Forward Kinematics

Objective 1: Custom Forward Kinematics Node

Introduction: In this task we have to make a custom node that takes joint angles as input from a topic and publishes output as end effector position on a topic.

Code:

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

import rospy
from sensor_msgs.msg import JointState
import numpy as np
from robot_controller.msg import Position
from geometry_msgs.msg import Pose
import math

class Manipulator:
    def __init__(self):
        sub = rospy.Subscriber("/joint_states1", JointState, self.callback)
        self.pub = rospy.Publisher("/end_effector_position", Pose,
queue_size=10)
        self.data_file = "end_effector_positions.txt"

        # Initialize an empty list to store end effector positions
        self.end_effector_positions = []

    def callback(self, msg):
        self.arr = msg.position
        self.end_effector_pos(self.arr)


    def store_end_effector_position(self, array):
        # Append the current end effector position to the list
        self.end_effector_positions.append(array)

        # Write the list to the data file (clearing the file first)
        with open(self.data_file, "w") as file:
            for position in self.end_effector_positions:
                file.write(",".join(map(str, position)) + "\n")

    def end_effector_pos(self,array):
        # msg=Position()
        msg=Pose()


theta=[array[0],(math.pi/2)-0.1853-array[1],-(math.pi/2)-array[2]+0.1853,
-array[3]]

        alpha=[0,math.pi/2,0,0]
        a=[0.012,0,0.130,0.124]
        d=[0.077,0,0,0]
        #
theta=[array[2],(math.pi/2)-0.1853-array[3],-(math.pi/2)-array[4]+0.1853,
-array[5]]

        Ttemp=np.eye(4)

        for i in range(4):
            Tim=[[np.cos(theta[i]),-np.sin(theta[i]),0,a[i]],

[np.sin(theta[i])*np.cos(alpha[i]),np.cos(theta[i])*np.cos(alpha[i]),-np.sin(
alpha[i]),-d[i]*np.sin(alpha[i])],

[np.sin(theta[i])*np.sin(alpha[i]),np.cos(theta[i])*np.sin(alpha[i]),np.cos(al
pha[i]),d[i]*np.cos(alpha[i])],
             [0,0,0,1]]

            Ttemp=np.dot(Ttemp,Tim)


        p54=np.array([[0.126],
                    [0],
                    [0],
                    [1]])
        p50=np.dot(Ttemp,p54)

        msg.position.x=p50[0]
        msg.position.y=p50[1]
        msg.position.z=p50[2]
        pos_arr=[msg.position.x ,msg.position.y ,msg.position.z]
        self.store_end_effector_position(pos_arr)
        self.pub.publish(msg)
        rospy.sleep(0.001)

if __name__ == '__main__':
    try:
        rospy.init_node("manipulator_F_Kin")
        man = Manipulator()
        rospy.spin()
    except rospy.ROSInterruptException:
        pass
    print("done")
```

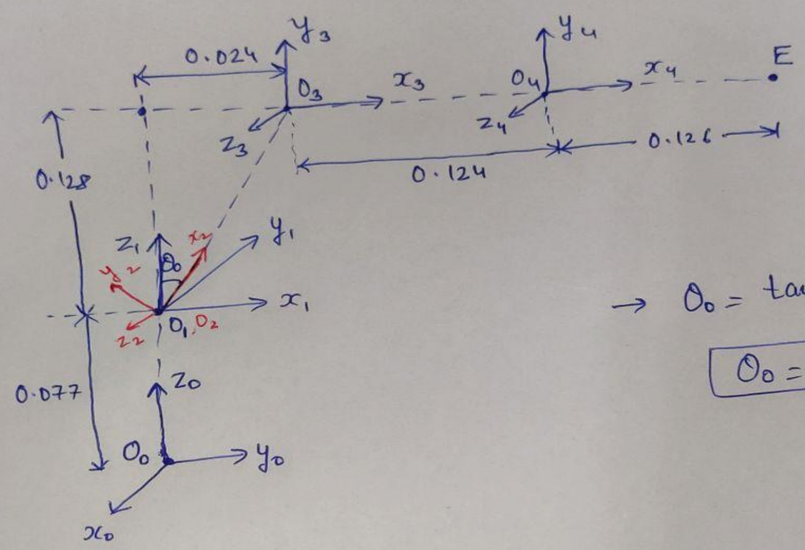
Code Explanation:

1. We are taking input from a topic /joint_states1.
2. It will contain an array of positions provided by the user.
3. Then we will calculate the modified DH parameters using frames and conventions.
4. After writing a forward kinematics node which will give the Transformation of the last frame with respect to the origin frame, we will transform and get the end effector position.

5. Publish the end effector position to the topic /end_effector_position.

DH Parameters Calculations and Frame Assignment:

Frame Assignment



$$\rightarrow \theta_0 = \tan^{-1}\left(\frac{0.024}{0.128}\right)$$

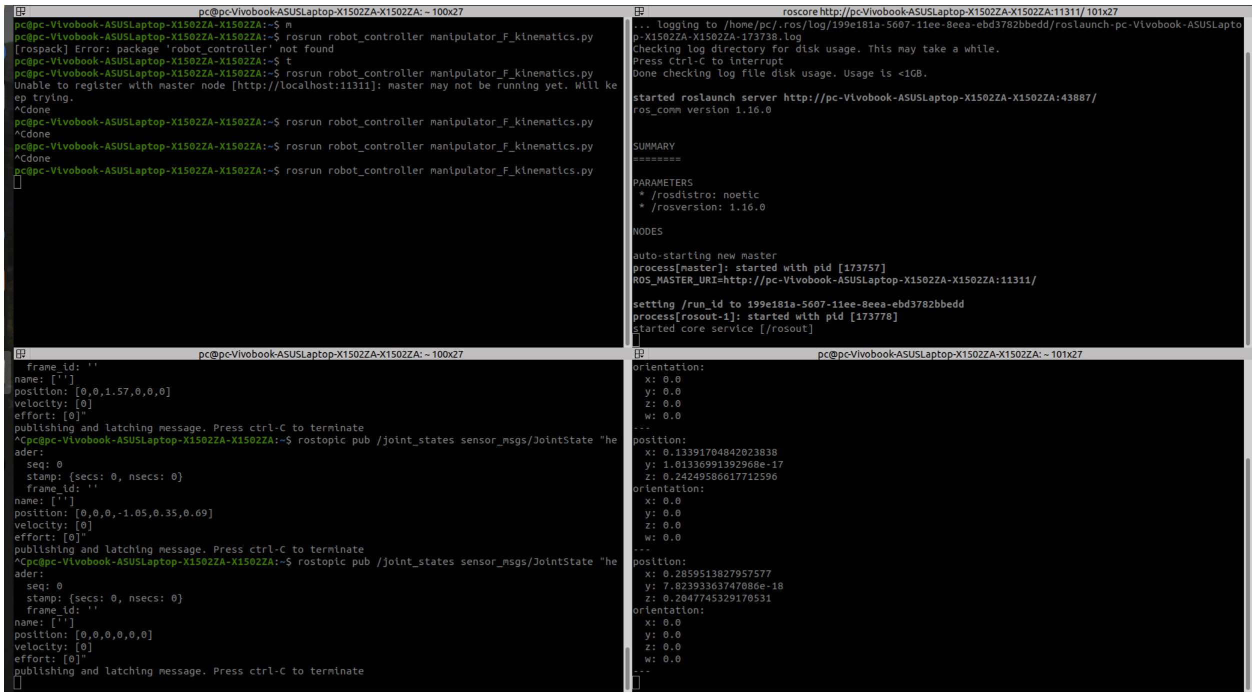
$$\boxed{\theta_0 = 0.1853^c}$$

DH Parameters

i	α_{i-1}	a_{i-1}	d_i	θ_i
1	0	0.012	0.077	θ_1
2	$\frac{\pi}{2}$	0	0	$\left(\frac{\pi}{2} - \theta_0\right) + \theta_2$
3	0	0.130	0	$\left(\frac{-\pi}{2} + \theta_0\right) + \theta_3$
4	0	0.124	0	θ_4

Here (0.012) is added due to offset in origin of gazebo

Output on Terminal:



pc@pc-Vivobook-ASUSLaptop-X1502ZA-X1502ZA: ~ 100x27

```
pc@pc-Vivobook-ASUSLaptop-X1502ZA-X1502ZA:~$ m
pc@pc-Vivobook-ASUSLaptop-X1502ZA-X1502ZA:~$ rosrun robot_controller manipulator_F_kinematics.py
[rospack] Error: package 'robot_controller' not found
pc@pc-Vivobook-ASUSLaptop-X1502ZA-X1502ZA:~$ t
pc@pc-Vivobook-ASUSLaptop-X1502ZA-X1502ZA:~$ rosrun robot_controller manipulator_F_kinematics.py
Unable to register with master node [http://localhost:11311]: master may not be running yet. Will ke
ep trying.
^Cdone
pc@pc-Vivobook-ASUSLaptop-X1502ZA-X1502ZA:~$ rosrun robot_controller manipulator_F_kinematics.py
^Cdone
pc@pc-Vivobook-ASUSLaptop-X1502ZA-X1502ZA:~$ rosrun robot_controller manipulator_F_kinematics.py
^Cdone
pc@pc-Vivobook-ASUSLaptop-X1502ZA-X1502ZA:~$ rosrun robot_controller manipulator_F_kinematics.py
```

roscore http://pc-Vivobook-ASUSLaptop-X1502ZA-X1502ZA:11311/ 101x27

```
... logging to /home/pc/.ros/log/199e181a-5607-11ee-8eea-ebd3782bbedd/roslaunch-pc-Vivobook-ASUSLapto
p-X1502ZA-X1502ZA-173738.log
Checking log directory for disk usage. This may take a while.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://pc-Vivobook-ASUSLaptop-X1502ZA-X1502ZA:43887/
ros_comm version 1.16.0


SUMMARY
========

PARAMETERS
 * /rosdistro: noetic
 * /rosversion: 1.16.0

NODES

auto-starting new master
process[master]: started with pid [173757]
ROS_MASTER_URI=http://pc-Vivobook-ASUSLaptop-X1502ZA-X1502ZA:11311/

setting /run_id to 199e181a-5607-11ee-8eea-ebd3782bbedd
process[rosout-1]: started with pid [173778]
started core service [/rosout]
```

pc@pc-Vivobook-ASUSLaptop-X1502ZA-X1502ZA: ~ 100x27

```
  frame_id: ''
name: ['']
position: [0,0,1.57,0,0,0]
velocity: [0]
effort: [0]"
publishing and latching message. Press ctrl-C to terminate
^Cpc@pc-Vivobook-ASUSLaptop-X1502ZA-X1502ZA:~$ rostopic pub /joint_states sensor_msgs/JointState "he
ader:
  seq: 0
  stamp: {secs: 0, nsecs: 0}
  frame_id: ''
name: ['']
position: [0,0,0,-1.05,0.35,0.69]
velocity: [0]
effort: [0]"
publishing and latching message. Press ctrl-C to terminate
^Cpc@pc-Vivobook-ASUSLaptop-X1502ZA-X1502ZA:~$ rostopic pub /joint_states sensor_msgs/JointState "he
ader:
  seq: 0
  stamp: {secs: 0, nsecs: 0}
  frame_id: ''
name: ['']
position: [0,0,0,0,0,0]
velocity: [0]
effort: [0]"
publishing and latching message. Press ctrl-C to terminate
```

pc@pc-Vivobook-ASUSLaptop-X1502ZA-X1502ZA: ~ 101x27

```
orientation:
  x: 0.0
  y: 0.0
  z: 0.0
  w: 0.0
---
position:
  x: 0.13391704842023838
  y: 1.01336991392968e-17
  z: 0.24249586617712596
orientation:
  x: 0.0
  y: 0.0
  z: 0.0
  w: 0.0
---
position:
  x: 0.2859513827957577
  y: 7.82393363747086e-18
  z: 0.2047745329170531
orientation:
  x: 0.0
  y: 0.0
  z: 0.0
  w: 0.0
---
```

Steps to run it:

1. Run the node using command:

   rosrun robot_controller manipulator_F_kinematics.py

2. Now give input from terminal to the topic using command:

   rostopic pub /joint_states sensor_msgs/JointState

   "Header:

   seq: 0

   stamp: {secs: 0, nsecs: 0}

   frame_id: ''

   name: ['']

   position: [0,0,0,0,0,0]

   velocity: [0]

   effort: [0]"

3. To get end effector position use:

   rostopic echo /end_effector_position

Video Link: https://youtu.be/-7lQpL-xB1k

Conclusion:

1. The end effector position is published on the desired topic.
2. The validity of the values will be checked in the next part.

Objective 2: Gazebo Simulation for verification of above node

Introduction: Here a custom node is made which links with the above node through a topic. With this node we take a list of joint positions from a text file and the required output is published on output topic to visualize the results on gazebo simulator. The output is the end position given by our first node and it is matched with the values given by gazebo. When the simulator reached the correct position we print **“reached”.**

Code:

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

import rospy
from std_msgs.msg import Float64
from sensor_msgs.msg import JointState
import os
import re  # Required for regular expressions

class control:
    def __init__(self):
        self.pub = [rospy.Publisher("/joint1_position/command", Float64, queue_size=10),
                    rospy.Publisher("/joint2_position/command", Float64, queue_size=10),
                    rospy.Publisher("/joint3_position/command", Float64, queue_size=10),
                    rospy.Publisher("/joint4_position/command", Float64, queue_size=10)]
        self.pub1 = rospy.Publisher("/joint_states1", JointState, queue_size=10)

    def publish(self, arr):
        msg1 = JointState()
        msg1.position = arr

        for i in range(4):
            if i < len(arr):
                msg = Float64()
                msg.data = arr[i]
                self.pub[i].publish(msg)
                rospy.sleep(0.5)

        self.pub1.publish(msg1)


if __name__ == "__main__":
    rospy.init_node("controller")

    obj = control()

    # Specify the file name
    file_name = "input.txt"  # Replace with the name of your input file

    # Get the path to the file
    file_path = os.path.join(os.path.dirname("/home/pc/turtlesim_ws/src/robot_controller"),
file_name)

    input_arrays = []  # List to store all arrays/lines from the file

    try:
        with open(file_path, "r") as file:
            lines = file.readlines()
            for line in lines:
                try:
                    # Use regular expressions to extract values enclosed in square brackets on
each line
                    values_str = re.findall(r'[-+]?\d*\.\d+|[-+]?\d+', line)
                    if values_str:
                        # Convert the values to floats
                        input_values = [float(num) for num in values_str]

                        # Append the array to the list
                        input_arrays.append(input_values)

                except ValueError:
                    rospy.logerr(f"Error parsing line: {line.strip()}")

    except FileNotFoundError:
        rospy.logerr(f"File not found: {file_path}")
    except Exception as e:
        rospy.logerr(f"Error reading the file: {str(e)}")

    # Iterate through the list and publish each array one by one
    for input_values in input_arrays:
        rate = rospy.Rate(5)

        obj.publish(input_values)  # Publish the values

        # Wait for 3 seconds
        rospy.sleep(2)

        # Print "reached"
        print("Reached")
```

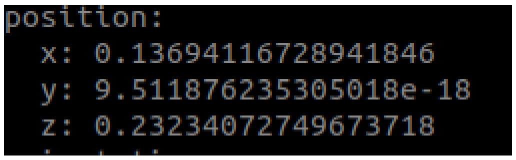
Code Explanation:

1. We are taking a set of joint position values from a text file.
2. These values are fed to the gazebo simulator and our node made above.
3. Our custom node will take these values and calculate the final end effector position.
4. In gazebo simulator the arm will move to the desired position and publishes the output end effector position on topic /gripper_kinematic_pose.
5. Then we will know if the values are the same or not.

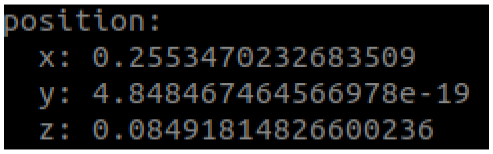
Text File Input(Joint Angles):
[0 -1.05 0.39 0.70]
[0 0 0.5 0]
[0 0 0 1.57]
[0 0 0 0]

True End Effector Positions corresponding to above joint angles:
[0.1369 0 0.2323]
[0.255 0 0.0849]
[0.160 0 0.0787]
[0.2859 0 0.2047]

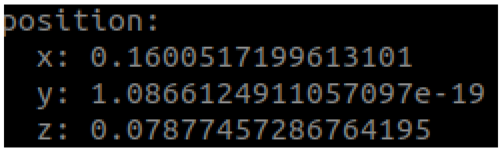
Positions got from code:(On **\end_effector_position** topic**)**



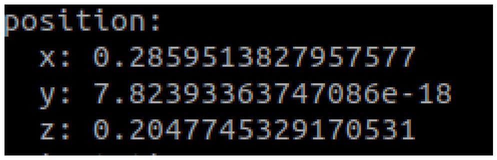
```
position:
  x: 0.13694116728941846
  y: 9.511876235305018e-18
  z: 0.23234072749673718
```



```
position:
  x: 0.2553470232683509
  y: 4.848467464566978e-19
  z: 0.08491814826600236
```

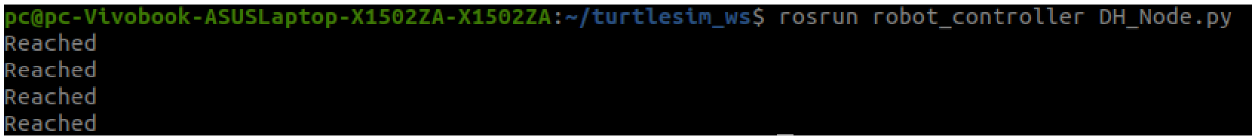


```
position:
  x: 0.1600517199613101
  y: 1.0866124911057097e-19
  z: 0.07877457286764195
```



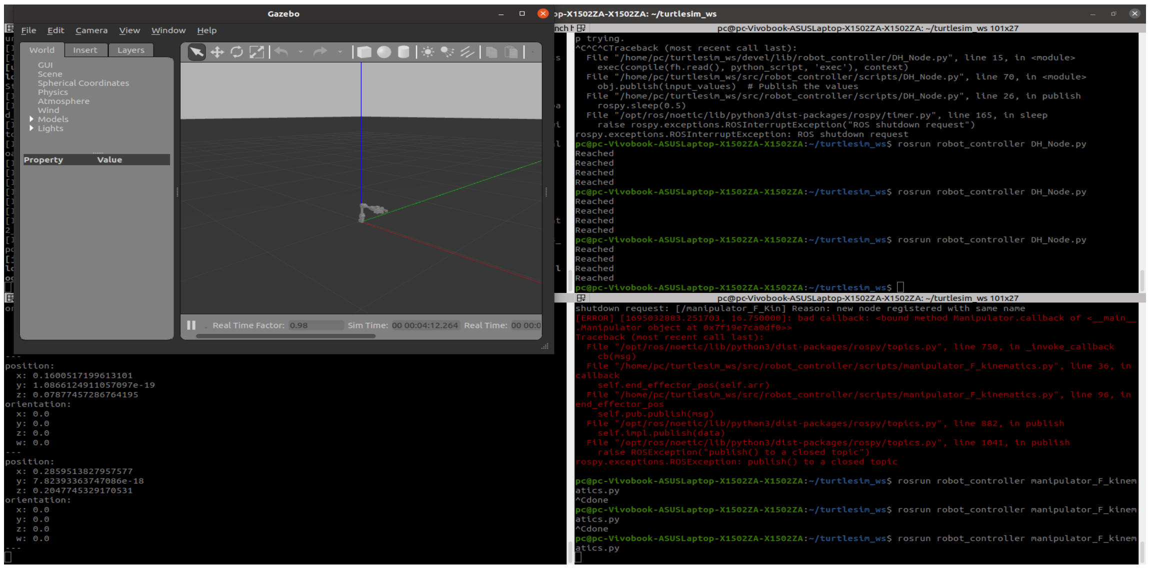
```
position:
  x: 0.2859513827957577
  y: 7.82393363747086e-18
  z: 0.2047745329170531
```

After end effector is reached:



```
pc@pc-Vivobook-ASUSLaptop-X1502ZA-X1502ZA:~/turtlesim_ws$ rosrun robot_controller DH_Node.py
Reached
Reached
Reached
Reached
```

Terminal:

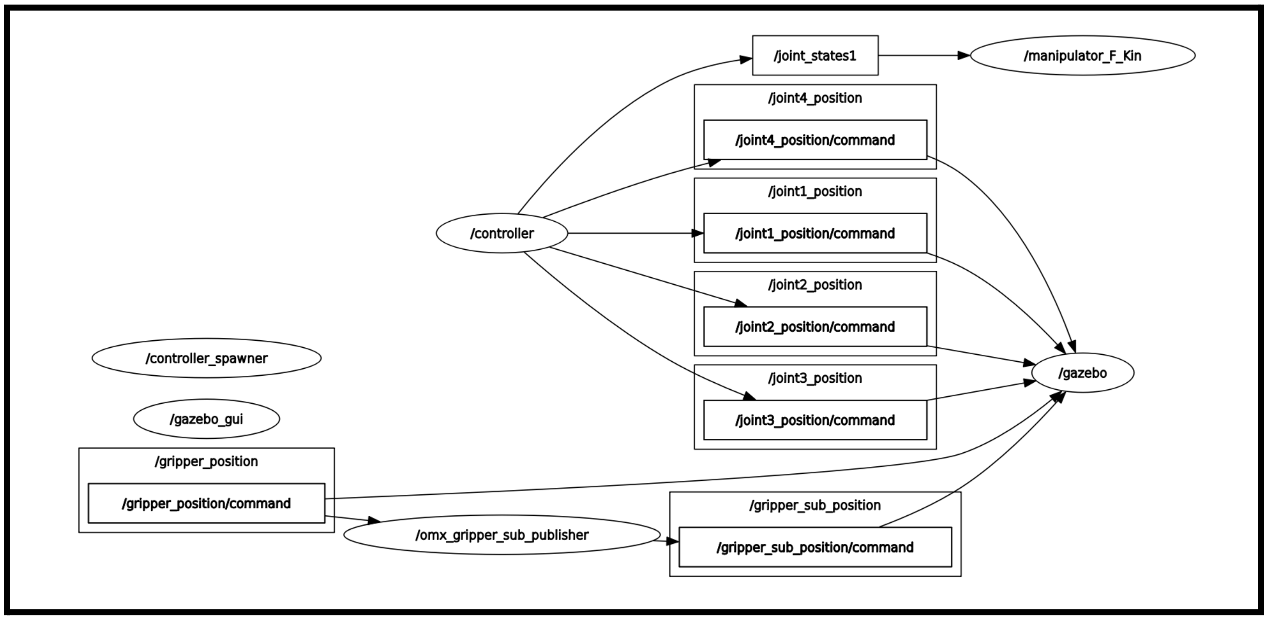


```
---
position:
  x: 0.1600517199613101
  y: 1.0866124911057097e-19
  z: 0.07877457286764195
orientation:
  x: 0.0
  y: 0.0
  z: 0.0
  w: 0.0
---
position:
  x: 0.2859513827957577
  y: 7.82393363747086e-18
  z: 0.2047745329170531
orientation:
  x: 0.0
  y: 0.0
  z: 0.0
  w: 0.0
---
```

```
pc@pc-Vivobook-ASUSLaptop-X1502ZA-X1502ZA: ~/turtlesim_ws 101x27
p trying.
^C^C^CTraceback (most recent call last):
  File "/home/pc/turtlesim_ws/devel/lib/robot_controller/DH_Node.py", line 15, in <module>
    exec(compile(fh.read(), python_script, 'exec'), context)
  File "/home/pc/turtlesim_ws/src/robot_controller/scripts/DH_Node.py", line 70, in <module>
    obj.publish(input_values)  # Publish the values
  File "/home/pc/turtlesim_ws/src/robot_controller/scripts/DH_Node.py", line 26, in publish
    rospy.sleep(0.5)
  File "/opt/ros/noetic/lib/python3/dist-packages/rospy/timer.py", line 165, in sleep
    raise rospy.exceptions.ROSInterruptException("ROS shutdown request")
rospy.exceptions.ROSInterruptException: ROS shutdown request
pc@pc-Vivobook-ASUSLaptop-X1502ZA-X1502ZA:~/turtlesim_ws$ rosrun robot_controller DH_Node.py
Reached
Reached
Reached
Reached
pc@pc-Vivobook-ASUSLaptop-X1502ZA-X1502ZA:~/turtlesim_ws$ rosrun robot_controller DH_Node.py
Reached
Reached
Reached
Reached
pc@pc-Vivobook-ASUSLaptop-X1502ZA-X1502ZA:~/turtlesim_ws$ rosrun robot_controller DH_Node.py
Reached
Reached
Reached
Reached
pc@pc-Vivobook-ASUSLaptop-X1502ZA-X1502ZA:~/turtlesim_ws$ 
```

```
pc@pc-Vivobook-ASUSLaptop-X1502ZA-X1502ZA: ~/turtlesim_ws 101x27
shutdown request: [/manipulator_F_Kin] Reason: new node registered with same name
[ERROR] [1695032883.251703, 16.750000]: bad callback: <bound method Manipulator.callback of <__main__
.Manipulator object at 0x7f19e7ca0df0>>
Traceback (most recent call last):
  File "/opt/ros/noetic/lib/python3/dist-packages/rospy/topics.py", line 750, in _invoke_callback
    cb(msg)
  File "/home/pc/turtlesim_ws/src/robot_controller/scripts/manipulator_F_kinematics.py", line 36, in
callback
    self.end_effector_pos(self.arr)
  File "/home/pc/turtlesim_ws/src/robot_controller/scripts/manipulator_F_kinematics.py", line 96, in
end_effector_pos
    self.pub.publish(msg)
  File "/opt/ros/noetic/lib/python3/dist-packages/rospy/topics.py", line 882, in publish
    self.impl.publish(data)
  File "/opt/ros/noetic/lib/python3/dist-packages/rospy/topics.py", line 1041, in publish
    raise ROSException("publish() to a closed topic")
rospy.exceptions.ROSException: publish() to a closed topic

pc@pc-Vivobook-ASUSLaptop-X1502ZA-X1502ZA:~/turtlesim_ws$ rosrun robot_controller manipulator_F_kinem
atics.py
^Cdone
pc@pc-Vivobook-ASUSLaptop-X1502ZA-X1502ZA:~/turtlesim_ws$ rosrun robot_controller manipulator_F_kinem
atics.py
^Cdone
pc@pc-Vivobook-ASUSLaptop-X1502ZA-X1502ZA:~/turtlesim_ws$ rosrun robot_controller manipulator_F_kinem
atics.py
```

rqt_graph:



Steps to run it:

1. Run gazebo simulator using command:
   roslaunch open_manipulator_gazebo open_manipulator_gazebo.launch

2. Run the first node to calculate end effector position using:
   rosrun robot_controller manipulator_F_kinematics.py

3. Then run second node to move the arm in simulator using:
   rosrun robot_controller DH_Node.py

4. End effector position is also saved in text file using above code. So we can also verify manually if they are correct or not.

Video Link: https://youtu.be/UOSpD1RQFgU

Conclusion:

1. We can see from the results that our end effector is reaching the desired position.
2. By calculating correct DH parameters we got the correct end effector position.

Inverse Kinematics

Objective 1: Custom Inverse Kinematics Node.

Introduction: Inverse kinematics basically involves the calculation of joint angles when end effector positions are given to us.
In our case input will be an array of end effector positions and phi(sum of angles of joint 2,3 and 4).

Code:

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

import rospy
from std_msgs.msg import Float64MultiArray
from math import atan2, sqrt, cos, sin, pi
import numpy as np
import math



class InverseKinematics:
   def __init__(self):
       rospy.init_node('calculate_joint_angles')
       # print("1")
       self.joint_angles_pub = rospy.Publisher('/joint_angles',
Float64MultiArray, queue_size=10)
       # print("2")
       self.end_effector_sub = rospy.Subscriber('/end_effector_pose',
Float64MultiArray, self.end_effector_callback)



   def end_effector_callback(self, pose_msg):
       # Extract end effector position (px, py) and angle phi from the
message

       px, py, pz, phi= pose_msg.data

       def P3R_inverse_kinematics(a1, a2, a3, x3, y3, theta):
           ## Inverse Kinematics of planar 3R robot
```

```
            x2 = x3 - a3*np.cos(theta)
            y2 = y3 - a3*np.sin(theta)

            cos_th2 = (x2**2+y2**2-a1**2-a2**2)/(2*a1*a2)
            # print(cos_th2)
            sin_th2 = [-np.sqrt(1-cos_th2**2), np.sqrt(1-cos_th2**2)]

            th2 = [np.arctan2(sin_th2[0], cos_th2),np.arctan2(sin_th2[1],
cos_th2)]

            sin_th1 =
[(y2*(a1+a2*np.cos(th2[0]))-a2*np.sin(th2[0])*x2)/(a1**2+a2**2+2*a1*a2*np.
cos(th2[0])),

(y2*(a1+a2*np.cos(th2[1]))-a2*np.sin(th2[1])*x2)/(a1**2+a2**2+2*a1*a2*np.c
os(th2[1]))]

            cos_th1 =
[(x2*(a1+a2*np.cos(th2[0]))+a2*np.sin(th2[0])*y2)/(a1**2+a2**2+2*a1*a2*np.
cos(th2[0])),

(x2*(a1+a2*np.cos(th2[1]))+a2*np.sin(th2[1])*y2)/(a1**2+a2**2+2*a1*a2*np.c
os(th2[1]))]

            th1 = [np.arctan2(sin_th1[0],
cos_th1[0]),np.arctan2(sin_th1[1], cos_th1[1])]

            th3 = [theta-th1[0]-th2[0], theta-th1[1]-th2[1]]

            return th1, th2, th3

        def inverse_kinematics(target_pos, target_rot=None,
target_phi=None):

            x = target_pos[0]
            y = target_pos[1]
            z = target_pos[2]

            d1 =0.077
            a1 = np.sqrt(0.024**2+0.128**2)
            alpha_2 = np.arctan(0.024/0.128)
            # print(np.rad2deg(alpha_2))
            a2 = 0.124
            a3 = 0.126

            x_new = np.sqrt(x**2+y**2)
            y_new = z-d1
            if target_phi is None:
                phi = calculate_angle_with_xy_plane(target_rot)
            else:
                phi = -target_phi

            theta_1 = [np.arctan2(y, x), np.arctan2(-y, -x)]
            thetas = []
            for i in range(1):
                th2, th3, th4 = P3R_inverse_kinematics(a1=a1,
                                                        a2=a2,
                                                        a3=a3,
                                                        x3=x_new,
                                                        y3=y_new,
                                                        theta=phi)



                theta_2 = [np.pi/2-th2[0]-alpha_2, np.pi/2-th2[1]-alpha_2]
                theta_3 = [-np.pi/2-th3[0]+alpha_2,
-np.pi/2-th3[1]+alpha_2]
                theta_4 = [-th4[0], -th4[1]]

                thetas.append([theta_1[i], theta_2[0], theta_3[0],
theta_4[0]])
                thetas.append([theta_1[i], theta_2[1], theta_3[1],
theta_4[1]])

            return thetas


        target_phi = phi
        target_pos = [px-0.012,py,pz]
        # target_pos[0] -= 0.012
        # print("Joint_Angles_Calculation:")
        joint_angles = np.array(inverse_kinematics(target_pos,
target_phi=target_phi))


        # Create a Float64MultiArray message to publish the joint angles
        joint_angles_msg = Float64MultiArray(data=joint_angles[0])
        # joint_angles_msg =
Float64MultiArray(data=[(theta11),(theta21),(theta31)])

        # Publish the joint angles
        # print("Joint_Angles:")
        rospy.sleep(1)
        self.joint_angles_pub.publish(joint_angles_msg)

if __name__ == '__main__':
    try:
        node = InverseKinematics()
        rospy.spin()
    except rospy.ROSInterruptException:
        pass
```

Calculations and Explanation of code:

1. First we have calculated the value of theta1(joint angle value of first joint). As we know that when our manipulator moves it always remains in plane. So using this concept we have calculated the value of theta1 using formula:

   tan(theta1)=py/px

   Where py and px are the given end effector positions.

2. Now we are in a single plane. We have to calculate the value of th2,th3 and th4 in this plane. So we have started with finding the end effector position in this plane. And we call this plane the r,z plane.

   x_new = np.sqrt(x**2+y**2)
   y_new = z-d1

3. Then we will find the value theta by using:

   cos_th2 = (x2**2+y2**2-a1**2-a2**2)/(2*a1*a2)
   sin_th2 = [-np.sqrt(1-cos_th2**2), np.sqrt(1-cos_th2**2)]
   th2 = [np.arctan2(sin_th2, cos_th2),np.arctan2(sin_th2, cos_th2)]

   This is basic cosine law applied to find th2 angle.
4. Now we will find the value of th1 using below formulae:

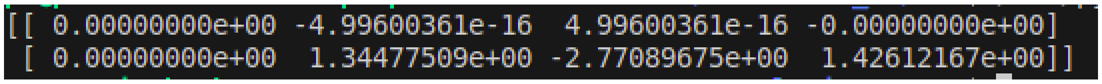
   sin(θ1) = (y2 * (a1 + a2 * cos(θ2)) - a2 * sin(θ2) * x2) / (a1^2 + a2^2 + 2 * a1 * a2 * cos(θ2))
   cos(θ1) = (x2 * (a1 + a2 * cos(θ2)) + a2 * sin(θ2) * y2) / (a1^2 + a2^2 + 2 * a1 * a2 * cos(θ2))
   θ1 = arctan2(sin(θ1), cos(θ1))

5. Now we will th3 using:
   θ3 = θ - θ1 - θ2

6. Mapping from theta angles to gazebo joint angles:

   theta_2 = [np.pi/2-th2-alpha_2]
   theta_3 = [-np.pi/2-th3+alpha_2]
   theta_4 = [-th4]

7. We got 2 solutions for each value of joint angles. But for giving the values to gazebo we have taken only one value that matches with the original solution.

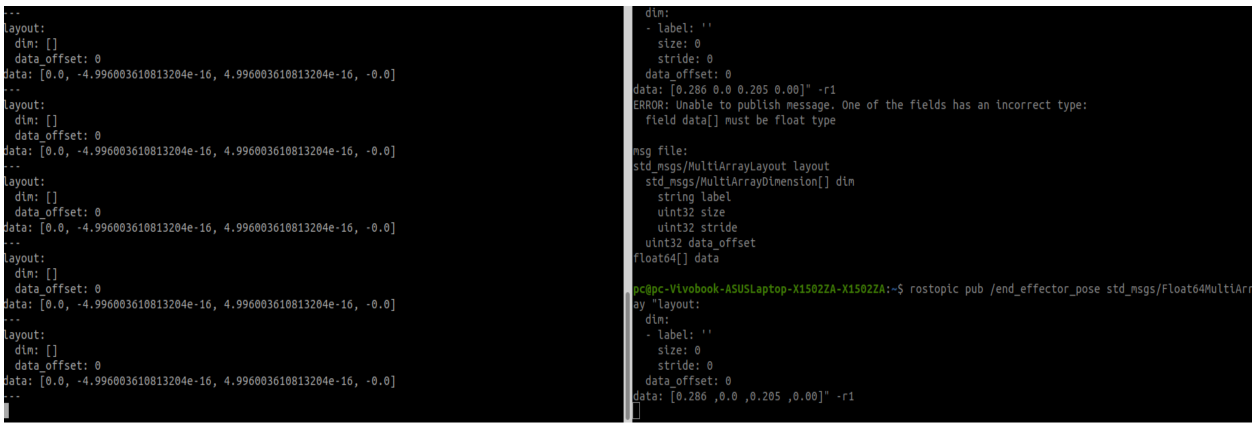


```
[[ 0.00000000e+00 -4.99600361e-16  4.99600361e-16 -0.00000000e+00]
 [ 0.00000000e+00  1.34477509e+00 -2.77089675e+00  1.42612167e+00]]
```

Results:

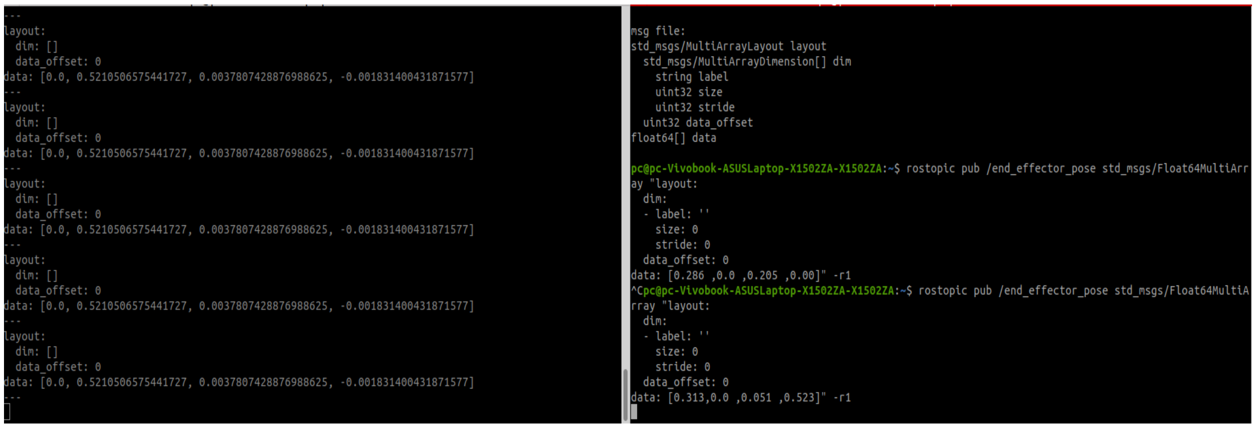
1. I have recorded some values from gazebo and tested this node on those values:

J1	J2	J3	J4	End Effector Position
0	0	0	0	[0.286,0,0.205]
0	0.523	0	0	[0.313,0,0.051]
0.780	0.523	-0.523	-1.570	[0.160,0.147,0.301]



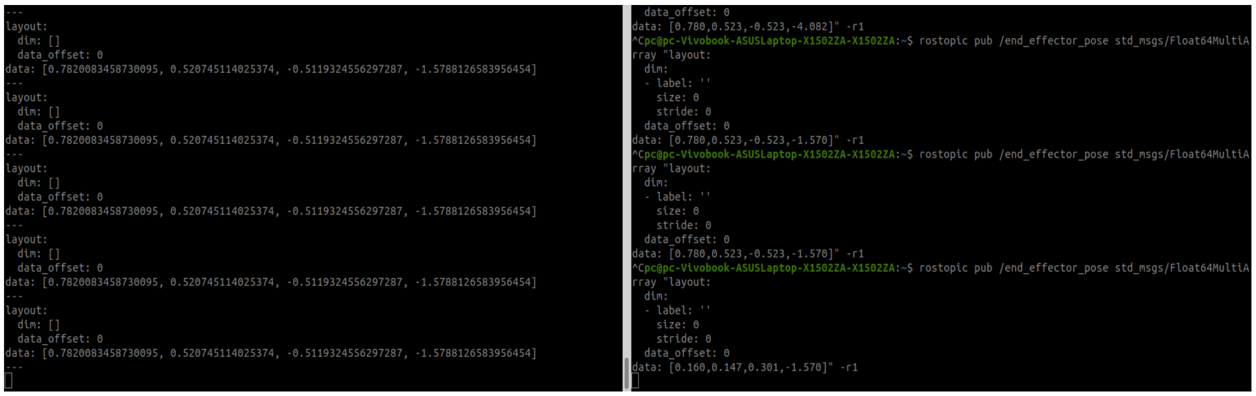
```
---
layout:
  dim: []
  data_offset: 0
data: [0.0, -4.996003610813204e-16, 4.996003610813204e-16, -0.0]
---
layout:
  dim: []
  data_offset: 0
data: [0.0, -4.996003610813204e-16, 4.996003610813204e-16, -0.0]
---
layout:
  dim: []
  data_offset: 0
data: [0.0, -4.996003610813204e-16, 4.996003610813204e-16, -0.0]
---
layout:
  dim: []
  data_offset: 0
data: [0.0, -4.996003610813204e-16, 4.996003610813204e-16, -0.0]
---
layout:
  dim: []
  data_offset: 0
data: [0.0, -4.996003610813204e-16, 4.996003610813204e-16, -0.0]
---
```

```
  dim:
  - label: ''
    size: 0
    stride: 0
  data_offset: 0
data: [0.286 0.0 0.205 0.00]" -r1
ERROR: Unable to publish message. One of the fields has an incorrect type:
  field data[] must be float type

msg file:
std_msgs/MultiArrayLayout layout
  std_msgs/MultiArrayDimension[] dim
    string label
    uint32 size
    uint32 stride
  uint32 data_offset
float64[] data

pc@pc-Vivobook-ASUSLaptop-X1502ZA-X1502ZA:~$ rostopic pub /end_effector_pose std_msgs/Float64MultiArr
ay "layout:
  dim:
  - label: ''
    size: 0
    stride: 0
  data_offset: 0
data: [0.286 ,0.0 ,0.205 ,0.00]" -r1
```



```
---
layout:
  dim: []
  data_offset: 0
data: [0.0, 0.5210506575441727, 0.0037807428876988625, -0.001831400431871577]
---
layout:
  dim: []
  data_offset: 0
data: [0.0, 0.5210506575441727, 0.0037807428876988625, -0.001831400431871577]
---
layout:
  dim: []
  data_offset: 0
data: [0.0, 0.5210506575441727, 0.0037807428876988625, -0.001831400431871577]
---
layout:
  dim: []
  data_offset: 0
data: [0.0, 0.5210506575441727, 0.0037807428876988625, -0.001831400431871577]
---
layout:
  dim: []
  data_offset: 0
data: [0.0, 0.5210506575441727, 0.0037807428876988625, -0.001831400431871577]
---
```

```
msg file:
std_msgs/MultiArrayLayout layout
  std_msgs/MultiArrayDimension[] dim
    string label
    uint32 size
    uint32 stride
  uint32 data_offset
float64[] data

pc@pc-Vivobook-ASUSLaptop-X1502ZA-X1502ZA:~$ rostopic pub /end_effector_pose std_msgs/Float64MultiArr
ay "layout:
  dim:
  - label: ''
    size: 0
    stride: 0
  data_offset: 0
data: [0.286 ,0.0 ,0.205 ,0.00]" -r1
^Cpc@pc-Vivobook-ASUSLaptop-X1502ZA-X1502ZA:~$ rostopic pub /end_effector_pose std_msgs/Float64MultiA
rray "layout:
  dim:
  - label: ''
    size: 0
    stride: 0
  data_offset: 0
data: [0.313,0.0 ,0.051 ,0.523]" -r1
```



```
---
layout:
  dim: []
  data_offset: 0
data: [0.7820083458730095, 0.520745114025374, -0.5119324556297287, -1.5788126583956454]
---
layout:
  dim: []
  data_offset: 0
data: [0.7820083458730095, 0.520745114025374, -0.5119324556297287, -1.5788126583956454]
---
layout:
  dim: []
  data_offset: 0
data: [0.7820083458730095, 0.520745114025374, -0.5119324556297287, -1.5788126583956454]
---
layout:
  dim: []
  data_offset: 0
data: [0.7820083458730095, 0.520745114025374, -0.5119324556297287, -1.5788126583956454]
---
layout:
  dim: []
  data_offset: 0
data: [0.7820083458730095, 0.520745114025374, -0.5119324556297287, -1.5788126583956454]
---
```

```
  data_offset: 0
data: [0.780,0.523,-0.523,-4.082]" -r1
^Cpc@pc-Vivobook-ASUSLaptop-X1502ZA-X1502ZA:~$ rostopic pub /end_effector_pose std_msgs/Float64MultiA
rray "layout:
  dim:
  - label: ''
    size: 0
    stride: 0
  data_offset: 0
data: [0.780,0.523,-0.523,-1.570]" -r1
^Cpc@pc-Vivobook-ASUSLaptop-X1502ZA-X1502ZA:~$ rostopic pub /end_effector_pose std_msgs/Float64MultiA
rray "layout:
  dim:
  - label: ''
    size: 0
    stride: 0
  data_offset: 0
data: [0.780,0.523,-0.523,-1.570]" -r1
^Cpc@pc-Vivobook-ASUSLaptop-X1502ZA-X1502ZA:~$ rostopic pub /end_effector_pose std_msgs/Float64MultiA
rray "layout:
  dim:
  - label: ''
    size: 0
    stride: 0
  data_offset: 0
data: [0.160,0.147,0.301,-1.570]" -r1
```

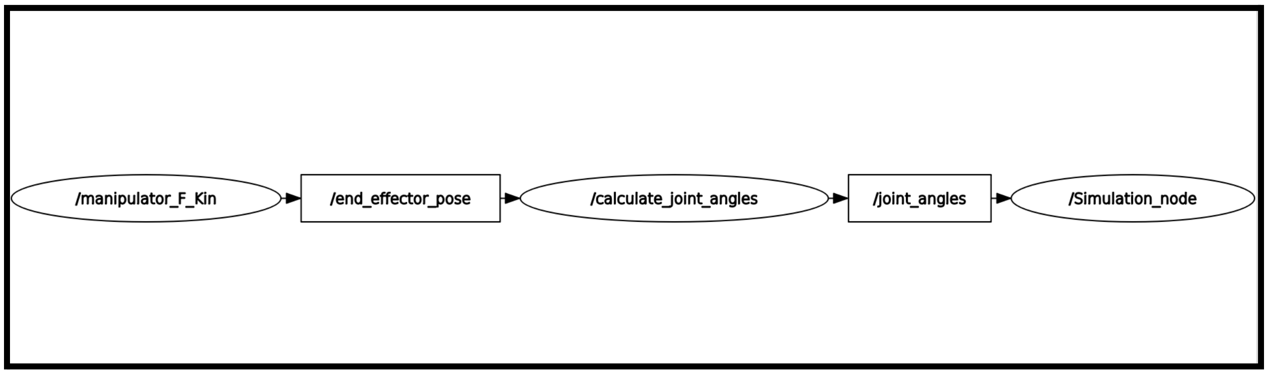
2. From the above values we can infer that we have got the same values for all the joint angles with our custom inverse kinematics node.

Objective2: Verification of the above node using gazebo and previous forward kinematics node.

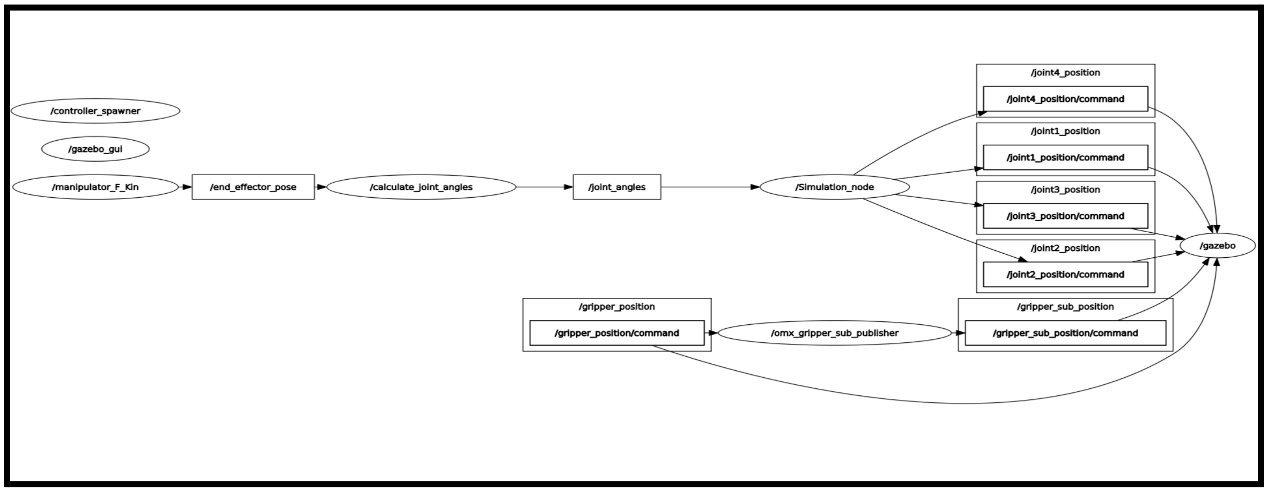
Introduction: In this part we have to verify the node by first making a list of end effector positions and phi. Then we will calculate the joint angles using the above node and send these joint angles to the gazebo. In this way we can check whether we have got correct values of joint angles or not.

Flow of Code:

1. When we are not running gazebo, and publishing joint angles directly on topic /joint_state2. The manipulator_F_Kin node is taking values of joint angles and finding the end effector position and publishing it on the end_effector_pose topic. Now these values of end effector and phi are taken by node calculate_jonint_nagles which our inverse kinematics node and it will calculate joint angles and publish values on topic /joint_angles, Then our simulation node will take these values and publish it on gazebo.



2. When we are using gazebo then graph will be as show below.



Code:

1. Simulation Node:

```
import rospy
from std_msgs.msg import Float64MultiArray, Float64

class manipulator:
   def __init__(self):
       # self.pub1 = rospy.Publisher("/end_effector_pose", Float64MultiArray,
queue_size=10)
       self.pub2 = [
           rospy.Publisher("/joint1_position/command", Float64, queue_size=10),
           rospy.Publisher("/joint2_position/command", Float64, queue_size=10),
           rospy.Publisher("/joint3_position/command", Float64, queue_size=10),
           rospy.Publisher("/joint4_position/command", Float64, queue_size=10)
       ]
       self.publishers()
       for pose in self.end_eff_pose:
           # Publish the current row of end_eff_pose to /end_effector_pose
           # print(pose)
           end_eff_pose_msg = Float64MultiArray(data=pose)
           # print("Repeat")
           rospy.sleep(1)
           # self.pub1.publish(end_eff_pose_msg)
           # rospy.wait_for_message("/joint_angles", Float64MultiArray)
           self.sub = rospy.Subscriber("/joint_angles", Float64MultiArray, self.callback)
       # rospy.spin()

   def publishers(self):
       self.end_eff_pose = [
           [0.347, 0, 0.175, 0],
           [0.2,-0.158,0.129,0.05],
           [0.286, 0, 0.205, 0]

       ]

   def callback(self, msg):
           self.theta1=msg.data[0]
           self.theta2=msg.data[1]
           self.theta3=msg.data[2]
           self.theta4=msg.data[3]

           # Assuming that you want to publish the same joint angles to all pub2 topics
           joint_angles_msg = Float64MultiArray(data=[self.theta1, self.theta2,
self.theta3, self.theta4])

           # Publish the received joint angles to self.pub2
           i=0
```

```
          for pub in self.pub2:
              pub.publish(joint_angles_msg.data[i])
              i=i+1
              rospy.sleep(0.1)
          rospy.sleep(1)

if __name__ == "__main__":
   rospy.init_node("Simulation_node")
   obj = manipulator()
   rospy.spin()
```

2.manipulator_F_Kin node

```
import rospy
from sensor_msgs.msg import JointState
import numpy as np
from robot_controller.msg import Position
from std_msgs.msg import Float64MultiArray
import math

class Manipulator:
   def __init__(self):
       sub = rospy.Subscriber("/joint_states2", Float64MultiArray, self.callback)
       self.pub = rospy.Publisher("/end_effector_pose", Float64MultiArray, queue_size=10)


       # Initialize an empty list to store end effector positions
       self.end_effector_positions = []

   def callback(self, msg):
       self.arr = msg.data
       self.phi=(msg.data[3]+msg.data[1]+msg.data[2])
       self.end_effector_pos(self.arr)
   def store_end_effector_position(self, array):
       # Append the current end effector position to the list
       self.end_effector_positions.append(array)



   def end_effector_pos(self,array):
       # msg=Position()
       msg=Float64MultiArray()


theta=[array[0],(math.pi/2)-0.1853-array[1],-(math.pi/2)-array[2]+0.1853,-array[3]]

       alpha=[0,math.pi/2,0,0]
       a=[0.012,0,0.130,0.124]
       d=[0.077,0,0,0]
       #
theta=[array[2],(math.pi/2)-0.1853-array[3],-(math.pi/2)-array[4]+0.1853,-array[5]]

       Ttemp=np.eye(4)

       for i in range(4):
           Tim=[[np.cos(theta[i]),-np.sin(theta[i]),0,a[i]],

[np.sin(theta[i])*np.cos(alpha[i]),np.cos(theta[i])*np.cos(alpha[i]),-np.sin(alpha[i]),-d
[i]*np.sin(alpha[i])],

[np.sin(theta[i])*np.sin(alpha[i]),np.cos(theta[i])*np.sin(alpha[i]),np.cos(alpha[i]),d[i
]*np.cos(alpha[i])],
                [0,0,0,1]]

           Ttemp=np.dot(Ttemp,Tim)



       p54=np.array([[0.126],
                    [0],
                    [0],
                    [1]])
       p50=np.dot(Ttemp,p54)
```

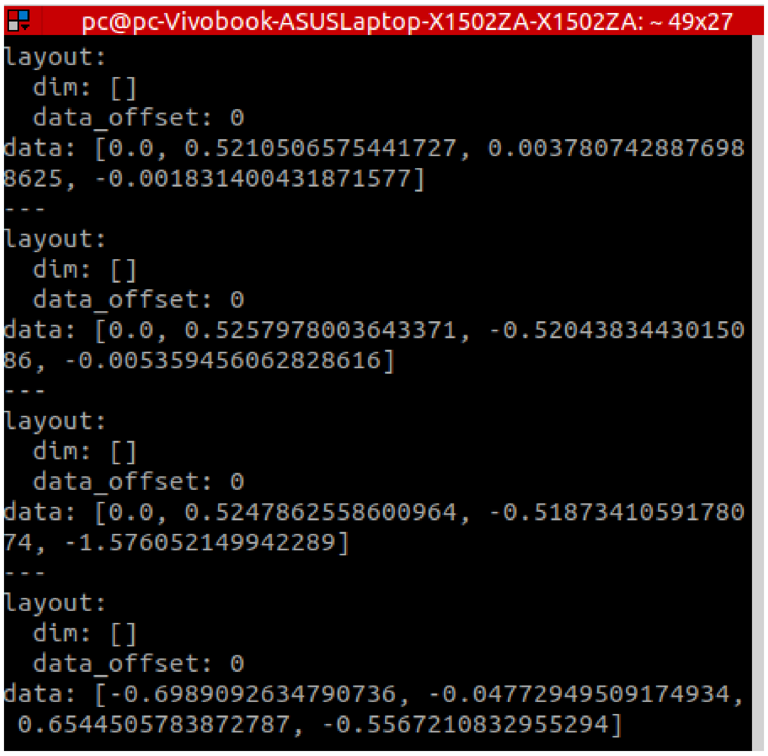
```
        msg.data.append(p50[0])
        msg.data.append(p50[1])
        msg.data.append(p50[2])
        msg.data.append(self.phi)
        # pos_arr=[msg.data[0] ,msg.data[1] ,msg.data[2],self.phi]

        # self.store_end_effector_position(pos_arr)
        self.pub.publish(msg)
        rospy.sleep(0.001)

if __name__ == '__main__':
    try:
        rospy.init_node("manipulator_F_Kin")
        man = Manipulator()
        rospy.spin()
    except rospy.ROSInterruptException:
        pass
    print("done")
```

Results:

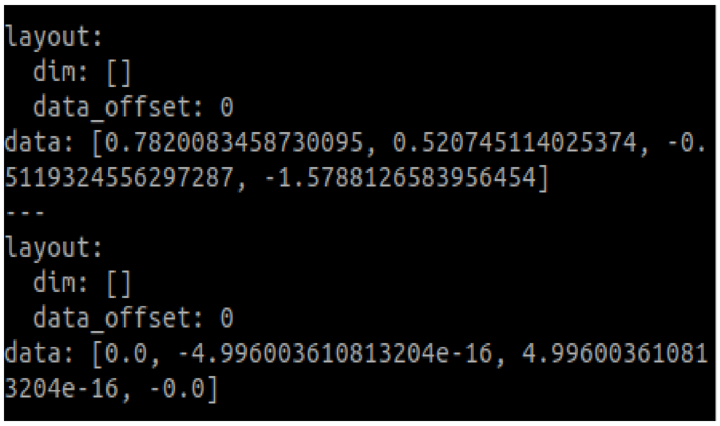
Video Link: https://youtu.be/12udx8wQQTI

1. First sending list of values as given in below table:

J1	**J2**	**J3**	**J4**	**End Effector Position**
0	0.523	0	0	[0.313,0,0.051]
0	0.523	-0.523	0	[0.347,0,0.175]
0	0.523	-0.523	-1.570	[0.221,0,0.301]
-0.693	-0.05	0.654	-0.567	[0.2,-0.158,0.129]
0.780	0.523	-0.523	-1.570	[0.160,0.147,0.301]
0	0	0	0	[0.286,0,0.205]



```
pc@pc-Vivobook-ASUSLaptop-X1502ZA-X1502ZA: ~ 49x27
layout:
  dim: []
  data_offset: 0
data: [0.0, 0.5210506575441727, 0.003780742887698
8625, -0.001831400431871577]
---
layout:
  dim: []
  data_offset: 0
data: [0.0, 0.5257978003643371, -0.52043834430150
86, -0.005359456062828616]
---
layout:
  dim: []
  data_offset: 0
data: [0.0, 0.5247862558600964, -0.51873410591780
74, -1.576052149942289]
---
layout:
  dim: []
  data_offset: 0
data: [-0.6989092634790736, -0.04772949509174934,
 0.6544505783872787, -0.5567210832955294]
```

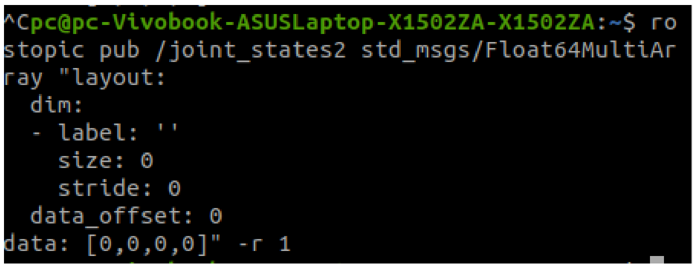


```
layout:
  dim: []
  data_offset: 0
data: [0.7820083458730095, 0.520745114025374, -0.
5119324556297287, -1.5788126583956454]
---
layout:
  dim: []
  data_offset: 0
data: [0.0, -4.996003610813204e-16, 4.99600361081
3204e-16, -0.0]
```

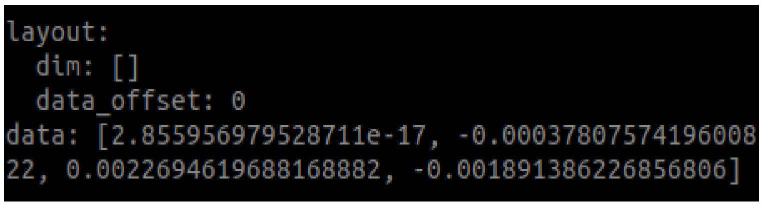
2. **Video link:** https://youtu.be/FNXsx1-2tTo
In the second case I have directly provided the values of all joint angles to the forward kinematics node and got the values of end effector position. Then these values are given to the inverse kinematics node to again get the value of joint angles. Now we can verify the values of joint angle from input and output.

Inputs

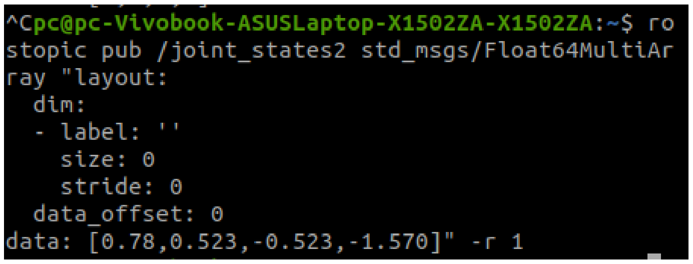
Outputs



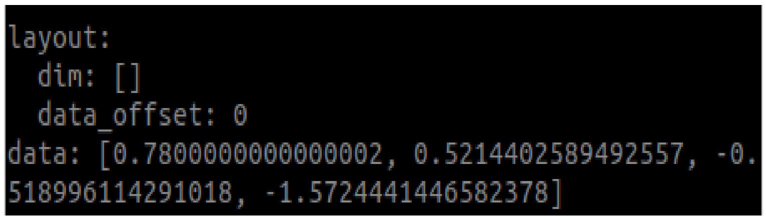
```
^Cpc@pc-Vivobook-ASUSLaptop-X1502ZA-X1502ZA:~$ ro
stopic pub /joint_states2 std_msgs/Float64MultiAr
ray "layout:
  dim:
  - label: ''
    size: 0
    stride: 0
  data_offset: 0
data: [0,0,0,0]" -r 1
```



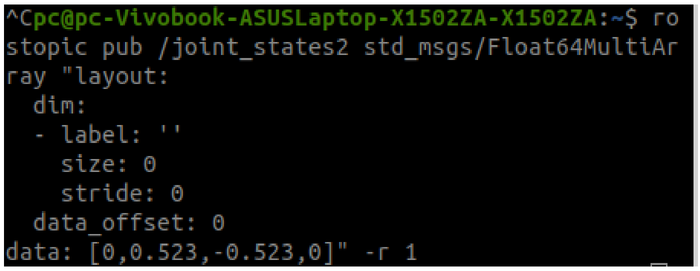
```
layout:
  dim: []
  data_offset: 0
data: [2.855956979528711e-17, -0.00037807574196008
22, 0.0022694619688168882, -0.001891386226856806]
```



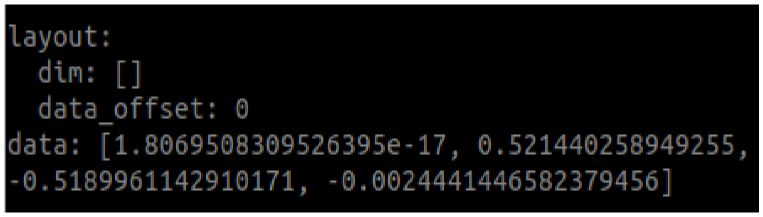
```
^Cpc@pc-Vivobook-ASUSLaptop-X1502ZA-X1502ZA:~$ ro
stopic pub /joint_states2 std_msgs/Float64MultiAr
ray "layout:
  dim:
  - label: ''
    size: 0
    stride: 0
  data_offset: 0
data: [0.78,0.523,-0.523,-1.570]" -r 1
```



```
layout:
  dim: []
  data_offset: 0
data: [0.7800000000000002, 0.5214402589492557, -0.
518996114291018, -1.5724441446582378]
```



```
^Cpc@pc-Vivobook-ASUSLaptop-X1502ZA-X1502ZA:~$ ro
stopic pub /joint_states2 std_msgs/Float64MultiAr
ray "layout:
  dim:
  - label: ''
    size: 0
    stride: 0
  data_offset: 0
data: [0,0.523,-0.523,0]" -r 1
```



```
layout:
  dim: []
  data_offset: 0
data: [1.8069508309526395e-17, 0.521440258949255,
-0.5189961142910171, -0.0024441446582379456]
```

So we can see from above that inputs and outputs are matching.

Problems Faced:

1. One of the major problems faced in this experiment is mapping our angles with gazebo angles. As there is variation in selecting the axes, it's very difficult to map them both.

2. Finding inverse kinematics is also very challenging. First I have solved it by making the equations and using fsolver for it. One problem is that it will give only one solution. And it is not working for every case as it requires some initial guess. If the initial guess is far from the solution it will not converge.

Below is the code for that method:

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

import rospy
from std_msgs.msg import Float64MultiArray
from math import atan2, sqrt, cos, sin, pi
import numpy as np
import math
from scipy.optimize import fsolve


class InverseKinematics:
   def __init__(self):
       rospy.init_node('calculate_joint_angles')
       # print("1")
       self.joint_angles_pub = rospy.Publisher('/joint_angles', Float64MultiArray,
queue_size=10)
       self.end_effector_sub = rospy.Subscriber('/end_effector_pose', Float64MultiArray,
self.end_effector_callback)




   def end_effector_callback(self, pose_msg):
       # Extract end effector position (px, py) and angle phi from the message
       # print("2")
       px, py, pz, phi= pose_msg.data
       # print(phi)
       # Calculate inverse kinematics



       # Define the equations as functions
       def equations(variables, x, y, z, phi):
           th1, th2, th3, th4 = variables
           # phi=(-th4)+(-th2)+(-th3)
           eq1 = x - (0.012+0.130 * np.cos(np.pi/2 - 0.1853 + (-th2)) + 0.124 *
np.cos((-th3) + (-th2)) + 0.126 * np.cos(phi)) * np.cos((th1))
           eq2 = y - np.tan((th1)) * x
           eq3 = z - (0.077 + 0.130 * np.sin(np.pi/2 - 0.1853 + (-th2)) + 0.124 *
np.sin((-th3) + (-th2)) + 0.126 * np.sin(phi))
           eq4 = phi - ((-th2) + (-th3)) - (-th4)
           return [eq1, eq2, eq3, eq4]

       # Initial guess for th1, th2, th3, and th4
       initial_guess = [0.0, 0.0, 0.0, 0.0]

       # Define bounds for the variables (optional but can help in convergence)
       bounds = ([-np.pi, -np.pi, -np.pi, -np.pi], [np.pi, np.pi, np.pi, np.pi])

       # Solve the equations as a least squares optimization problem
       result = fsolve(equations, initial_guess, args=(px, py, pz, phi))

       # Extract the solutions
       th1_solution, th2_solution, th3_solution, th4_solution = result

       # Create a Float64MultiArray message to publish the joint angles
       joint_angles_msg = Float64MultiArray(data=[th1_solution, th2_solution,
th3_solution, th4_solution])
       # joint_angles_msg = Float64MultiArray(data=[(theta11),(theta21),(theta31)])

       # Publish the joint angles
       self.joint_angles_pub.publish(joint_angles_msg)

if __name__ == '__main__':
   try:
       node = InverseKinematics()
       rospy.spin()
   except rospy.ROSInterruptException:
       pass
```

Service To Publish Values

Objective 1: Control joints of the manipulator (hardware) by publishing the joint positions and matching the end-effector pose with the custom forward kinematics node in Rviz.

Introduction: In this part we have to publish the value of joint angles to hardware. In turtlebot3 hardware the joint positions are published on a service named /goal_joint_space_path. So we will be calling this service with the help of a client node to publish values of joint angles to it.

Code:

1.To publish values to the service in gazebo.

```
#!/usr/bin/env python
import rospy
from std_msgs.msg import Float64MultiArray
from open_manipulator_msgs.srv import SetJointPosition, SetJointPositionRequest

class JointPositionClient:
    def __init__(self):
        rospy.init_node('joint_position_client')

        # Subscribe to the /joint_angles topic
        rospy.Subscriber('/joint_angles', Float64MultiArray, self.joint_angles_callback)

        # # Wait for the service to become available
        # rospy.wait_for_service('/goal_joint_space_path')

        # Create a service proxy
        self.set_joint_position = rospy.ServiceProxy('/goal_joint_space_path', SetJointPosition)

        # Initialize joint angles as None
        self.joint_angles = None

    def joint_angles_callback(self, msg):
        self.joint_angles = list(msg.data)

    def send_joint_positions(self, max_accelerations_scaling, max_velocity_scaling, path_time):
        if self.joint_angles is not None:
            try:
                # Create a request message
                print("Got Angles")
                self.joint_angles.append(0)
                print(self.joint_angles)
                request = SetJointPositionRequest()
                request.joint_position.joint_name = ['joint1', 'joint2', 'joint3', 'joint4', 'joint5']
                request.joint_position.position = self.joint_angles
                request.joint_position.max_accelerations_scaling_factor =
max_accelerations_scaling
                request.joint_position.max_velocity_scaling_factor = max_velocity_scaling
                request.path_time = path_time

                # Call the service
                response = self.set_joint_position(request)

                if response.is_planned:
                    print("Successfully sent joint positions.")
                    rospy.sleep(2)

                # Reset joint_angles to None after sending
                self.joint_angles = None

            except rospy.ServiceException as e:
                rospy.logerr("Service call failed: %s" % e)
        # else:
        #     rospy.logwarn("No joint angles received from the /joint_angles topic.")

if __name__ == '__main__':
    try:

        joint_position_client = JointPositionClient()

        # Define the parameters
        max_accelerations_scaling = 0.0
        max_velocity_scaling = 0.0
        path_time = 2.0


        rate = rospy.Rate(10)  # Adjust the rate as needed
        while not rospy.is_shutdown():
            joint_position_client.send_joint_positions(max_accelerations_scaling,
max_velocity_scaling, path_time)
            rate.sleep()

    except KeyboardInterrupt:
        rospy.loginfo("Ctrl+C pressed. Shutting down the node.")
```

Jacobian Based Inverse Kinematics Controller

Jacobian is a matrix that maps joint velocities to linear and angular velocities of end effector.

Inverse jacobian matrix maps end effector velocities with joint velocities.

Xdot=J*qdot

Where qdot=joint velocity matrix of size nx1 (n=number of joints)

Xdot=End effector velocity matrix of size mx1 (m=6x1 for spatial robot)

J=Jacobian matrix of size mxn

Jacobian Matrix:

1.Each column of jacobian matrix represents the effect on end effector velocity due to variation in each joint velocity. Therefore number of columns will be equal to number of joints.

2. In each row of jacobian matrix, first three entries represent linear velocities of end effector due to change in velocities of all joint angles and last three entries represent angular velocities of end effector due to change in velocities of all joint angles.

3. Upper part of jacobian is called linear velocity jacobian while lower part is called angular velocity jacobian.

4. To get Jv we need to differentiate the position function of x,y,z of the end effector wrt joint variables. The last column of jacobian matrix will give functions for x,y,z.

$$\boxed{J_v = \begin{bmatrix} \frac{\partial x}{\partial q_1} & \frac{\partial x}{\partial q_2} & \frac{\partial x}{\partial q_3} & \cdot\cdot & \cdot\cdot & \cdot\cdot & \frac{\partial x}{\partial q_n} \\ \frac{\partial y}{\partial q_1} & \frac{\partial y}{\partial q_2} & \frac{\partial y}{\partial q_3} & \cdot\cdot & \cdot\cdot & \cdot\cdot & \frac{\partial y}{\partial q_3} \\ \frac{\partial z}{\partial q_1} & \frac{\partial z}{\partial q_2} & \frac{\partial z}{\partial q_3} & \cdot\cdot & \cdot\cdot & \cdot\cdot & \frac{\partial z}{\partial q_n} \end{bmatrix}_{3\times n}}$$

5. To get Jw: Actually we already have joint velocity wrt each of the local frame of the joints. But in the jacobian matrix we have to put this joint velocities wrt base frame. We are already having the rotation information in transformation matrix. We just need to find out the right column and then we can directly multiply it with local omega to get Jw.

$$\boxed{\begin{array}{ccc} \hat{\omega}_i^b = R_{bi} * \hat{\omega}_i^i & \hat{\omega}_j^b = R_{bj} * \hat{\omega}_j^j & \hat{\omega}_k^b = R_{bk} * \hat{\omega}_k^k \\ \text{Axis of joint i} & \text{Axis of joint j} & \text{Axis of joint k} \\ \text{w.r.t frame \{b\}} & \text{w.r.t frame \{b\}} & \text{w.r.t frame \{b\}} \end{array}}$$

$$T_{bi} = \begin{array}{cccc} & & \hat{\omega}_i^b & \\ & & \uparrow & \\ \end{array} \begin{bmatrix} X & X & \boxed{a_1} & X \\ X & X & \boxed{a_2} & X \\ X & X & \boxed{a_3} & X \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad T_{bj} = \begin{array}{c} \hat{\omega}_j^b \\ \uparrow \end{array} \begin{bmatrix} \boxed{a_1} & X & X & X \\ \boxed{a_2} & X & X & X \\ \boxed{a_3} & X & X & X \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad T_{bk} = \begin{array}{c} \hat{\omega}_k^b \\ \uparrow \end{array} \begin{bmatrix} X & \boxed{a_1} & X & X \\ X & \boxed{a_2} & X & X \\ X & \boxed{a_3} & X & X \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$J = \begin{bmatrix} \frac{\partial x}{\partial q_1} & \frac{\partial x}{\partial q_2} & \frac{\partial x}{\partial q_3} & \cdots & \cdots & \frac{\partial x}{\partial q_n} \\ \frac{\partial y}{\partial q_1} & \frac{\partial y}{\partial q_2} & \frac{\partial y}{\partial q_3} & \cdots & \cdots & \frac{\partial y}{\partial q_n} \\ \frac{\partial z}{\partial q_1} & \frac{\partial z}{\partial q_2} & \frac{\partial z}{\partial q_3} & \cdots & \cdots & \frac{\partial z}{\partial q_n} \\ \hat{\omega}_1^b & \hat{\omega}_2^b & \hat{\omega}_3^b & \cdots & \cdots & \hat{\omega}_n^b \end{bmatrix}_{6\times n}$$

Pseudo Inverse Calculation: https://www.youtube.com/watch?v=vXk-o3PVUdU

Inverse Kinematics using Jacobians:

When joint actuators accept position commands we have to find qnew and give them to joint actuators. So velocity method cannot be used here. But same jacobian equation holds true for displacement domain, but it holds good only for small displacements.

$$\delta q = J^{-1}\delta X$$

So we can find small displacements, multiply it with the Jacobian inverse and get the small joint angle increment required.

We can do it until the displacement between current and goal angle becomes 0.

Problems:

1. As we are dealing with the inverse of jacobians here, so if the matrix is not square then calculating inverse is not possible.

2. When the robot is at singularity then we are not able to find the inverse of jacobian. Means matrix loses its rank as determinant becomes 0. Losing rank means losing a degree of freedom. This generally happens at the end of workspace.

The bigger problem is not being at singular configuration but being near to singular configuration as the robot starts behaving abnormally when we are close to singular configuration. This is because when we are approaching singular configuration, jacobian inverse method will produce very high joint velocities which is not acceptable.

Doing some finite movement in task space by end effector will result in infinite(very large) movement in joint space. So we have to control it to not go near to singular configuration.

Solution:

We will calculate the pseudo inverse of the jacobian matrix. This method uses Singular value decomposition to find the inverse of a non square matrix. Additionally, The joint velocities or the delta q computed using pseudo inverse won't allow any additional movement towards the singularity but will allow any movement that doesn't get us any closer to the singularity.

A) Derivation of Jacobian

Derivation:

```
# Calculating Transformation Matrices
    T10 = Matrix([[cos(th1), -sin(th1), 0, 0.012],
              [sin(th1), cos(th1), 0, 0],
              [0, 0, 1, 0.077],
              [0, 0, 0, 1]])

    T21 = Matrix([[sin(th0 - th2), -cos(th0 - th2), 0, 0],
              [0, 0, -1, 0],
              [cos(th0 - th2), sin(th0 - th2), 0, 0],
              [0, 0, 0, 1]])

    T32 = Matrix([[sin(th0+th3), cos(th0+th3), 0, 0.130],
              [-cos(th0+th3), sin(th0+th3), 0, 0],
              [0, 0, 1, 0],
              [0, 0, 0, 1]])

    T43 = Matrix([[cos(th4), -sin(th4), 0, 0.124],
              [sin(th4), cos(th4), 0, 0],
              [0, 0, 1, 0],
              [0, 0, 0, 1]])

    TE4 = Matrix([[1, 0, 0, 0.126],
              [0, 1, 0, 0],
              [0, 0, 1, 0],
              [0, 0, 0, 1]])

    # Calculating Transformations wrt global frame
    T20 = T10 * T21

    T30 = T10 * T21 * T32

    T40 = T10 * T21 * T32 * T43
```

```
TE0 = T10 * T21 * T32 * T43 * TE4

# Positions
x=TE0[0,3]
y=TE0[1,3]
z=TE0[2,3]

# Finding Partial Derivates
dx_by_dth1=diff(x, th1)
dx_by_dth2=diff(x, th2)
dx_by_dth3=diff(x, th3)
dx_by_dth4=diff(x, th4)

dy_by_dth1=diff(y, th1)
dy_by_dth2=diff(y, th2)
dy_by_dth3=diff(y, th3)
dy_by_dth4=diff(y, th4)

dz_by_dth1=diff(z, th1)
dz_by_dth2=diff(z, th2)
dz_by_dth3=diff(z, th3)
dz_by_dth4=diff(z, th4)

R10=T10[0:3,2]
R20=T20[0:3,2]
R30=T30[0:3,2]
R40=T40[0:3,2]

# Defining Jacobians
J = Matrix([[dx_by_dth1,dx_by_dth2,dx_by_dth3,dx_by_dth4],
   [dy_by_dth1,dy_by_dth2,dy_by_dth3,dy_by_dth4],
   [dz_by_dth1,dz_by_dth2,dz_by_dth3,dz_by_dth4],
   [R10[0,0],R20[0,0],R30[0,0],R40[0,0]],
   [R10[1,0],R20[1,0],R30[1,0],R40[1,0]],
   [R10[2,0],R20[2,0],R30[2,0],R40[2,0]]
   ])
```

Singular Configuration is the values of joint angles where some row or colum of jacobian matrix will be zero completely such that we loose one degree of freedom.

Singular Configuration(Degrees)	Jacobian matrix
0,0,0,0	[[0, -0.127774532917053, 0, 0], [0.273951382795758, 0, 0, 0], [0, 0.273951382795758, 0.250000000000000,

	0.126000000000000], [0, 0, 0, 0], [0, -1.00000000000000, -1.00000000000000, -1.00000000000000], [1.00000000000000, 0, 0, 0]]
0,0,90,0	[[0, -0.240418112530723, -0.112643579613670, -0.112643579613670], [0.0914941071634823, 0, 0, 0], [0, 0.0914941071634823, 0.0675427243677246, -0.0564572756322754], [0, 0, 0, 0], [0, -1.00000000000000, -1.00000000000000, -1.00000000000000], [1.00000000000000, 0, 0, 0]]
0,0,-54,0	[[0, -0.0573671127617494, 0.0704074201553036, 0.0704074201553036], [0.0434583438550008, 0, 0, 0], [0, 0.0434583438550008, 0.0195069610592431, -0.104493038940757], [0, 0, 0, 0], [0, -1.00000000000000, -1.00000000000000, -1.00000000000000], [1.00000000000000, 0, 0, 0]]
0,-90,-54,0	[[0, -0.0440905451532307, -0.122755398474617, -0.0618687208312071], [0.321284873678462, 0, 0, 0], [0, 0.321284873678462, 0.217786850258086, 0.109764572530075], [0, 0, 0, 0], [0, -1.00000000000000, -1.00000000000000, -1.00000000000000], [1.00000000000000, 0, 0, 0]]
-90,-90,-54,90	[[0.222958263642905, 0.0458803449376448, 0.0811279902276293, 0.0538462764018554], [-0.111747302315430, 0.0915404831318453, 0.161866599543792, 0.107434112872861], [0, 0.249394961660085, 0.145896938239709, 0.0378746605116983], [0, -0.893996663600558, -0.893996663600558, -0.893996663600558], [0, 0.448073616129170, 0.448073616129170, 0.448073616129170], [1.00000000000000, 0, 0, 0]]

B) Custom Node For Jacobian

Code:

```
def jacobian_calculation(self):
    # Defining Symbols
    th1, th2, th3, th4, th0 = symbols('th1 th2 th3 th4 th0')
```

```
# Calculating Transformation Matrices
T10 = Matrix([[cos(th1), -sin(th1), 0, 0.012],
          [sin(th1), cos(th1), 0, 0],
          [0, 0, 1, 0.077],
          [0, 0, 0, 1]])

T21 = Matrix([[sin(th0 - th2), -cos(th0 - th2), 0, 0],
          [0, 0, -1, 0],
          [cos(th0 - th2), sin(th0 - th2), 0, 0],
          [0, 0, 0, 1]])

T32 = Matrix([[sin(th0+th3), cos(th0+th3), 0, 0.130],
          [-cos(th0+th3), sin(th0+th3), 0, 0],
          [0, 0, 1, 0],
          [0, 0, 0, 1]])

T43 = Matrix([[cos(th4), -sin(th4), 0, 0.124],
          [sin(th4), cos(th4), 0, 0],
          [0, 0, 1, 0],
          [0, 0, 0, 1]])

TE4 = Matrix([[1, 0, 0, 0.126],
          [0, 1, 0, 0],
          [0, 0, 1, 0],
          [0, 0, 0, 1]])

# Calculating Transformations wrt global frame
T20 = T10 * T21

T30 = T10 * T21 * T32

T40 = T10 * T21 * T32 * T43

TE0 = T10 * T21 * T32 * T43 * TE4

# Positions
x=TE0[0,3]
y=TE0[1,3]
z=TE0[2,3]

# Finding Partial Derivates
dx_by_dth1=diff(x, th1)
dx_by_dth2=diff(x, th2)
```

```
dx_by_dth3=diff(x, th3)
dx_by_dth4=diff(x, th4)

dy_by_dth1=diff(y, th1)
dy_by_dth2=diff(y, th2)
dy_by_dth3=diff(y, th3)
dy_by_dth4=diff(y, th4)

dz_by_dth1=diff(z, th1)
dz_by_dth2=diff(z, th2)
dz_by_dth3=diff(z, th3)
dz_by_dth4=diff(z, th4)

R10=T10[0:3,2]
R20=T20[0:3,2]
R30=T30[0:3,2]
R40=T40[0:3,2]

# Defining Jacobians
J = Matrix([[dx_by_dth1,dx_by_dth2,dx_by_dth3,dx_by_dth4],
   [dy_by_dth1,dy_by_dth2,dy_by_dth3,dy_by_dth4],
   [dz_by_dth1,dz_by_dth2,dz_by_dth3,dz_by_dth4],
   [R10[0,0],R20[0,0],R30[0,0],R40[0,0]],
   [R10[1,0],R20[1,0],R30[1,0],R40[1,0]],
   [R10[2,0],R20[2,0],R30[2,0],R40[2,0]]
   ])

return J
```

Explanation:

1.Each column of jacobian matrix represents the effect on end effector velocity due to variation in each joint velocity. Therefore number of columns will be equal to number of joints.

2. In each row of jacobian matrix, first three entries represent linear velocities of end effector due to change in velocities of all joint angles and last three entries represent angular velocities of end effector due to change in velocities of all joint angles.

3. Upper part of jacobian is called linear velocity jacobian while lower part is called angular velocity jacobian.

4. To get Jv we need to differentiate the position function of x,y,z of the end effector wrt joint variables. The last column of jacobian matrix will give functions for x,y,z.

5. To get Jw: Actually we already have joint velocity wrt each of the local frame of the joints. But in the jacobian matrix we have to put this joint velocities wrt base frame. We are already having the rotation information in transformation matrix. We just need to find out the right column and then we can directly multiply it with local omega to get Jw.

C) End effector Control of Open Manipulator X in Gazebo

Code:

```
import rospy
from sensor_msgs.msg import JointState
from gazebo_msgs.msg import LinkStates
from std_msgs.msg import Float64MultiArray
from open_manipulator_msgs.srv import SetJointPosition, SetJointPositionRequest
import numpy as np
from sympy import *
import sympy as sp
import math

class ik:
  def __init__(self):
    rospy.Subscriber("/joint_states",JointState,self.joint_angles_callback)
    rospy.Subscriber("/gazebo/link_states",LinkStates,self.end_eff_position_callback)
    self.joint_pub=rospy.Publisher("/joint_angles",Float64MultiArray,queue_size=10)
    # rospy.Subscriber("/joint_angles",Float64MultiArray,self.desired_angles_callback)
    self.send_joint_angles_to_service=rospy.ServiceProxy("/goal_joint_space_path", SetJointPosition)
    # self.joint_angles
    # self.current_end_eff_position=None
    self.desired_position=[0.134, -0.021024306644566202, 0.241]

  def desired_angles_callback(self,msg):
    self.joint_angles=list(msg.data)

  def joint_angles_callback(self,msg):
    self.current_angles=[msg.position[2],msg.position[3],msg.position[4],msg.position[5]]
    # print("Current Angles: ",self.current_angles)

  def end_eff_position_callback(self,msg):
    self.current_end_eff_position=[msg.pose[7].position.x+0.045,msg.pose[7].position.y,msg.pose[7].position.z]
    # print("Current End Effector Position: ",self.current_end_eff_position)

  def send_joint_angles(self,des_q):
    # print("Service")
    request = SetJointPositionRequest()
    request.joint_position.joint_name = ['joint1', 'joint2', 'joint3', 'joint4']
    request.path_time=0.01
```

```
        # if self.joint_angles is not None:
        # self.joint_angles.append(0)
        # self.joint_angles = np.append(self.joint_angles, 0)
        request.joint_position.position = (des_q)
        # print("Joint Angles: ",des_q)
        response=self.send_joint_angles_to_service(request)
        if response.is_planned:
                print("Successfully sent joint positions.")
                rospy.sleep(0.1)
           # self.joint_angles=None

    def main_ik(self):
        msg=Float64MultiArray()
        th1, th2, th3, th4, th0 = symbols('th1 th2 th3 th4 th0') #Defined symbols to use later in calculation
        theta = sp.symbols('theta[0:%d]' % 4)

        self.start_position=self.current_end_eff_position
        # exit()
        positions=np.linspace(self.start_position,self.desired_position,200)
        positions=[positions[0],positions[-1]]
        for desired_position in positions:
            error=desired_position-self.current_end_eff_position

            while(np.linalg.norm(error)>0.03):
                print("Norm To Exit: ",np.linalg.norm(np.array(self.desired_position)-self.current_end_eff_position))
                if(np.linalg.norm(np.array(self.desired_position)-self.current_end_eff_position) < 0.02):
                    rospy.loginfo("Reached")
                    exit()
                print("Desired Position: ",self.desired_position)
                print("Current Position: ",self.current_end_eff_position)
                print("Error: ",error)
                print("Error Norm: ",np.linalg.norm(error))
                # print("\n")
                # Function call for jacobian calculation
                # J=self.jacobian_calculation()
                J=self.forward_kinematics_inv_Jacobian()



                # print("Jacobian Shape1 :",J.shape)
                # print("Jacobian Shape2 :",J_for.shape)
                # rospy.sleep(30)
                # print("Current Angles: ",self.current_angles)
                # Set joint angles to specific values
                # th1_val, th2_val, th3_val, th4_val, th0_val = self.current_angles[0], self.current_angles[1],
self.current_angles[2], self.current_angles[3], 0.1853
                # Substitute numerical values for joint angles
                # J_with_values = J.subs({th0: th0_val, th1: th1_val, th2: th2_val, th3: th3_val, th4: th4_val})


                q=[0.1853,self.current_angles[0], self.current_angles[1], self.current_angles[2], self.current_angles[3]]

                theta_values = [q[0], q[1] + math.pi/2 - 0.1853, q[2] - math.pi/2 + 0.1853, q[3]]
                J_with_values=J.subs({theta[0]: theta_values[0], theta[1]: theta_values[1],
                        theta[2]: theta_values[2], theta[3]: theta_values[3]})
        # Calculate the result of the Jacobian matrix with numerical values
        result_J = J_with_values.evalf()
        # print(result_J)
        # rospy.sleep(100)
        # print("Type of matrix: ",result_J.type)
        # Convert to a numeric NumPy array
        result_J_np = np.array(result_J).astype(float)

        pseudo_J=np.linalg.pinv(result_J_np)
        # Convert the numpy array to a list

        # Calculate joint positions using the pseudo Jacobian and inverse kinematics(delta q calculation)
        # delta_theta = np.dot(pseudo_J[:,0:3], error)
        delta_theta = np.dot(pseudo_J, error)


        # Provide some sleep to update current angles and position

        print("Delta Theta:",delta_theta)
        des_q = self.current_angles - 0.05*delta_theta
        print("Current Angles: ",self.current_angles)
        print("Desired q: ",des_q)
        print("\n")
        # des_q[0]=-des_q[0]
        # des_q=-des_q
        self.send_joint_angles(des_q)
        # print("x")
        # print("Q:",q)
        # # self.joint_angles=q
        # msg.data = q  #Value updated
        # self.joint_pub.publish(msg) #Value published to topic
        # rospy.sleep(0.1) #wait for msg to publish
        rospy.sleep(0.1)
        error=desired_position-self.current_end_eff_position




@staticmethod
def jacobian_calculation():
    # Defining Symbols
    th1, th2, th3, th4, th0 = symbols('th1 th2 th3 th4 th0')

    # Calculating Transformation Matrices
    T10 = Matrix([[cos(th1), -sin(th1), 0, 0.012],
            [sin(th1), cos(th1), 0, 0],
            [0, 0, 1, 0.077],
            [0, 0, 0, 1]])

    T21 = Matrix([[sin(th0 - th2), -cos(th0 - th2), 0, 0],
            [0, 0, -1, 0],
            [cos(th0 - th2), sin(th0 - th2), 0, 0],
            [0, 0, 0, 1]])

T32 = Matrix([[sin(th0+th3), cos(th0+th3), 0, 0.130],
            [-cos(th0+th3), sin(th0+th3), 0, 0],
            [0, 0, 1, 0],
            [0, 0, 0, 1]])

T43 = Matrix([[cos(th4), -sin(th4), 0, 0.124],
            [sin(th4), cos(th4), 0, 0],
            [0, 0, 1, 0],
            [0, 0, 0, 1]])

TE4 = Matrix([[1, 0, 0, 0.126],
            [0, 1, 0, 0],
            [0, 0, 1, 0],
            [0, 0, 0, 1]])

# Calculating Transformations wrt global frame
T20 = T10 * T21

T30 = T10 * T21 * T32

T40 = T10 * T21 * T32 * T43

TE0 = T10 * T21 * T32 * T43 * TE4

# Positions
x=TE0[0,3]
y=TE0[1,3]
z=TE0[2,3]

# Finding Partial Derivates
dx_by_dth1=diff(x, th1)
dx_by_dth2=diff(x, th2)
dx_by_dth3=diff(x, th3)
dx_by_dth4=diff(x, th4)

dy_by_dth1=diff(y, th1)
dy_by_dth2=diff(y, th2)
dy_by_dth3=diff(y, th3)
dy_by_dth4=diff(y, th4)

dz_by_dth1=diff(z, th1)
dz_by_dth2=diff(z, th2)
dz_by_dth3=diff(z, th3)
dz_by_dth4=diff(z, th4)

R10=T10[0:3,2]
R20=T20[0:3,2]
R30=T30[0:3,2]
R40=T40[0:3,2]

# Defining Jacobians
J = Matrix([[dx_by_dth1,dx_by_dth2,dx_by_dth3,dx_by_dth4],
    [dy_by_dth1,dy_by_dth2,dy_by_dth3,dy_by_dth4],
            [dz_by_dth1,dz_by_dth2,dz_by_dth3,dz_by_dth4],
            [R10[0,0],R20[0,0],R30[0,0],R40[0,0]],
            [R10[1,0],R20[1,0],R30[1,0],R40[1,0]],
            [R10[2,0],R20[2,0],R30[2,0],R40[2,0]]
            ])

        return J

    @staticmethod
    def forward_kinematics_inv_Jacobian():
        # Function for forward kinematics and inverse Jacobian calculation

        epsilon = sp.Matrix([1, 1, 1, 1])
        n = 4

        # Define symbolic variables
        theta = sp.symbols('theta[0:%d]' % n)
        alpha = np.array([0, np.pi/2, 0, 0])
        a = np.array([0.012, 0, 0.130, 0.125])
        d = np.array([0.077, 0, 0, 0])

        # Initialize the transformation matrices
        Ttemp = sp.eye(4)
        HTM = [sp.eye(4) for _ in range(n)]

        for i in range(n):
            # Calculate the transformation matrix elements using symbolic variables
            t11 = sp.cos(theta[i])
            t12 = -sp.sin(theta[i])
            t13 = 0
            t14 = a[i]

            t21 = sp.sin(theta[i]) * sp.cos(alpha[i])
            t22 = sp.cos(theta[i]) * sp.cos(alpha[i])
            t23 = -sp.sin(alpha[i])
            t24 = -d[i] * sp.sin(alpha[i])

            t31 = sp.sin(theta[i]) * sp.sin(alpha[i])
            t32 = sp.cos(theta[i]) * sp.sin(alpha[i])
            t33 = sp.cos(alpha[i])
            t34 = d[i] * sp.cos(alpha[i])

            Tiim1 = sp.Matrix([[t11, t12, t13, t14],
                        [t21, t22, t23, t24],
                        [t31, t32, t33, t34],
                        [0, 0, 0, 1]])

            Ti0 = Ttemp * Tiim1
            HTM[i] = Ti0
            Ttemp = Ti0

        # Homogeneous Transformation matrix
        Tn0 = HTM[n - 1]
        HTM_list = [HTM[0], HTM[1], HTM[2], HTM[3]]

        # End effector's position
        pE0_h = Tn0 * sp.Matrix([0.126, 0, 0, 1])
        pE0 = sp.Matrix(pE0_h[:3])

        # Initialize the Jacobian matrix
        Jv = sp.Matrix.ones(3, n)

        for i, item in enumerate(HTM_list):
            v1 = epsilon[i] * sp.Matrix(item[:3, 2])
            v2 = pE0 - sp.Matrix(item[:3, 3])
            cross_product = v1.cross(v2)
            Jv[:, i] = cross_product

        return Jv

        # # Substitute joint values into the Jacobian matrix
        # theta_values = [q[0], q[1] + math.pi/2 - 0.1853, q[2] - math.pi/2 + 0.1853, q[3]]
        # Jv_substituted = Jv.subs({theta[0]: theta_values[0], theta[1]: theta_values[1],
        #                           theta[2]: theta_values[2], theta[3]: theta_values[3]})
        # Jv_substituted = N(Jv_substituted, 3)
        # Jv_substituted = Jv_substituted.applyfunc(lambda x: round(x, 3))

        # # Substitute joint values into end effector position
        # ee_pos = pE0.subs({theta[0]: theta_values[0], theta[1]: theta_values[1],
        #                    theta[2]: theta_values[2], theta[3]: theta_values[3]})
        # Jv = np.array(Jv_substituted, dtype=float)
        # ee_pos = np.array(ee_pos, dtype=float).reshape(3)

        # return np.linalg.pinv(Jv)

if __name__=="__main__":
    try:
        rospy.init_node("IK_using_jacobians_method")
        obj=ik()
        rospy.sleep(0.1) #Give some sleep for program to read value from topics

        # while not rospy.is_shutdown():
        obj.main_ik()
            # obj.send_joint_angles()
            # rospy.Rate(10)
    except KeyboardInterrupt:
        rospy.loginfo("Keyboard Interrupted!!")
```

Explanation:

1. We are continuously reading angles from topic /joint_states.
2. We will provide it with a desired angle.
3. Then we based on the error norm of both the values, we will design the control law.
4. Mean while the jacobian calculation function will calculate the jacobians and return it to main function.
5. Below part of the code is used to send the end effector to desired position using a service:

```
def send_joint_angles(self,des_q):
       # print("Service")
       request = SetJointPositionRequest()
       request.joint_position.joint_name = ['joint1', 'joint2', 'joint3', 'joint4']
       request.path_time=0.01
       # if self.joint_angles is not None:
       # self.joint_angles.append(0)
       # self.joint_angles = np.append(self.joint_angles, 0)
       request.joint_position.position = (des_q)
       # print("Joint Angles: ",des_q)
       response=self.send_joint_angles_to_service(request)
       if response.is_planned:
                   print("Successfully sent joint positions.")
                   rospy.sleep(0.1)
           # self.joint_angles=None
```

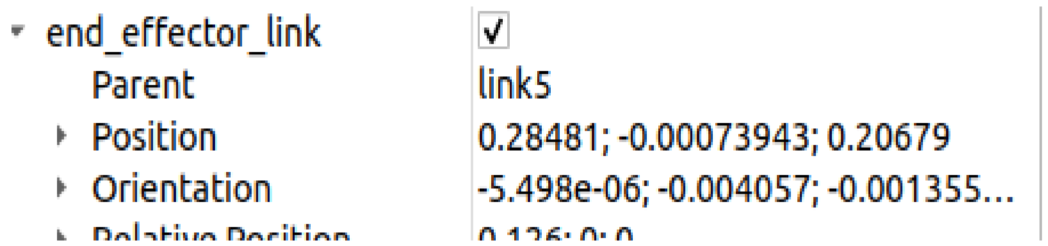
6. Below part is used to read current joint angles and end effector position from desired topic:

```
   def desired_angles_callback(self,msg):
       self.joint_angles=list(msg.data)
```

```

   def end_eff_position_callback(self,msg):

self.current_end_eff_position=[msg.pose[7].position.x+0.045,msg.pose[7].position.y,m
sg.pose[7].position.z]
       # print("Current End Effector Position: ",self.current_end_eff_position)
```

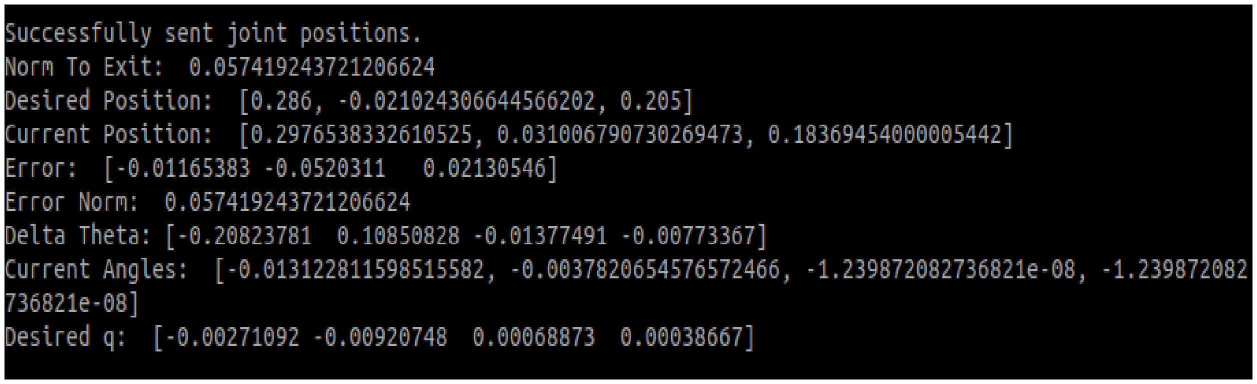
Results:

1. Reached Position



```
end_effector_link      ✓
    Parent             link5
    Position           0.28481; -0.00073943; 0.20679
    Orientation        -5.498e-06; -0.004057; -0.001355...
```

2. Desired Position



```
Successfully sent joint positions.
Norm To Exit:  0.057419243721206624
Desired Position:  [0.286, -0.021024306644566202, 0.205]
Current Position:  [0.2976538332610525, 0.031006790730269473, 0.18369454000005442]
Error:  [-0.01165383 -0.0520311   0.02130546]
Error Norm:  0.057419243721206624
Delta Theta: [-0.20823781  0.10850828 -0.01377491 -0.00773367]
Current Angles:  [-0.013122811598515582, -0.0037820654576572466, -1.239872082736821e-08, -1.239872082
736821e-08]
Desired q:  [-0.00271092 -0.00920748  0.00068873  0.00038667]
```

Video: https://youtu.be/CL5X377r1SA

D) End effector Control of Open Manipulator X

The code and explanation will be same as above one in the simulation. The video link is attached below:

Video: https://youtu.be/p_y5e9w9X9o

Trajectory Planning of Open Manipulator

Objective : Create a node for Trajectory generation, which takes inputs such as time, total time and initial and end effector's position and gives outputs as the joint angles.

Introduction: In this we have to make a node that makes the end effector to follow cycloidal trajectory. So we are dividing the difference between two positions in a certain number of steps and then achieving that small value by following a cycloidal trajectory.

Code:

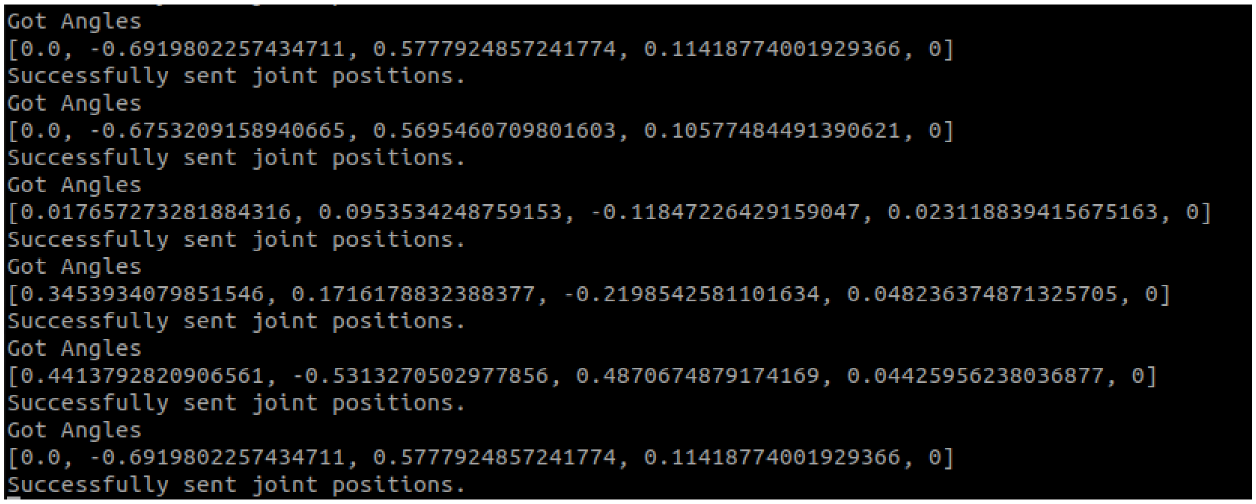
```
#!/usr/bin/env python
import rospy
from std_msgs.msg import Float64MultiArray
import numpy as np
import matplotlib.pyplot as plt

class InverseKinematics:
    def __init__(self):
        rospy.init_node('calculate_joint_angless')
        self.joint_angles_pub = rospy.Publisher('/joint_angles', Float64MultiArray, queue_size=10)
        self.end_effector_positions = [
            [0.186,0,0.205,0],
            [0.286,0,0.205,0],
            [0.286,0.1,0.205,0],
            [0.186,0.1,0.205,0],
            [0.186,0,0.205,0],
            [0.186,0,0.205,0]
        ]
        self.current_position_index = 0

    def P3R_inverse_kinematics(self, a1, a2, a3, x3, y3, theta):
        # Inverse Kinematics of planar 3R robot
        x2 = x3 - a3 * np.cos(theta)
        y2 = y3 - a3 * np.sin(theta)

        cos_th2 = (x2**2 + y2**2 - a1**2 - a2**2) / (2 * a1 * a2)
        sin_th2 = [-np.sqrt(1 - cos_th2**2), np.sqrt(1 - cos_th2**2)]

        th2 = [np.arctan2(sin_th2[0], cos_th2), np.arctan2(sin_th2[1], cos_th2)]

        sin_th1 = [(y2 * (a1 + a2 * np.cos(th2[0])) - a2 * np.sin(th2[0]) * x2) / (a1**2 + a2**2 + 2 * a1
* a2 * np.cos(th2[0])),
                (y2 * (a1 + a2 * np.cos(th2[1])) - a2 * np.sin(th2[1]) * x2) / (a1**2 + a2**2 + 2 * a1 * a2
* np.cos(th2[1]))]

        cos_th1 = [(x2 * (a1 + a2 * np.cos(th2[0])) + a2 * np.sin(th2[0]) * y2) / (a1**2 + a2**2 + 2 * a1
* a2 * np.cos(th2[0])),
                (x2 * (a1 + a2 * np.cos(th2[1])) + a2 * np.sin(th2[1]) * y2) / (a1**2 + a2**2 + 2 * a1 * a2
* np.cos(th2[1]))]

        th1 = [np.arctan2(sin_th1[0], cos_th1[0]), np.arctan2(sin_th1[1], cos_th1[1])]

        th3 = [theta - th1[0] - th2[0], theta - th1[1] - th2[1]]

        return th1, th2, th3

    def omx_inverse_kinematics(self, target_pos, target_rot=None, target_phi=None):
        x = target_pos[0]
        y = target_pos[1]
        z = target_pos[2]
        print(x,y,z)
        d1 = 0.077
        a1 = np.sqrt(0.024**2 + 0.128**2)
        alpha_2 = np.arctan(0.024/0.128)
        a2 = 0.124
        a3 = 0.126

        x_new = np.sqrt(x**2 + y**2)
        y_new = z - d1
        if target_phi is None:
            phi = self.calculate_angle_with_xy_plane(target_rot)
        else:
            phi = -target_phi

        sm_th = (phi + np.pi) % (2 * np.pi) - np.pi
        if -np.pi/2 <= sm_th and sm_th <= np.pi/2:
            x_ph = [[x_new, phi],
                    [-x_new, np.pi - phi]]
        else:
            x_ph = [[x_new, np.pi - phi],
                    [-x_new, phi]]

        theta_1 = [np.arctan2(y, x), np.arctan2(-y, -x)]
        thetas = []
        for i in range(2):
            th2, th3, th4 = self.P3R_inverse_kinematics(a1=a1,
                                                a2=a2,
                                                a3=a3,
                                                x3=x_ph[i][0],
                                                y3=y_new,
                                                theta=x_ph[i][1])

            theta_2 = [np.pi/2 - th2[0] - alpha_2, np.pi/2 - th2[1] - alpha_2]
            theta_3 = [-np.pi/2 - th3[0] + alpha_2, -np.pi/2 - th3[1] + alpha_2]
            theta_4 = [-th4[0], -th4[1]]

            thetas.append([theta_1[i], theta_2[0], theta_3[0], theta_4[0]])
            thetas.append([theta_1[i], theta_2[1], theta_3[1], theta_4[1]])

        return thetas

    def calculate_joint_angles(self, start_pos, end_pos, duration):
        num_steps = 100  # Number of steps for the cycloidal profile
        joint_angles_trajectory = []

        for t in np.linspace(0, 1, num_steps):
            intermediate_pos = [
                start_pos[i] + (end_pos[i] - start_pos[i]) * t for i in range(3)
            ]
            target_phi = end_pos[3]

            # Calculate the desired joint angles for the intermediate position using the cycloid equation
            t_j = t - np.sin(2 * np.pi * t) / (2 * np.pi)
            joint_angles = np.array(self.omx_inverse_kinematics(intermediate_pos,
target_phi=target_phi))

            # Append the joint angles to the trajectory
            joint_angles_trajectory.append(joint_angles[0])

        return joint_angles_trajectory

    def plot_joint_angles_trajectory(self, start_pos, end_pos, duration):
        num_steps = 100  # Number of steps for the cycloidal profile
        time_points = np.linspace(0, 1, num_steps)
        joint_angles_trajectory = self.calculate_joint_angles(start_pos, end_pos, duration)

        plt.figure(figsize=(10, 6))
        plt.plot(time_points, joint_angles_trajectory)
        plt.title('Joint Angles vs. Time')
        plt.xlabel('Time')
        plt.ylabel('Joint Angles')
        plt.grid(True)
        # plt.show()

    def run(self):
        rate = rospy.Rate(10)

        while not rospy.is_shutdown():
            if self.current_position_index < len(self.end_effector_positions) - 1:
                start_pos = self.end_effector_positions[self.current_position_index]
                end_pos = self.end_effector_positions[self.current_position_index + 1]
                duration = 2.0
                self.plot_joint_angles_trajectory(start_pos, end_pos, duration)
                joint_angles_trajectory = self.calculate_joint_angles(start_pos, end_pos, duration)

                for joint_angles in joint_angles_trajectory:
                    joint_angles_msg = Float64MultiArray(data=joint_angles)
                    self.joint_angles_pub.publish(joint_angles_msg)
                    rospy.sleep(duration / len(joint_angles_trajectory))
                self.plot_joint_angles_trajectory

                self.current_position_index += 1
            else:
                rospy.loginfo("Reached the last position. Shutting down the node.")
                rospy.signal_shutdown("End of positions reached")

if __name__ == '__main__':
    try:
        node = InverseKinematics()
        node.run()
    except rospy.ROSInterruptException:
        pass
```

Code Explanation:

1. The inverse kinematics node is the same, I have just added the calculate_joint_angle function to divide the position in small steps where each of the desired positions is calculated using cycloidal trajectory.
2. A loop iterates over t for values between 0 and 1, dividing this range into num_steps equal intervals. This loop is used to generate intermediate positions between the start_pos and end_pos.
3. Inside the loop, intermediate_pos is calculated as a linear interpolation between start_pos and end_pos for each of the three dimensions (presumably, X, Y, and Z coordinates).
4. The code then calculates the desired joint angles for the intermediate position using the cycloid equation. The variable t_j is calculated using the formula t - np.sin(2 * np.pi * t) / (2 * np.pi). This formula is a modification of the original t value to account for the cycloidal profile.
5. The function self.omx_inverse_kinematics is called with the intermediate_pos and target_phi as arguments, presumably to calculate the joint angles required for the robot's end effector to reach the desired intermediate position. The result is assumed to be an array of joint angles, and it's converted to a NumPy array and stored in the joint_angles variable.
6. The joint_angles[0] is appended to the joint_angles_trajectory. This seems to be assuming that the result from the inverse kinematics function is a list or array of joint angles, and only the first element is added to the trajectory.
7. The loop continues, and this process is repeated for num_steps intermediate positions.

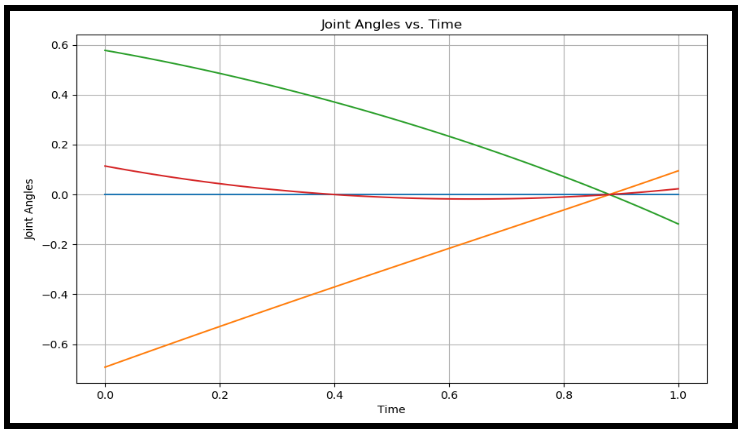
8. Finally, the joint_angles_trajectory is returned, which is a list of joint angles that should be followed by the robot's joints as it moves from start_pos to end_pos following a cycloidal profile.

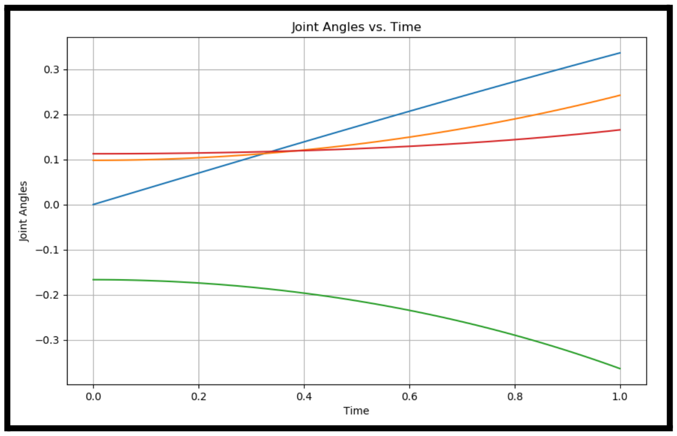
Result:

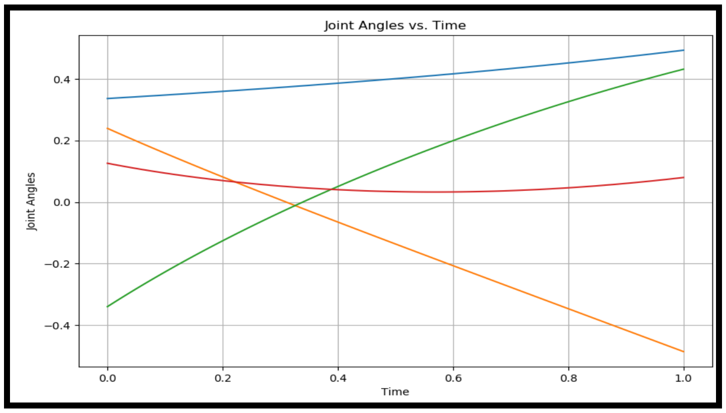


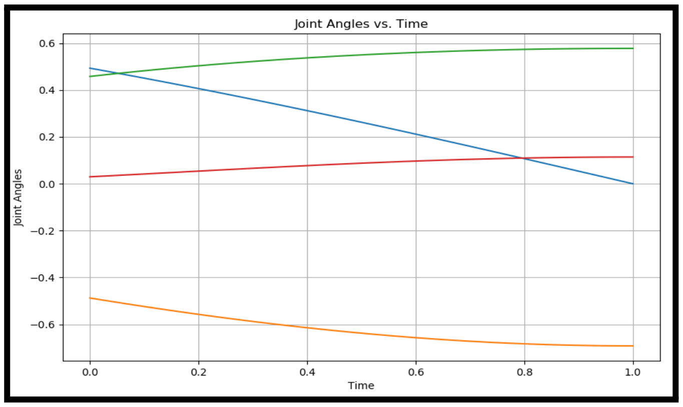
```
Got Angles
[0.0, -0.6919802257434711, 0.5777924857241774, 0.11418774001929366, 0]
Successfully sent joint positions.
Got Angles
[0.0, -0.6753209158940665, 0.5695460709801603, 0.10577484491390621, 0]
Successfully sent joint positions.
Got Angles
[0.017657273281884316, 0.0953534248759153, -0.11847226429159047, 0.023118839415675163, 0]
Successfully sent joint positions.
Got Angles
[0.3453934079851546, 0.1716178832388377, -0.2198542581101634, 0.048236374871325705, 0]
Successfully sent joint positions.
Got Angles
[0.4413792820906561, -0.5313270502977856, 0.4870674879174169, 0.04425956238036877, 0]
Successfully sent joint positions.
Got Angles
[0.0, -0.6919802257434711, 0.5777924857241774, 0.11418774001929366, 0]
Successfully sent joint positions.
```

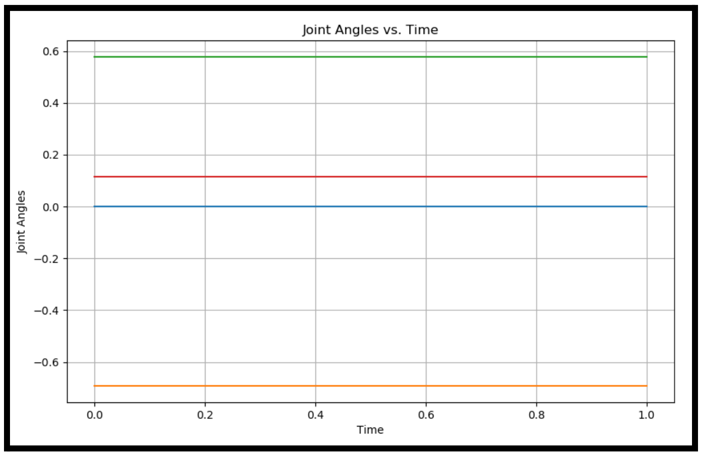
1. Cycloidal Trajectory











Video Link: **https://youtu.be/bdK_IYgnhFk**

Objective 3: Visualization in Rviz

Introduction: We have already done all the code explanations in the above parts.

Video Link: **https://youtu.be/jXpb2JAtY7I**

Objective 4: Trajectory Control on Hardware

Introduction: Explanation of code is already done.

Video Link: **https://youtu.be/-JCoyBe7ILs**

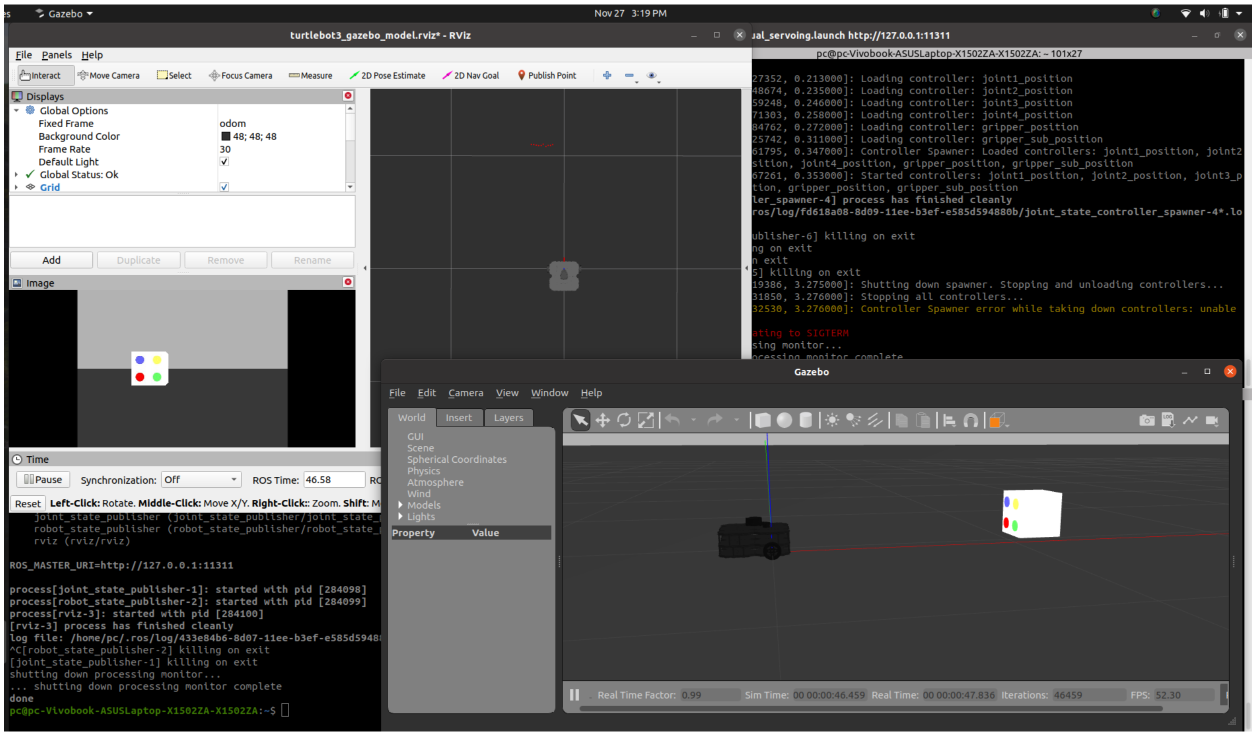
Visual Servoing

A) Simulation Setup

1. All the files are downloaded in a ros workspace.

2. The package is build using catkin_make.

3. Source the package and launch it using below command:



Global Options
Fixed Frame: odom
Background Color: 48; 48; 48
Frame Rate: 30
Default Light: ✓
Global Status: Ok

Pause | Synchronization: Off | ROS Time: 46.58

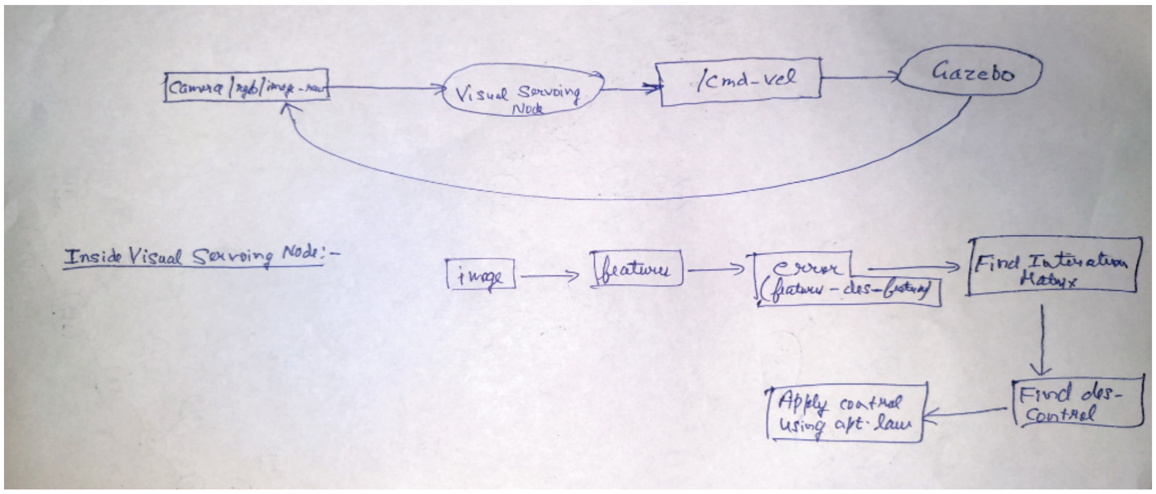
Reset **Left-Click:** Rotate. **Middle-Click:** Move X/Y. **Right-Click:**: Zoom. **Shift**: M

```
ual_servoing.launch http://127.0.0.1:11311
pc@pc-Vivobook-ASUSLaptop-X1502ZA-X1502ZA: ~ 101x27
27352, 0.213000]: Loading controller: joint1_position
48674, 0.235000]: Loading controller: joint2_position
59248, 0.246000]: Loading controller: joint3_position
71303, 0.258000]: Loading controller: joint4_position
84762, 0.272000]: Loading controller: gripper_position
25742, 0.311000]: Loading controller: gripper_sub_position
61795, 0.347000]: Controller Spawner: Loaded controllers: joint1_position, joint2
sition, joint4_position, gripper_position, gripper_sub_position
67261, 0.353000]: Started controllers: joint1_position, joint2_position, joint3_p
tion, gripper_position, gripper_sub_position
ler_spawner-4] process has finished cleanly
ros/log/fd618a08-8d09-11ee-b3ef-e585d594880b/joint_state_controller_spawner-4*.lo

ublisher-6] killing on exit
ng on exit
n exit
5] killing on exit
19386, 3.275000]: Shutting down spawner. Stopping and unloading controllers...
31850, 3.276000]: Stopping all controllers...
32530, 3.276000]: Controller Spawner error while taking down controllers: unable

ating to SIGTERM
sing monitor...
ocessing monitor complete
```

```
    joint_state_publisher (joint_state_publisher/joint_state_
    robot_state_publisher (robot_state_publisher/robot_state_
    rviz (rviz/rviz)

ROS_MASTER_URI=http://127.0.0.1:11311

process[joint_state_publisher-1]: started with pid [284098]
process[robot_state_publisher-2]: started with pid [284099]
process[rviz-3]: started with pid [284100]
[rviz-3] process has finished cleanly
log file: /home/pc/.ros/log/433e84b6-8d07-11ee-b3ef-e585d5948
^C[robot_state_publisher-2] killing on exit
[joint_state_publisher-1] killing on exit
shutting down processing monitor...
... shutting down processing monitor complete
done
pc@pc-Vivobook-ASUSLaptop-X1502ZA-X1502ZA:~$
```

Real Time Factor: 0.99 | Sim Time: 00 00:00:46.459 | Real Time: 00 00:00:47.836 | Iterations: 46459 | FPS: 52.30

B) Design Visual Servoing Architecture



C) Implement the Visual Servoing in Simulation

Code:

```
import rospy
from sensor_msgs.msg import Image
import cv2
from cv_bridge import CvBridge,CvBridgeError
import numpy as np
from geometry_msgs.msg import Twist
from nav_msgs.msg import Odometry
import matplotlib.pyplot as plt

class VS:
  def __init__(self):

    self.des_image=cv2.imread("/home/pc/turtlesim_ws/src/robot_controller/scripts/Reference_image.jpg")
    # cv2.imshow("Colored",des_image)
    # cv2.waitKey(1)
    gray_des=cv2.cvtColor(self.des_image,cv2.COLOR_BGR2GRAY)
    circles1= cv2.HoughCircles(gray_des,cv2.HOUGH_GRADIENT, dp=1, minDist=50, param1=50,
param2=30, minRadius=10, maxRadius=100)
    # print("Circles",circles1)
    # rospy.sleep(10)

    # If circles are found, draw them
    features1 = []
    if circles1 is not None:
      self.circles1 = np.uint16(np.around(circles1))

      for i in self.circles1[0, :]:
        # Extract circle features (center_x, center_y, radius)
        features1.append([i[0], i[1], i[2]])
        # Draw the outer circle
        cv2.circle(self.des_image, (i[0], i[1]), i[2], (0, 255, 0), 2)
        # Draw the center of the circle
        cv2.circle(self.des_image, (i[0], i[1]), 2, (0, 0, 255), 3)

    # Convert the features to NumPy arrays
    self.desired_features = np.array(features1)
    print("Desired_Features: ",self.desired_features)

    # cv2.imshow("Desired_Image",des_image)
    # cv2.waitKey(0)
    # cv2.destroyAllWindows()
    self.pub=rospy.Publisher("/cmd_vel",Twist,queue_size=10)
    rospy.Subscriber("/camera/rgb/image_raw",Image,self.callback)
    rospy.Subscriber("/odom",Odometry,self.pos_callback)
    rospy.sleep(0.1)
  def pos_callback(self,msg):
    self.current_depth=msg.pose.pose.position.x
    self.current_depth=self.current_depth * 1000
  def callback(self,msg):

    # print(msg.data)
    bridge=CvBridge()
    self.cv_image=bridge.imgmsg_to_cv2(msg,desired_encoding='passthrough')
    gray=cv2.cvtColor(self.cv_image,cv2.COLOR_BGR2GRAY)

    # Use HoughCircles to detect circles in the image
    self.circles = cv2.HoughCircles(gray,cv2.HOUGH_GRADIENT, dp=1, minDist=50, param1=50,
param2=30, minRadius=10, maxRadius=100)

    # print(circles)


  def visual_Ser(self):
    feature_e_norm_array=[]
    iterations_array=[]
    iterations=0
    # zd=174
    zd=229
    pos_e_norm=(np.linalg.norm(self.current_depth-zd))/1000
    feature_e_norm=0
    while( feature_e_norm<680):
      # If circles are found, draw them
      features=[]
      if self.circles is not None:
        self.circles = np.uint16(np.around(self.circles))
        # print("Entered")
        # rospy.sleep(20)
        for i in self.circles[0, :]:
            features.append([i[0],i[1],i[2]])
            # Draw the outer circle
            cv2.circle(self.cv_image, (i[0], i[1]), i[2], (0, 255, 0), 2)
            # Draw the center of the circle
            cv2.circle(self.cv_image, (i[0], i[1]), 2, (0, 0, 255), 3)

      current_features=np.array(features)
      # print("Current_Features: ",current_features)

      # # Save the result
      # output_path = 'output_image.jpg'
      # cv2.imwrite(output_path, self.cv_image)
      # rospy.sleep(10)

      # cv2.imshow("Current Image",cv_image)
      # cv2.waitKey(1)
      # return
# Cast matrices to a larger integer type or to float
self.desired_features = self.desired_features.astype(np.int64)
current_features = current_features.astype(np.int64)
rospy.sleep(0.2)
# Interaction Matrix
if(self.desired_features.shape != current_features.shape):
  rospy.loginfo("Out of Field of View of Camera")
  exit()
e=self.desired_features-current_features
# print("Desired Features Shape: ",self.desired_features.shape)
# print("Current Features Shape: ",current_features.shape)
# print("Features Error Norm: ",np.linalg.norm(e))

print("Current Features: ",abs(current_features))
print("Desired Features: ",abs(self.desired_features))

# Feature 1
u1=current_features[0][0]
v1=current_features[0][1]

# Feature 2
u2=current_features[1][0]
v2=current_features[1][1]

# Feature 3
u3=current_features[2][0]
v3=current_features[2][1]

# Feature 4
u4=current_features[3][0]
v4=current_features[3][1]

# zd=174  #In mm
f=825   #In mm
IM=[
  [-f/zd,0,u1/zd,(u1*v1)/f,-(f**2+u1**2)/f,v1],
  [0,-f/zd,v1/zd,(f**2+v1**2)/f,-(u1*v1)/f,-u1],

  [-f/zd,0,u2/zd,(u2*v2)/f,-(f**2+u2**2)/f,v2],
  [0,-f/zd,v2/zd,(f**2+v2**2)/f,-(u2*v2)/f,-u2],

  [-f/zd,0,u3/zd,(u3*v3)/f,-(f**2+u3**2)/f,v3],
  [0,-f/zd,v3/zd,(f**2+v3**2)/f,-(u3*v3)/f,-u3],

  [-f/zd,0,u4/zd,(u4*v4)/f,-(f**2+u4**2)/f,v4],
  [0,-f/zd,v4/zd,(f**2+v4**2)/f,-(u4*v4)/f,-u4]
  ]
# print("Interaction Matrix: ",np.linalg.pinv(IM))

e=e[:,0:2]
```

```
# print("error: ",e)
e_column_vector=e.reshape(-1,1)
# print("e_column: ",e_column_vector.size)
print("Error vector: ",e_column_vector)
ctrl=np.dot(np.linalg.pinv(IM),e_column_vector)
# print("Control: ",ctrl.shape)

#  Control Law
prop_ctrl=0.001*ctrl

vx=prop_ctrl[0][0]
vy=prop_ctrl[1][0]
omegaz=prop_ctrl[5][0]

pos=Twist()
pos.linear.x=vx
pos.linear.y=vy
pos.angular.z=10*omegaz
print("Angular velocity: ",10*omegaz)
self.pub.publish(pos)
# rospy.sleep(0.2)
pos_e_norm=(np.linalg.norm(self.current_depth-zd))/1000
print("Position_Error_Norm: ",pos_e_norm)

feature_e_norm=np.linalg.norm(e)
t=rospy.Time.now()
feature_e_norm_array.append(700-feature_e_norm)
print("Feature_Error_Norm: ",feature_e_norm)
print("\n")
iterations+=1
iterations_array.append(iterations)
if(feature_e_norm>700):
    pos.linear.x=0
    pos.linear.y=0
    pos.angular.z=0
    self.pub.publish(pos)
    # rospy.sleep(0.5)
    rospy.loginfo("Reached")
    # Plot
    plt.plot(iterations_array,feature_e_norm_array)
    # Add labels and title
    plt.xlabel('Iterations')
    plt.ylabel('Feature Error Norm')
    plt.title('Errror vs Time')
    plt.show()
    features1=[]
    for i in self.circles1[0, :]:
        # Extract circle features (center_x, center_y, radius)
        features1.append([i[0], i[1], i[2]])
        # Draw the outer circle
        cv2.circle(self.cv_image, (i[0], i[1]), i[2], (0, 255, 0), 2)
```

```
                    # Draw the center of the circle
                    cv2.circle(self.cv_image, (i[0], i[1]), 2, (0, 0, 255), 3)

            if self.circles is not None:
                self.circles = np.uint16(np.around(self.circles))
                for i in self.circles[0, :]:
                    features.append([i[0],i[1],i[2]])
                    # Draw the outer circle
                    cv2.circle(self.cv_image, (i[0], i[1]), i[2], (0, 255, 0), 2)
                    # Draw the center of the circle
                    cv2.circle(self.cv_image, (i[0], i[1]), 2, (0, 0, 255), 3)

            cv2.imshow("Current Image",self.cv_image)
            cv2.imshow("Desired Image",self.des_image)
            cv2.waitKey(0)
            # cv2.destroyAllWindows()

            exit()



if __name__=="__main__":
    try:
        rospy.init_node("Visual_Servoing_Node")
        obj=VS()
        obj.visual_Ser()
        rospy.spin()
    except KeyboardInterrupt:
        rospy.loginfo("Ctrl+C pressed. Shutting down the node.")
```

Explanation:

1. We are initially saving the desired image and capturing the desired depth.
2. Current features are calculated by subscribing continuously to topic /camera_raw_image.
3. Intercation matrix is calculated based on current features as:

```
IM=[
    [-f/zd,0,u1/zd,(u1*v1)/f,-(f**2+u1**2)/f,v1],
    [0,-f/zd,v1/zd,(f**2+v1**2)/f,-(u1*v1)/f,-u1],

    [-f/zd,0,u2/zd,(u2*v2)/f,-(f**2+u2**2)/f,v2],
    [0,-f/zd,v2/zd,(f**2+v2**2)/f,-(u2*v2)/f,-u2],

    [-f/zd,0,u3/zd,(u3*v3)/f,-(f**2+u3**2)/f,v3],
    [0,-f/zd,v3/zd,(f**2+v3**2)/f,-(u3*v3)/f,-u3],
    [-f/zd,0,u4/zd,(u4*v4)/f,-(f**2+u4**2)/f,v4],
    [0,-f/zd,v4/zd,(f**2+v4**2)/f,-(u4*v4)/f,-u4]
    ]
```

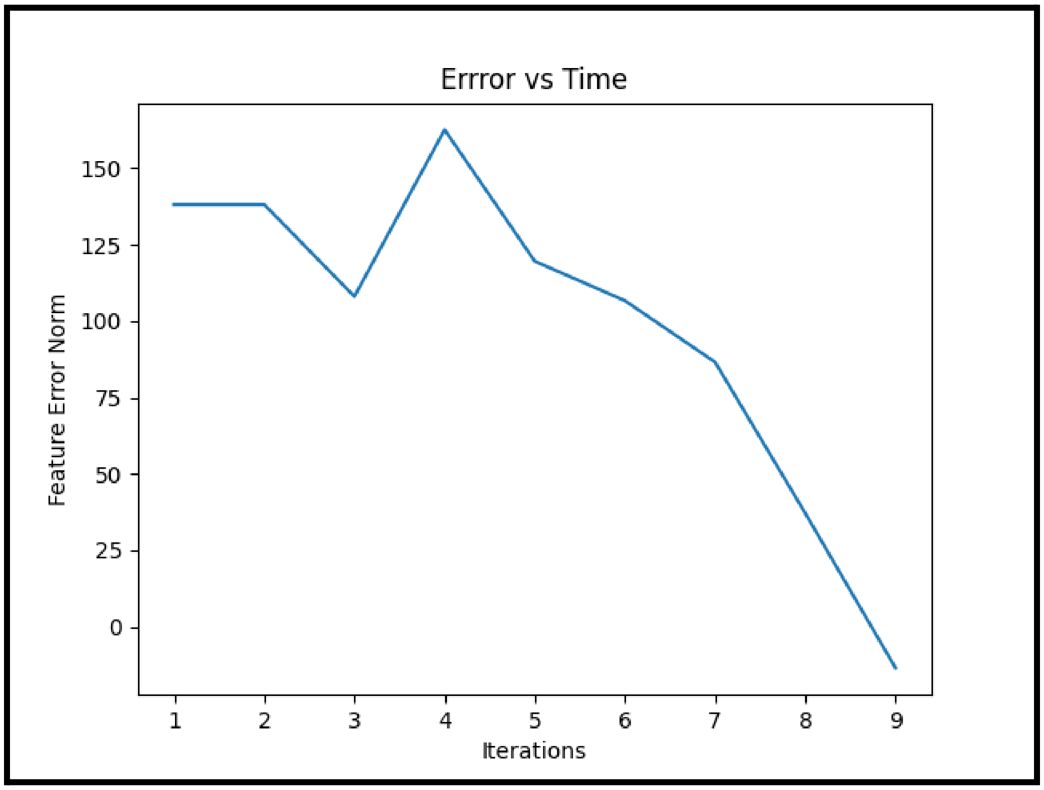
Here u and v are current features taken from the image.

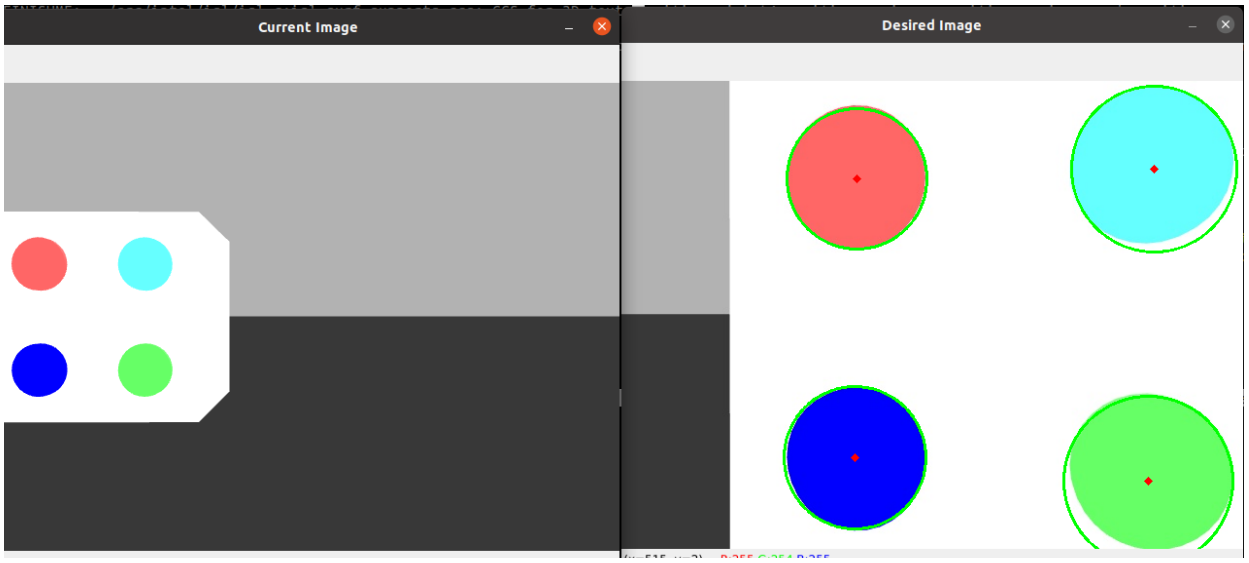
4. Then the control law is defined based on error between features and Pseudo inverse of interaction matrix:

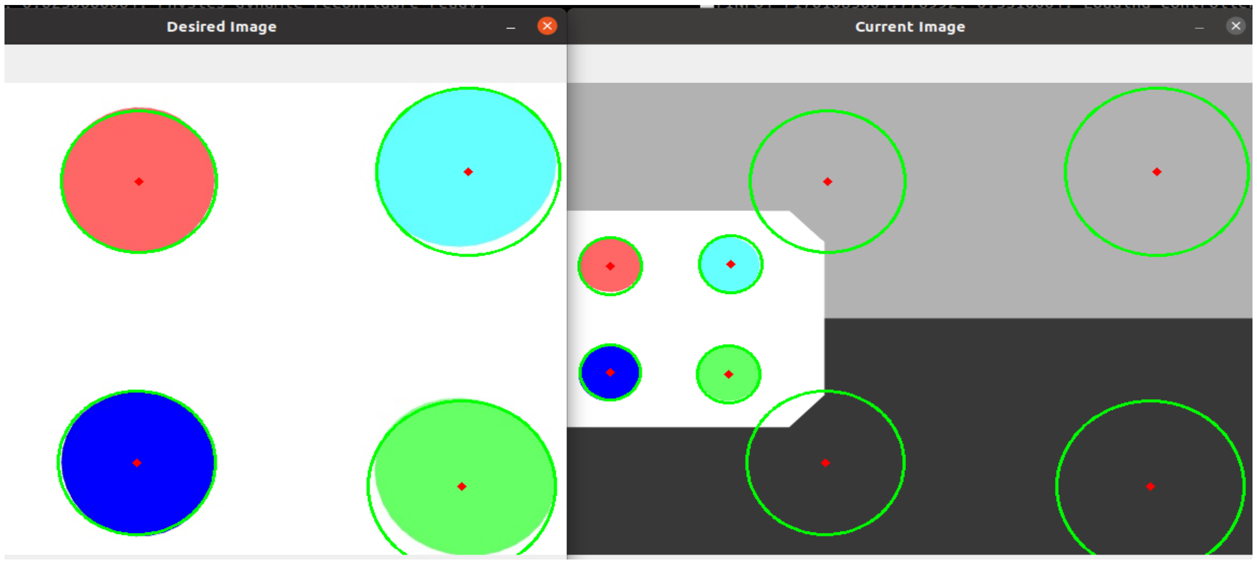
```
e=e[:,0:2]
    # print("error: ",e)
    e_column_vector=e.reshape(-1,1)
    # print("e_column: ",e_column_vector.size)
    print("Error vector: ",e_column_vector)
    ctrl=np.dot(np.linalg.pinv(IM),e_column_vector)
    # print("Control: ",ctrl.shape)

    #  Control Law
    prop_ctrl=0.001*ctrl
```

5. As the error norm reduces between the features , we are able to do the control of robot velocity based on velocity i =n image plane.

Results:







Problems Faced:

1. One of the major problem is that the error norm is not reducing but increasing, so I have taken the maximum value of error norm and normalizes it for ecah value based on the value required at desired Position.

2.It is moving back initially due to which sometimes the circles are not detecting in the camera and we loose the control of robot.

3. Control law is not same for all the steps, as we move closer to target I have to change the control law accordingly.
4. This code is not working to exactly locate robot at same position but can move it very close to desired position as verified from odometry of robot.

Video Link: https://youtu.be/c6GJrzBPL38