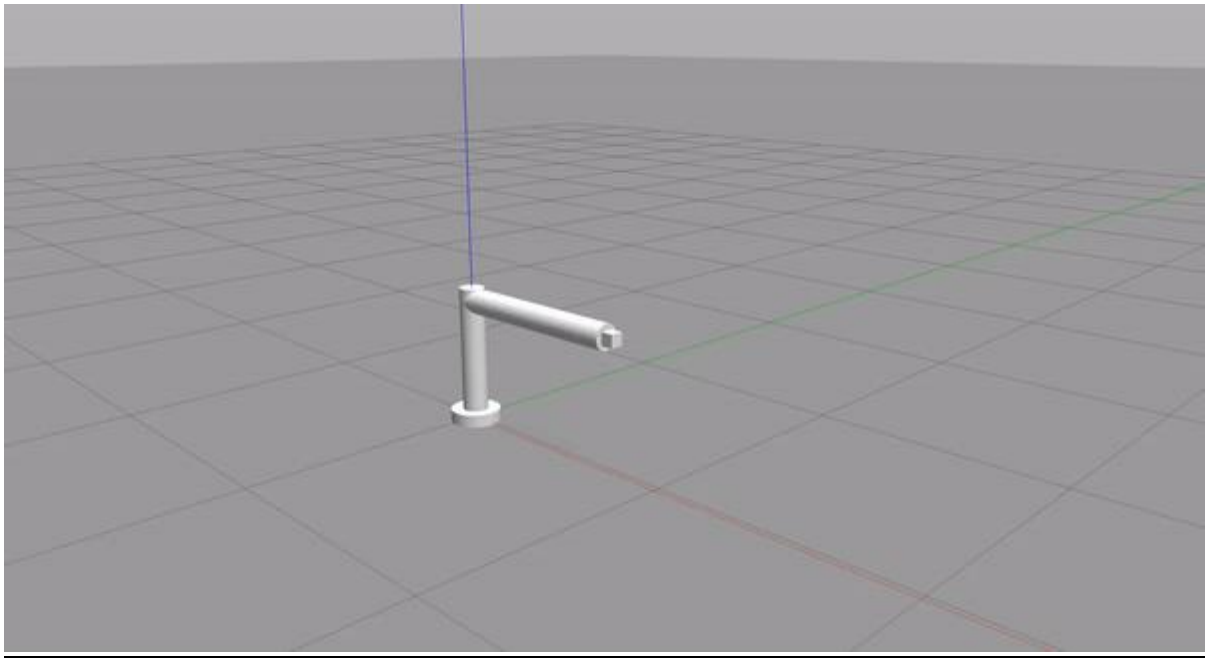
LETTER A:



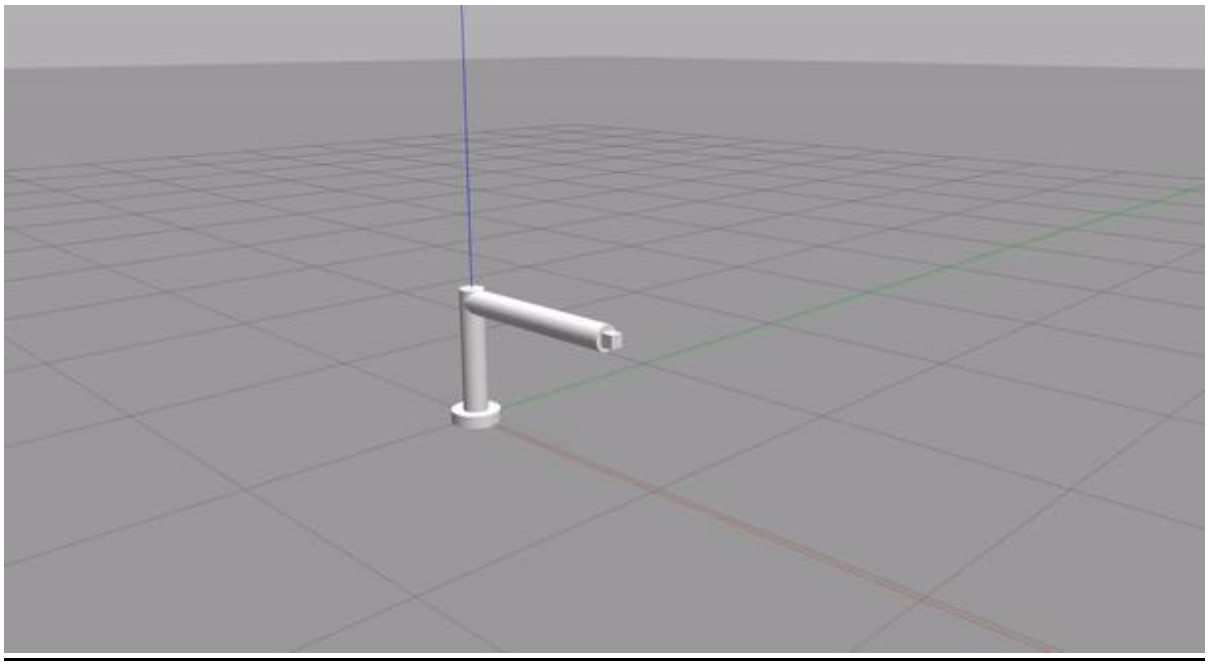
LETTER H:



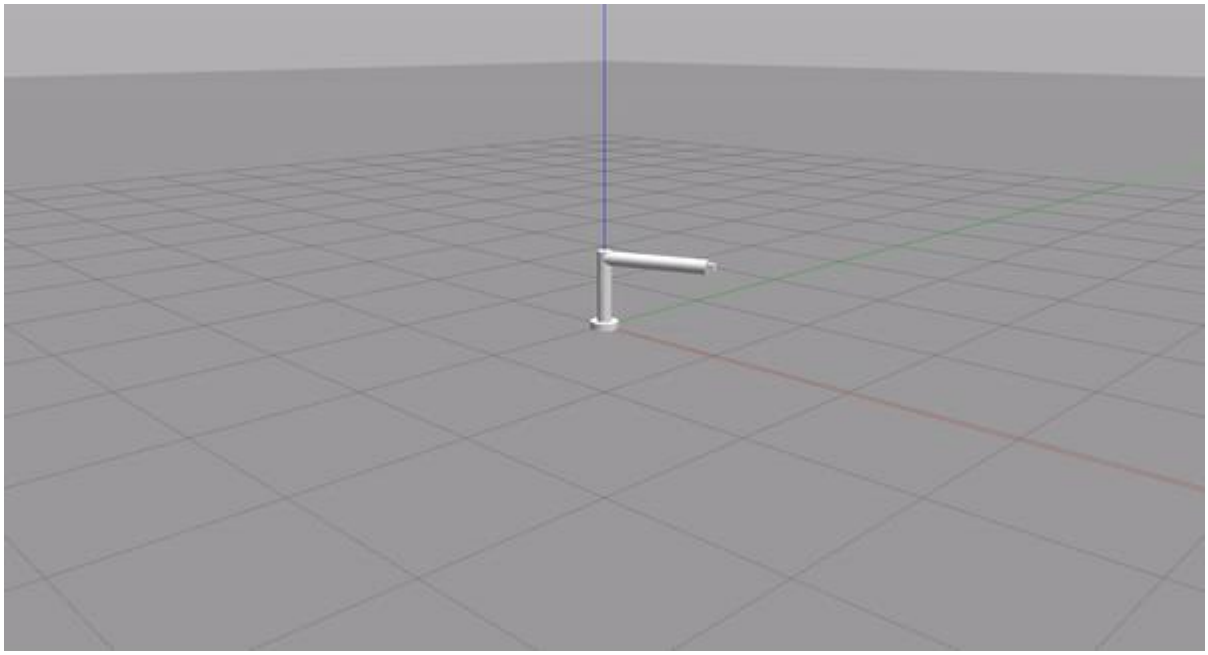
LETTER E:



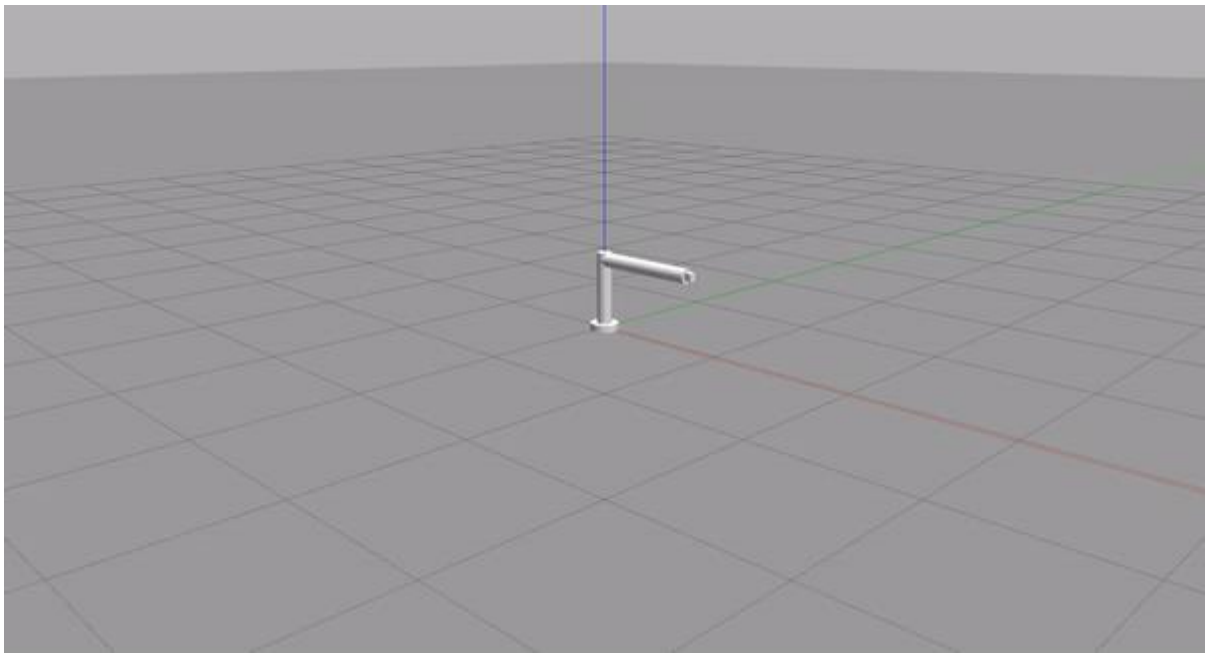
LETTER L:



LETTER T:



LETTER F:



CONCLUSION:

Thus, we have implemented an alphabet tracing manipulator of 4 degrees of freedom and 4 joints with the help of the co-ordinates tracing out alphabets was also done.

As with any operating system, the benefit of ROS is the hardware abstraction and its ability to control a robot without the user having to know all of the details of the robot. ROS lowers the technical level required to work on robotics projects. This can make it easier to design complex systems more quickly

REFERENCES

1. https://robots.ros.org/category/manipulator/
2. http://www.techno-press.org/fulltext/j_arr/arr2_2/arr0202001.pdf
3. https://automaticaddison.com/how-to-build-a-simulated-mobile-manipulator-using-ros/
4. https://www.theconstructsim.com/mastering-ros-robot-manipulators/#:~:text=Description,them%20to%20the%20proper%20location.