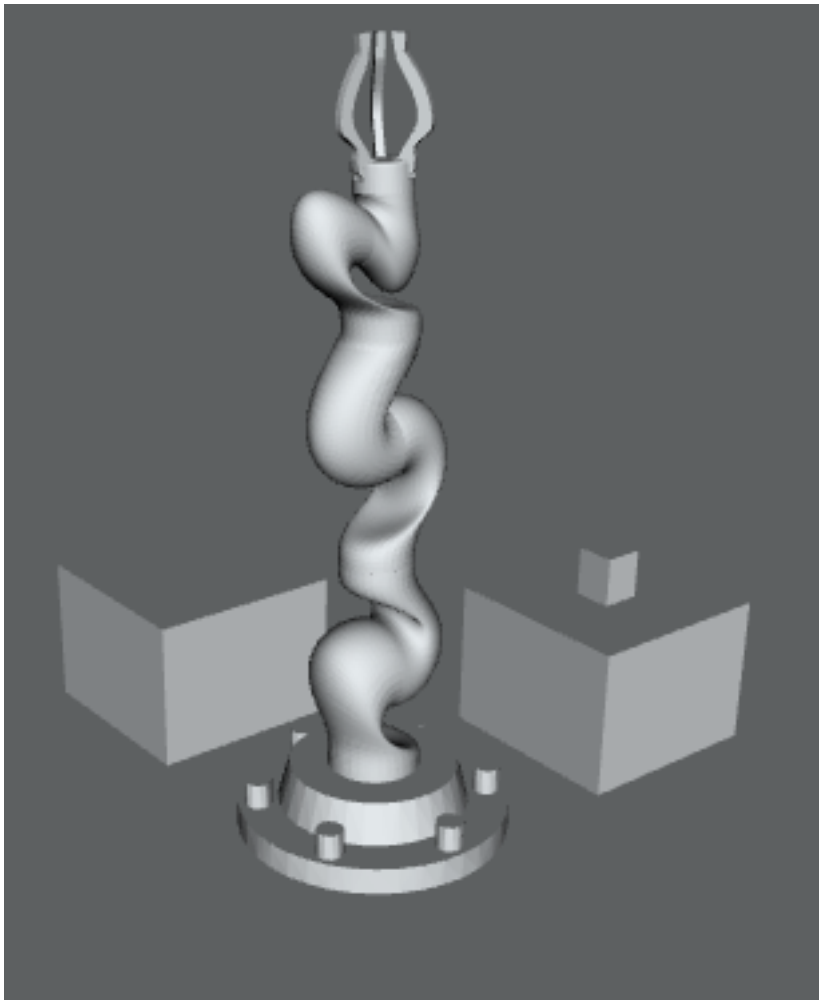
ENPM662: Introduction to Robot Modeling Final Project

Shantanu Suhas Parab (sparab), Vineet Kumar Singh(vsingh03)

December 2022



Contents

1 Abstract

KUKA LBR iiwa R820 is a 7R serial link manipulator. This robot was designed to be capable of very precise motion and repeatability and is inspired by the human hand. This project discusses the use of this robot manipulator with a gripper to pick up an object from one station and placing at another station within its reachable workspace.

The Kinematics of the robot which is forward and Inverse Kinematics, both position, and velocity, are analyzed. The results of the calculation are validated using a robot model simulated to perform the task in the Gazebo simulation environment.

2 Introduction

In this project, we modeled a robot based on the specifications of the KUKA LWR iiwa R820 robot. LBR stands for "Leichtbauroboter" which in German means lightweight robot. The term "iiwa" means intelligent industrial work assistant and belongs to the category of redundant robots designed to work alongside humans in a complex environment. It is designed for high precision and repeatability and has torque controllers at the joints to have very precise movements.

First, we modeled our robot parts in Solidworks and completed the assembly with a gripper. Once the model assembly was completed, we exported the model as URDF, and then worked on installing the controllers on the robot. We did the forward and inverse kinematics model and used it to derive the Transformation and Jacobian matrices for all joints.

Then the workspace analysis for the robot was done, using the transformation matrix and visualizing it in RVIZ. Based on the final workspace the simulation world was designed in Solidworks. Then we wrote publisher and subscriber nodes to control all the joints of the robot and simulate the robot in Gazebo. Once the nodes were communicating perfectly, then we performed the objective of robot end-effector tracking the desired trajectory.

2.1 Motivation

The motivation for building this robot lies in the current manufacturing field, where the need for dynamic assembly lines is increasing day by day. This calls for collaborative assembly lines, where humans can work alongside robots to harmonize the assembly process. Our robot is designed keeping this in mind. It is a redundant robot that allows for multiple configurations for any particular position and thus can have various poses within its workspace. This allows for easy reconfiguring of any changes in its environment. This characteristic also leads to Human-Robot Collaborative (HRC) workspaces, and the assembly lines are much more dynamic and efficient to reconfigure.

2.2 Application

These robots belong to the class of co-bots which are capable of HRC workspaces and are used in many industries which need precision and are capable to work in environments with Human Interaction. The major factor for its adaptation is its HRC compatibility and precision which is difficult to achieve for redundant robots with high DOF. Its adaptability and ease of reconfiguration along with modular design have made these much more popular in fields that need dynamic robots with ease of use and modification as per specific requirements.
A few examples include

1. Used in high-precision manufacturing lines as torque control provides precise repeatability.
2. Used in medical robotics for its Human-Robot Collaborative workspace certifications. Used in rehabilitation for humans.
3. Used in dynamic assembly lines which need frequent changes to the manufacturing plan.

3 Robot Description

Our robot model mimics the specifications of the KUKA LBR iiwa R820 robot manipulator. It has 7 degrees of freedom and 6 links in the robot arm. All the arm joints are revolute, and they have a 3-link gripper as an end effector. We designed a custom gripper for this project which has three curved fingers and each of them is actuated independently. In home position, the links of the robots have axes aligned in a straight line and have no lateral displacement. The links with the vertical axis in the arms have a motion range of ±170°and those with horizontal axis have ±120°. The gripper fingers have a rotation limitation of 90°.

3.1 Workspace Study

The robot design has a total height of 1.42m of which the robot arm length is 1.22m (Figure 2). The detailed dimensions (2D Drawing) of the robot are given in Appendix 1. All the joints in the robot are revolving, and have a rotational limit of about 170°(except the joints of the end effectors which have a rotational limit of 90°). The robot workspace and its mapping are thus given below as per Figure 1.

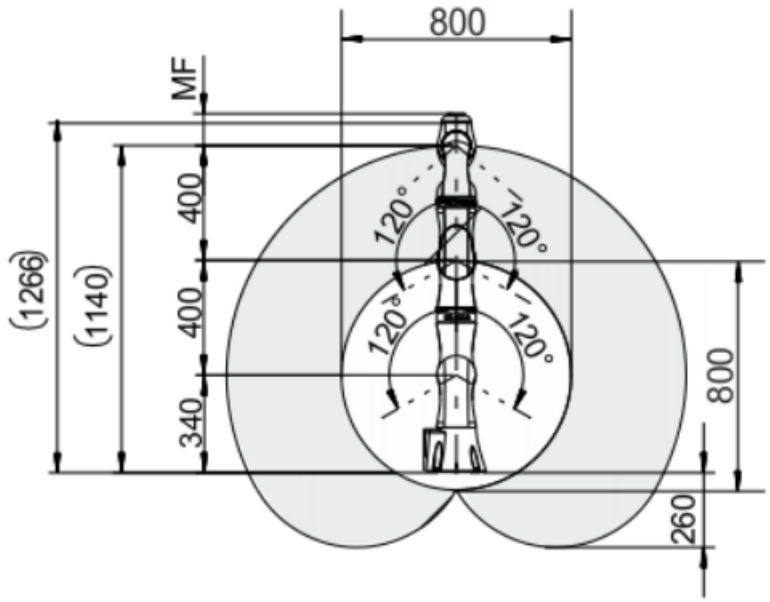


Figure 1: Workspace of Robot

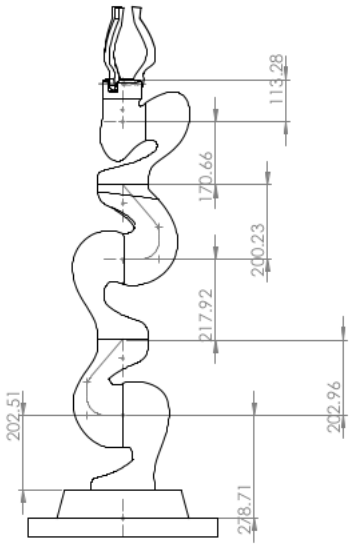


Figure 2: Link Lengths of our Model

4 Model Assumptions

In order to do the kinematic analysis the following assumptions were taken:

- The position of the objects to be picked up is fixed, and there is no sensing element on the robot.
- There is no real-time path planning done for the arm. It travels a fixed predefined path to complete the objective of picking and placing the object.
- In dynamics modeling, we only considered the Potential energy due to gravity, and other components were neglected as the arm is moved very slowly to attain the goal.
- The contact model of the robot gripper with the object is assumed to be constant for all fingers and does not vary significantly with time.
- There is no back slip for the gears of the motors.
- The motors can provide any value of torque needed by the robot joints.
- The robot base is grouted to sustain any weight put on it.
- There is no mechanical loss in the transmission of torque/force.

5 Kinematics

The Kinematics model analyzes the motion of any physical system without considering the forces/torque that causes it. It basically gives an idea about the movement of the body in space. For the study of robots, this allows us to compute the position and orientation of the robot's different links/joints/end-effectors in terms of its joint variables. The kinematics model can be Forward or Inverse Kinematics, where the former is used to calculate the values of the end effector when the joint variables are defined, and the latter

Link	d_i	a_i	α_i	α_i
1	d_1	0	$\theta_1 - \frac{\pi}{2}$	$-\frac{\pi}{2}$
2	0	0	θ_2	$\frac{\pi}{2}$
3	d_3	0	θ_3	$\frac{\pi}{2}$
4	0	0	θ_4	$-\frac{\pi}{2}$
5	d_5	0	θ_5	$-\frac{\pi}{2}$
6	0	0	θ_6	$\frac{\pi}{2}$
7	d_7	0	θ_7	0

Table 1: DH Parameter

does the vice-verse (i.e. computes the values of the joint variables, given the position and orientation of the end effector).

To represent the forward kinematics model mathematically, let us define a function f between the joint space R^n and the workspace R^m:

$$x = f(q); x \in R^m, q \in R^n$$

Inverse Kinematics is defined as a function:

$$q = g(x) = f^-1(x); q \in R^n, x \in R^m$$

Using the forward and inverse kinematics models, one can mathematically define the motion of the robot's individual links in the 3D space.

5.1 DH Parameters

In home position. Table available. Type in the Latex final report. DH Parameters are commonly used to define the parameters of the joint variables of a robot manipulator. It is efficient as it needs only 4 variables (a, α, d, θ) to fully define the position and orientation of the robot as compared to 6 variables (x, y, z, ϕ, θ, ψ) needed if defined using conventional methods.

The four parameters of the DH parameters are respectively:

- θ_i : angle between x_{i1} and x_i measured about z_{i1}
- d_i : distance along z_{i1} to the common normal
- a_i: distance along x_i to the common normal
- α_i : angle between $xi1$ and x_i measured about z_{i1}

There are two rules to be followed for the frame assignment as per the conventional DH method:
1. X_i should intersect with z_{i-1} for all the joints.
2. Z is always along the axis of rotation for a revolute joint and the direction of motion for a prismatic joint.

After frame assignment, the DH parameters were defined for all the joints as given in Table 1.

The DH frame assignment used is given in Figure 3. The dimensions of link lengths in DH parameters are as below:

d_1: 328.71mm
d_3 : 420.90mm
d_5 : 370.92mm
d_7 : 113.45mm
d_7 - end effector: 195.46mm

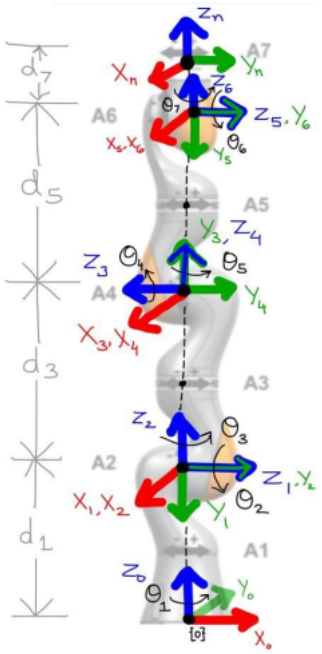


Figure 3: DH frame assignment

5.2 Forward Kinematics

Forward Kinematics defines the study of the robot's motion and position of its links when each of the joints is actuated. For this study, we use matrix transformations for efficient calculations in a single matrix. These are called Homogeneous transformations matrices denoted by H and are given by:

$$H = H_n^0 = \prod_{i=1}^{n} A_i^{i-1}(\theta_i)$$

Where A matrix is given as

$$A_i^{i-1} = \begin{bmatrix} cos\theta_i & -sin\theta_i cos\alpha_i & sin\theta_i cos\alpha_i & a_i cos\theta_i \\ sin\theta_i & cos\theta_i cos\alpha_i & -cos\theta_i cos\alpha_i & a_i sin\theta_i \\ 0 & sin\alpha_i & cos\alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

We calculate the A matrix for all the links, and it gives the transformation of the n^{th} joint with respect to the $n-1^{th}$ joint. The first 3*3 row and column of the matrix denote the rotation of the nth frame w.r.t to the $n-1^{th}$ frame. The three elements of the last column denote the position (x,y,z) of the nth link in the $n-1^{th}$ frame.

We use the above form to calculate the individual A matrices for all joints.

Transformation matrices are used to define any link w.r.t. the base frame. To get the transformation of end-effector w.r.t. the base frame is given as
$T_n^0 = A1 * A2 * A3 * A4 * A5 * A6 * A7$
The result of the transformation matrix for each link and the final transformation matrix of the end effector w.r.t base is included in Appendix 2.

5.3 Inverse Kinematics

The robot motion in terms of joint angles is discussed in the previous section, but in a real environment, the problem that needs to be solved is a bit different. Most of the use cases need the robot end effector to be at a given position and every joint parameter should be adjusted to reach the given position. This is the opposite of Forward Kinematics and is called Inverse Kinematics.

Inverse kinematics for a robot can have multiple approaches, and each one of these has its own advantages and disadvantages. We will discuss the method of Jacobian which is used in this project for Inverse Velocity Kinematics.

The velocity relationships are then determined by the Jacobian of this function. The Jacobian is a matrix that can be thought of as the vector version of the ordinary derivative of a scalar function. It is one of the most important quantities in the analysis and control of robot motion when the input to the system is in terms of velocity profiles. The concept of Jacobian matrices, their construction, and their uses are discussed briefly.

Let us consider an n-link manipulator with joint variables $q_1, q_2, ..., q_n$. Let

$$T_n^0(q) = \begin{bmatrix} R_n^0(q) & o_{n0}(q) \\ 0 & 1 \end{bmatrix}$$

denote the transformation from the end-effector frame to the base frame, where $q = (q_1, ..., q_n)^T$ is the vector of joint variables. As the robot moves about, both the joint variables q_i and the end-effector position $o_n^0$and orientation R_n^0 will be functions of time. Relation between the angular velocity of the end-effector to the vector of joint velocities $\dot{q}(t)$ is needed for further analysis. Let

$$S(\omega_n^0) = \dot{R}_n^0 (R_{n0})^T$$

define the angular velocity vector ω_n^0 of the end-effector, and let

$$v_n^0 = \dot{o}_n^0$$

denote the linear velocity of the end effector. We seek expressions of the form

$$v_n^0 = J_v \dot{q} \omega_n^0 = J_\omega \dot{q}$$

where J_v and J_ω are 3 × n matrices. We may write the above equations together as:

$$\xi = J\dot{q}$$

in which ξ and J are given by:

$$\xi = \begin{bmatrix} v_n^0 \\ \omega_n^0 \end{bmatrix}; J = \begin{bmatrix} J_v \\ J_\omega \end{bmatrix}$$

The vector ξ is called a body velocity and the matrix J is called the Manipulator Jacobian or Jacobian for short. J is a 6 $\times$ n matrix where n is the number of links. We next derive the calculation of individual Jacobian matrix elements.

The upper 3 rows of the Jacobian matrix are made up of the linear velocity components of each joint of the arm and is given as:

$$J_v = [J_{v1}, J_{v2}, ..., J_{vn}]$$

where the i^th column of the J_{vi} is given as:

$$J_{vi} = z_{i-1} * (o_n - o_{i-1})$$

for revolute joint i.

The low 3 rows of the Jacobian matrix are made up of the angular velocity components of each joint of the arm and is given as:

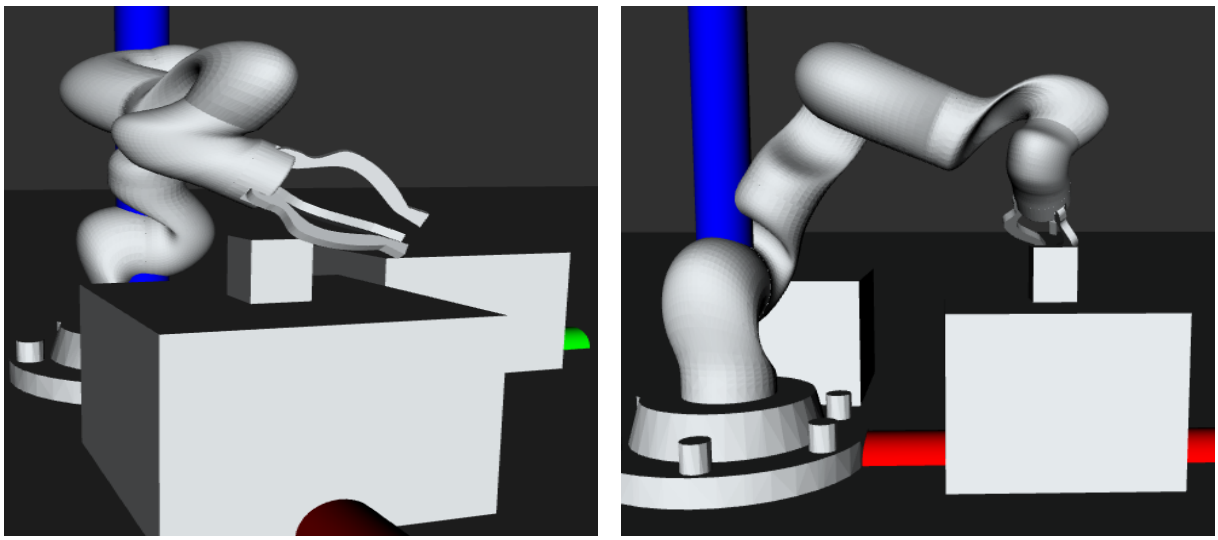
$$J_\omega = [J_{\omega 1}, J_{\omega 2}, ..., J_{\omega n}]$$

where the i^th column of the $J_{\omega i}$ is given as:

$$J_{\omega i} = z_{i-1}$$

for revolute joint i

Then we worked on the end effector orientation w.r.t. the object to enable the gripper to properly pick up the object, and it was an important concept in this project. As shown in Figure 4, even though the end effector is at the final position as per the Inverse kinematics calculation, the orientation of the gripper is not correct to pick up the object properly.

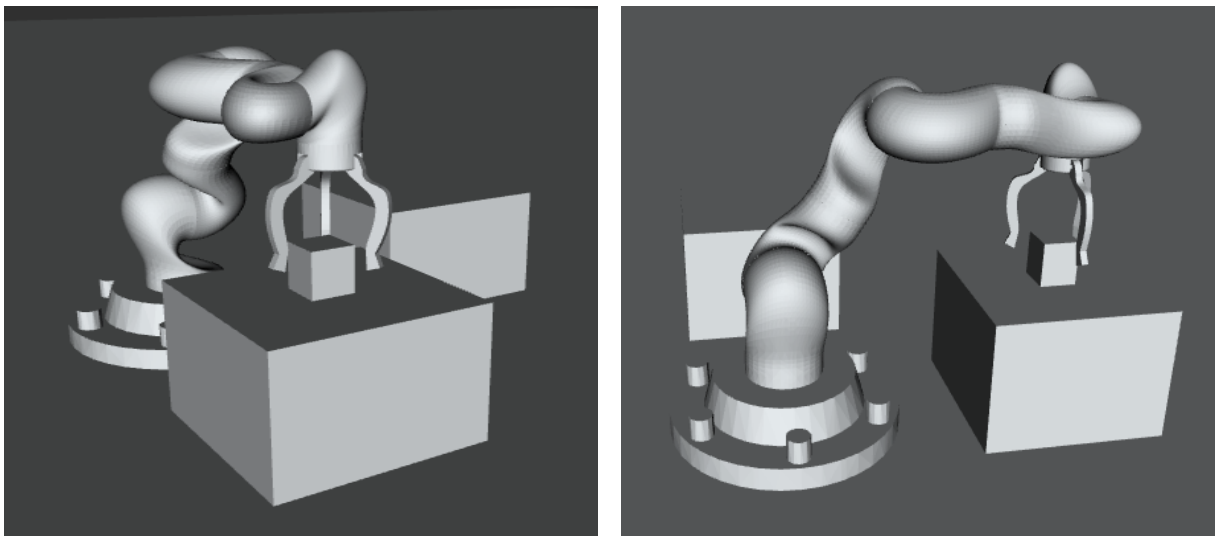


(a) Camera Angle 1 (b) Camera Angle 2

Figure 4: Incorrect Positioning of End effector

The approach taken was to align the n-1 link directly above the object at a suitable height. Then the end effector was rotated to align vertically above the object, and this was implemented using the z-axis coordinate of the object frame to be aligned with the origin of the $(n-1)^{th}$ and n^{th} link axes (Figure 5). Once the links are aligned, then the gripper arms are actuated with torque controllers to grip the object.

The final Jacobian Matrix calculated is shown in Appendix 1.



(a) Camera Angle 1 (b) Camera Angle 2

Figure 5: Correct Position of End Effector

5.4 Forward Kinematics Validation

The forward kinematic equations were derived for 5 different configurations, and the values for joint variables were taken for some random cases and then run through code to move the robot in Gazebo. The robot poses from Gazebo and the corresponding joint angles are as shown below (Fig 6).

5.5 Inverse Kinematics Validation

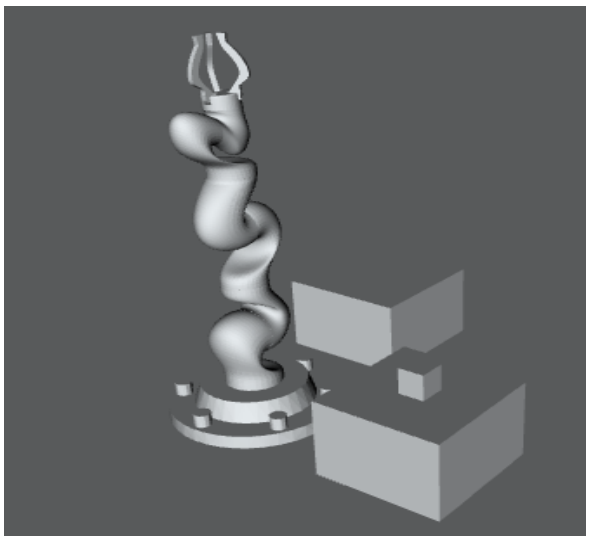
The validation of the Inverse Kinematics is done by plotting the desired trajectory of the robot as a 3D plot in code. The desired velocity plot is given as input to the I.K. solver and it outputs the end effector poses. Figure 7 shows the trajectory followed by the robot end effector in 3D space and its projection on the ground frame. The robot first picks up the object and then follows the path to another station and keeps the object there.

In Figure 8, the robot gripper can be tracked to move down to the object location and then track the path to the next platform. Figure 9 shows the path followed as an almost straight line, which conforms to our desired result based on the given velocity profile.

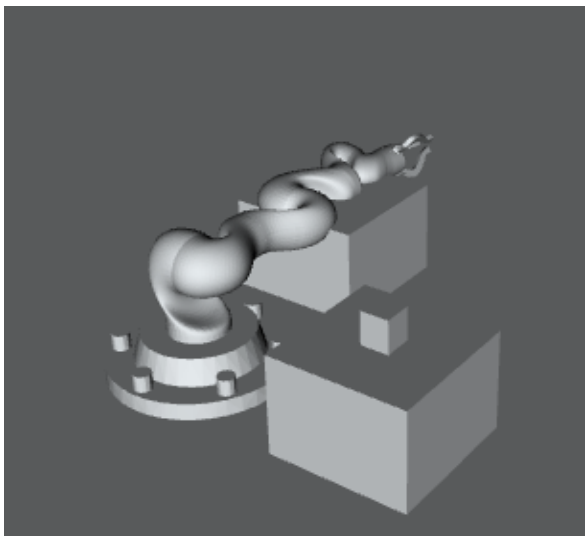
The final result/simulation also serves as a validation for the Inverse Kinematics solution.

6 Control method

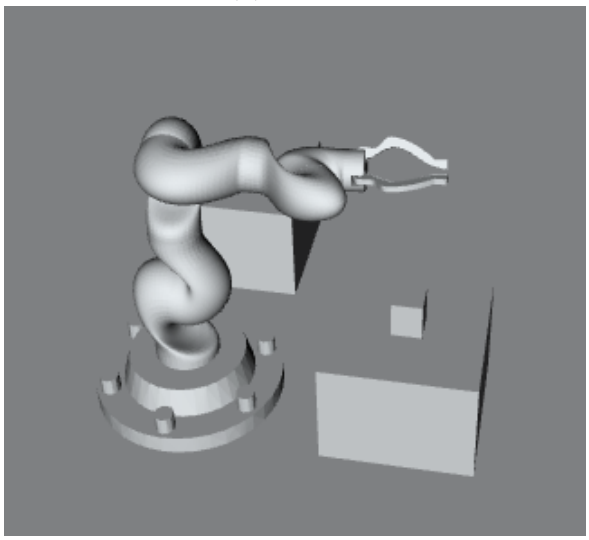
In this project, we used an open loop scheme to control the robot's trajectory as the robot lacks any sensing element. The joints are equipped with a PID controller to tune the motion. Each joint of the robot arm is equipped with an Effort Position controller, as each joint is being published the next coordinates that it should go to. Robots deployed in the real world usually use torque controllers as the dynamic model is also considered for the robot motion. But in our case, the velocity is considered almost zero, so the dynamic model is ignored, and so we decided to use the Position controller for our arm joints. The fingers of the gripper are actuated with torque controllers, as this provides the continuous force, and thus friction, for the objects to be picked up. The PID is tuned individually for each joint to reduce the error in tracking.



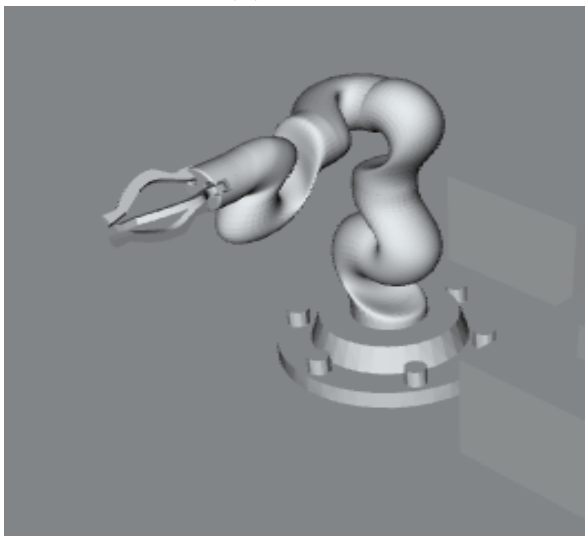
(a) θ_1=90°



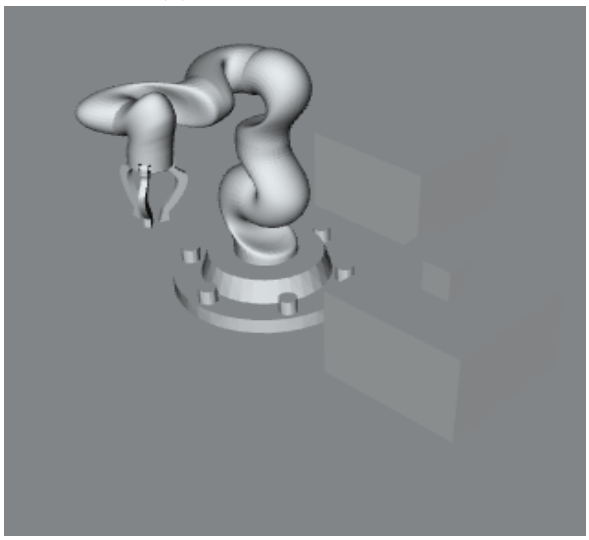
(b) θ_2=90°



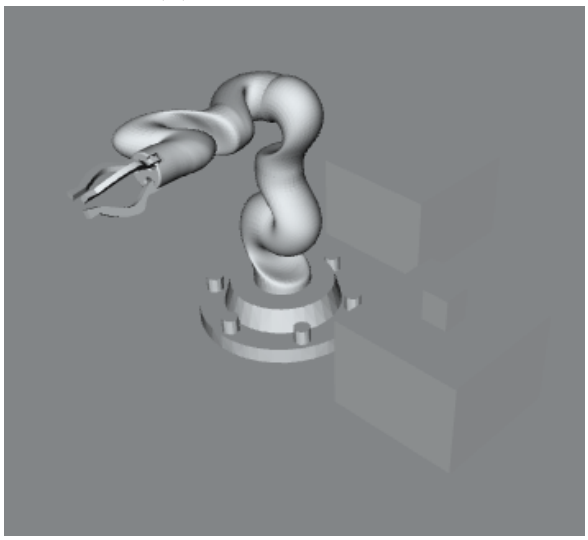
(c) θ_3=90°and θ_4=90°



(d) θ_4=90°and θ_5=90°



(e) θ_4=90°and θ_6=90°



(f) θ_4=90°

Figure 6: Robot Positions in Various Joint Angle Configuration

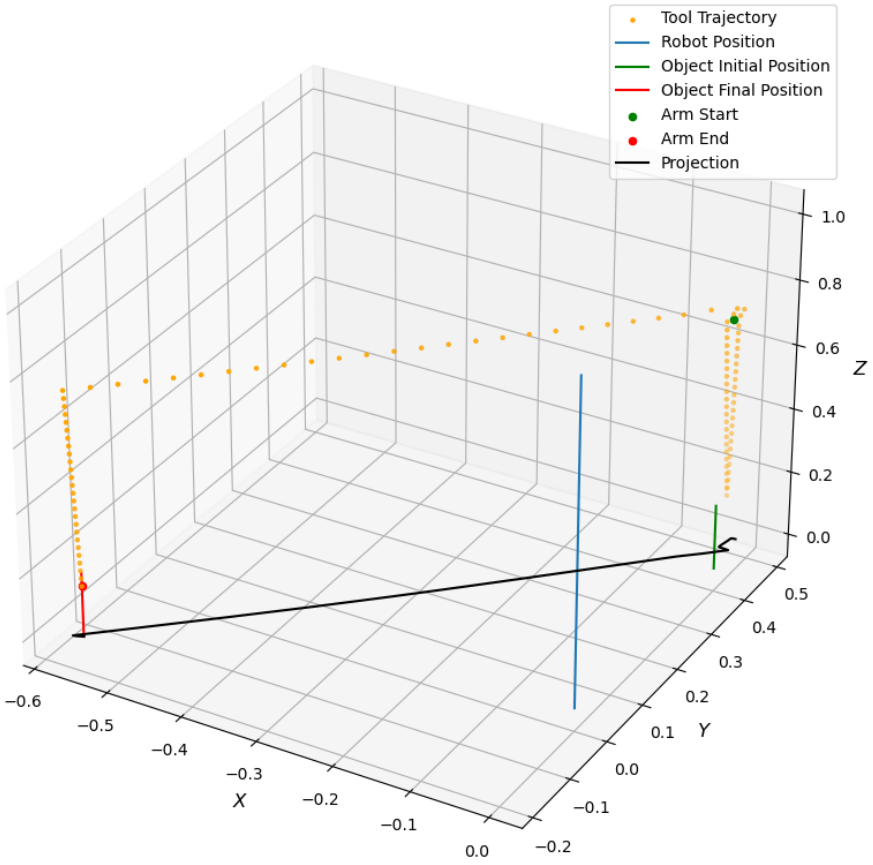


Figure 7: 3D trajectory followed by end effector

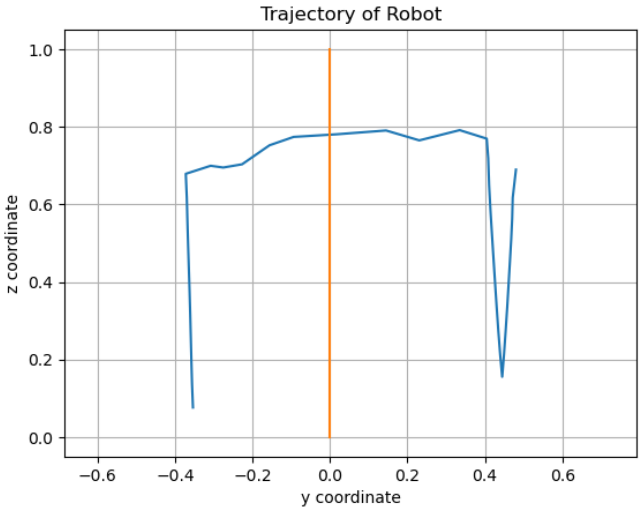


Figure 8: Trajectory as seen in XZ plane

7 Gazebo Simulation

The simulation was done in Gazebo, and the world was designed in Solidworks. It has objects placed on stations, and the robot will pick objects from one station and place them on another station. Figure 10 shows the initial pose robot when launched in the simulation world in the Gazebo.

The publisher node computes the forward and inverse kinematics based on the velocity profiles provided. It publishes the positions for each joint at the predefined frequency.

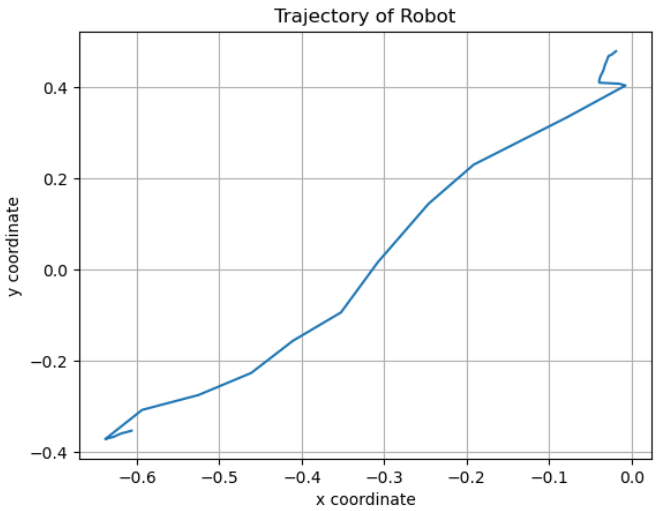


Figure 9: Trajectory as seen in XY plane

The Subscriber node subscribes to the published frequency and sends these to Controllers on the robot. Based on these poses, the robot joint moves to the desired location. Once at the desired location, the gripper grasps the object, picks up the object, and then places it in the desired location. Link to the Simulation

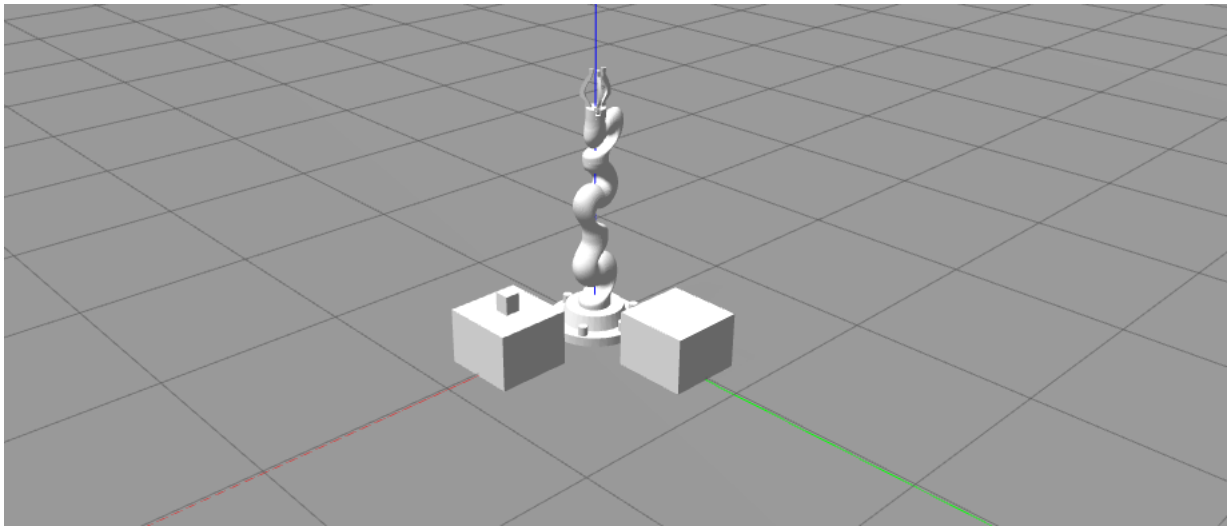


Figure 10: Launch of Robot in Gazebo

8 Problems Faced

The major bottleneck was to choose such a trajectory where the Inverse Jacobian is invertible for all the points on the trajectory. We had to reiterate multiple times to understand the behavior of the joint motions and their singularities. We also faced issues because of loss of accuracy because of rounding off errors in cartesian coordinates. This led us to do manual adjustments to the trajectory to ensure the path is as linear in a single axis as possible. We planned the path as multiple velocity profiles to meet the desired trajectory.

9 Lessons Learned

We observed that the analytical method is much faster to compute than the Jacobian method. However, it is difficult to get the equations using the Analytical method in the first place. We also learned that it is very useful to use Quaternions for such robots, as the consolidated effects of rounding errors are significant when using Cartesian coordinates and thus lead to much deviation in a trajectory which is undesirable. Using Quaternions is also fast and computes quickly compared to Cartesian or polar.

10 Conclusion

In this project, we analyzed the kinematics of the 7-link manipulator. The study of forward kinematics gave the position of the end effector for certain joint angles. The values of joint angles for the desired end effector position were calculated using Inverse Kinematics. The model was simulated in Gazebo and verified using a simulation world environment to pick and place the object which validated our solution.

11 Future work

In continuation to this project, future work can include three major aspects:
1. Including path planning for the robot arm which gives it the flexibility to pick objects at any random location in its workspace.
2. Inclusion of perception component so that robot can detect its surroundings.
3. Analysis of the dynamic model of the robot and use of Quaternion instead of the Cartesian coordinate system.

12 Reference

[1] : Inverse kinematics of the KUKA LBR iiwa R800 (7 DOF) by Sebastian Doliwa, Oct 2020, publisher = Zenodo, doi = 10.5281/zenodo.4063575

[2] : Carlos Faria Wolfram Erlhagen Flora Ferreira Sergio Monteiro. Position-based kinematics for 7-DOF serial manipulators with global configuration control, joint limit, and singularity avoidance. Technical report, 2018.

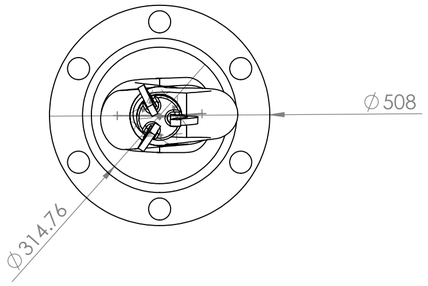
[3] : Robot Dynamics and Control, by Spong and Vidyasagar (Wiley, 1989).

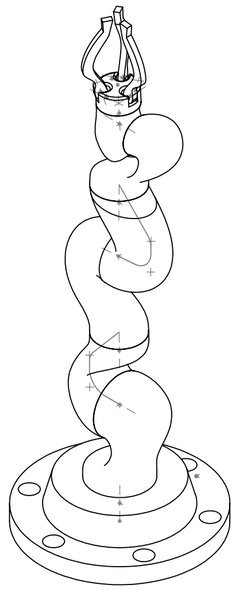
[4] : KUKA LBR iiwa R82 brochure and specification sheet Issued: 28.01.2015. Version: Spez LBR iiwa V5.

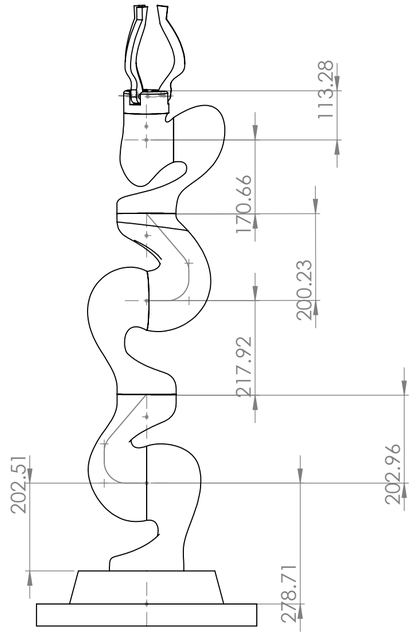
[5] : Solution HW3 ENPM662 F'22.

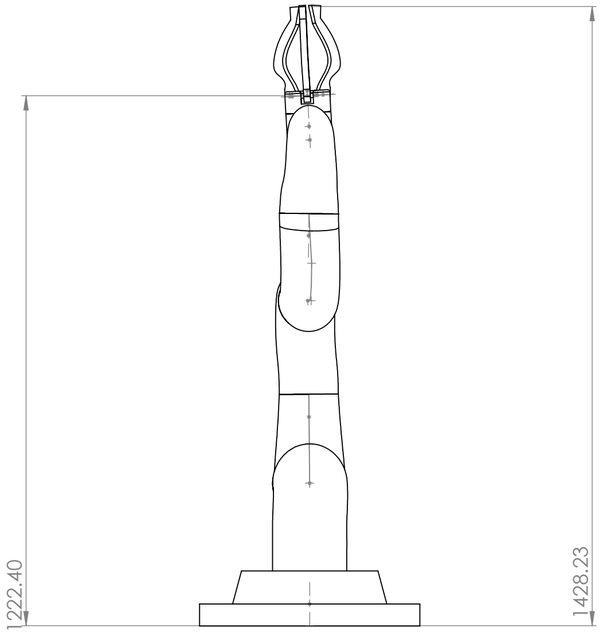
[6] : Lecture Notes.

Appendix 1









UNLESS OTHERWISE SPECIFIED: DIMENSIONS ARE IN MILLIMETERS SURFACE FINISH: TOLERANCES: LINEAR: ANGULAR:	FINISH:		DEBURR AND BREAK SHARP EDGES	DO NOT SCALE DRAWING	REVISION	1
				Units: MMGS		
	NAME	SIGNATURE	DATE	TITLE: ENPM662 Final Project KUKA LWR iiwa Custom Robot Team Members: Shantanu Parab Vineet Singh		
DRAWN	Vineet Singh					
CHK'D						
APPV'D						
MFG						
Q.A				MATERIAL:	DWG NO. kuka_robot	A2
				WEIGHT:	SCALE:1:10	SHEET 1 OF 1

Appendix 2

December 7, 2022

[59]:
```
from sympy import Matrix, symbols, cos, sin, simplify, pi, pprint, diff
from numpy import linspace, meshgrid, ones_like
import matplotlib.pyplot as plt
import time
import math
# Simulation parameters
tool_length = 10      # (cm)
N = 100               # no. of data points
delta_time = 20.0/N      # time between successive data points
# Panda Link offsets (cm)
d1 = 0.32871
d3 = 0.42090
d5 = 0.37092
d7 = 0.11345
# Defining the symbolic joint angle symbolic variables
q1, q2, q4, q5, q6, q7 = symbols('q1, q2, q4, q5, q6, q7')
# Function to obtain Transformation matrix between consecutive links
def get_tf(q,d,a,alpha):
    T =␣
 ↪Matrix([[cos(q),-sin(q)*cos(alpha),sin(q)*sin(alpha),a*cos(q)],[sin(q),cos(q)*cos(alpha),-cos
 ↪[0,0,0,1]])
    return T
# D-H table for the Panda robot as given in the question.
T_01 = get_tf(q1-(pi/2),  d1,   0,  -pi/2) # A_1
T_12 = get_tf(q2,   0,   0, pi/2)
T_23 = get_tf(0,   d3,  0, pi/2)
T_34 = get_tf(q4,   0, 0,  -pi/2)
T_45 = get_tf(q5,  d5,   0, -pi/2)
T_56 = get_tf(q6,   0,  0, pi/2)
T_6n = get_tf(q7, d7,   0,     0)
T_02 = T_01 * T_12 # A_2
T_04 = T_02 * T_23 * T_34 # A_4
T_05 = T_04 * T_45 # A_5
T_06 = T_05 * T_56 # A_6
T_0n = T_06 * T_6n # A_7
print("\n**************************************************************************")
print("       Transformation Matrix (T_0n) between end-effector & base frames:")
```

```
print("**************************************************************************\n")
pprint(T_0n)
# Setting up the Jacobian Matrix
Z_0 = Matrix([0, 0, 1])
Z_1 = simplify(T_01[0:3,2])
Z_2 = simplify(T_02[0:3,2])
Z_4 = simplify(T_04[0:3,2])
Z_5 = simplify(T_05[0:3,2])
Z_6 = simplify(T_06[0:3,2])
#######################################
# Using Method-2 to compute Jacobian
#######################################
x_p = T_0n[0:3,3]
h_1 = diff(x_p,q1)
h_2 = diff(x_p,q2)
h_3 = diff(x_p,q4)
h_4 = diff(x_p,q5)
h_5 = diff(x_p,q6)
h_6 = diff(x_p,q7)
J_v1 = simplify(h_1)
J_v2 = simplify(h_2)
J_v3 = simplify(h_3)
J_v4 = simplify(h_4)
J_v5 = simplify(h_5)
J_v6 = simplify(h_6)
#######################################
# Using Method-1 to compute Jacobian
#######################################
# O_0 = Matrix([0, 0, 0])
# O_1 = T_01[0:3,3]
# O_2 = T_02[0:3,3]
# O_4 = T_04[0:3,3]
# O_5 = T_05[0:3,3]
# O_6 = T_06[0:3,3]
# O_n = T_0n[0:3,3]
# J_v1 = simplify(Z_0.cross((O_n-O_0)))
# J_v2 = simplify(Z_1.cross((O_n-O_1)))
# J_v3 = simplify(Z_2.cross((O_n-O_2)))
# J_v4 = simplify(Z_4.cross((O_n-O_4)))
# J_v5 = simplify(Z_5.cross((O_n-O_5)))
# J_v6 = simplify(Z_6.cross((O_n-O_6)))
J_v = Matrix().col_insert(0,J_v1).col_insert(1,J_v2).col_insert(2,J_v3).
 ↪col_insert(3,J_v4).col_insert(4,J_v5).col_insert(5,J_v6)
J_w = Matrix().col_insert(0,Z_0).col_insert(1,Z_1).col_insert(2,Z_2).
 ↪col_insert(3,Z_4).col_insert(4,Z_5).col_insert(5,Z_6)
J = Matrix().row_insert(0,J_v).row_insert(3,J_w)
print("\n**************************************************************************")
```

```
print("                              Jacobian Matrix (J):                    " )
print("*************************************************************************\n")
pprint(J)
# Generating circle velocity trajectory using cylindrical coordinate equations␣
 ↪of circle points
def generate_circle_velocity(i):
#     x_dot = -0.04*pi*sin(th)
#     z_dot = 0.04*pi*cos(th)
#     X_dot = Matrix([x_dot, 0.0, z_dot, 0.0, 0.0, 0.0]) # Generalized␣
 ↪end_effector cartesian velocity components
    v=0.02
    ax=0.1*cos((2*pi*i)/100)*(3*pi)/5
    ay=0.1*sin((2*pi*i)/100)*(3*pi)/5
    X1=Matrix([[0],[0],[v*2],[0],[0],[0]])
    X2=Matrix([[0],[0],[v*2],[0],[0],[0]])
    X3=Matrix([[-v*6],[-v*6],[0],[0],[0],[0]])
    X4=Matrix([[0],[0],[-v*6],[0],[0],[0]])

    X_profile=[X1,X2,X3,X4]
    if(i<N/4):
        X_eval=X_profile[0]
    elif(i<N/2):
        X_eval=X_profile[1]
    elif(i<3*N/4):
        X_eval=X_profile[2]
    elif(i<N):
        X_eval=X_profile[3]
    print(i)
    return X_eval
# Function to compute the inverse kinematics q_dot values from end-effector␣
 ↪velocities.
def inverse_velocity_kinematics(X_dot, q_joint):
    offset=0.0000001
    J_inv = J.evalf(subs={q1: q_joint[0]+offset,q2: q_joint[1]+offset, q4:␣
 ↪q_joint[2]+offset, q5:
q_joint[3]+offset, q6: q_joint[4]+offset, q7: q_joint[5]+offset}).pinv()
    q_dot = J_inv * X_dot # Generalized joint velocity components from Jacobian␣
 ↪inverse
    return q_dot
# Function to compute new value of joint angle based on q_dot and previous angle
def update_joint_angle(q, q_dot):
    q = q + q_dot*delta_time
    return q
# Function to compute forward kinematics (position) to obtain the end-effector␣
 ↪(Pen) (x,y,z) co-ords wrt base frame
def forward_position_kinematics(q):
```

```
    T = T_0n.evalf(subs={q1: q[0], q2: q[1], q4: q[2], q5: q[3], q6: q[4], q7:
 ↪q[5]})
    return (T[0,3].round(4),T[1,3].round(4),T[2,3].round(4))

x_tool=[]
y_tool=[]
z_tool=[]

if __name__ == '__main__':
    fig = plt.figure()
#     ax = fig.add_subplot(111, projection='3d')
#     ax.axes.set_xlim(left=-10, right=90)
#     ax.axes.set_ylim(bottom=-30, top=55)
#     ax.axes.set_zlim(bottom=0, top=85)
#     ax.set_xlabel('X')
#     ax.set_ylabel('Y')
#     ax.set_zlabel('Z')
    q_joint = Matrix([0.0, math.radians(-40.0), math.radians(100.0),0.0,math.
 ↪radians(30.0), 0.0]) # initial joint angles of the robot

    # Looping cylindrical coordinate theta from pi/2 to 5pi/2 for full circle
    for theta in range(N):
        circle_vel_traj = generate_circle_velocity(theta)
        q_dot_joint = inverse_velocity_kinematics(circle_vel_traj, q_joint)
        q_joint = update_joint_angle(q_joint, q_dot_joint)
        (x_0p, y_0p, z_0p) = forward_position_kinematics(q_joint)
        x_tool.append(x_0p)
        y_tool.append(y_0p)
        z_tool.append(z_0p)

        # plot on circle as live points from forward kinematics for every␣
 ↪delta_time seconds
#         ax.scatter(x_0p,y_0p,z_0p)        plt.pause(delta_time)
#     plt.show()
```

```
**********************************************************************
       Transformation Matrix (T_0n) between end-effector & base frames:
**********************************************************************

(((sin(q)sin(q)sin(q) + sin(q)cos(q)cos(q))cos(q) + sin(q)cos(q

(((-sin(q)sin(q)cos(q) - cos(q)cos(q)cos(q))cos(q) + sin(q)sin(

                                          ((-sin(q)sin(q) - cos(q)cos(
```

```
))cos(q) + (-sin(q)sin(q)cos(q) + sin(q)sin(q)cos(q))sin(q))co
q))cos(q) + (sin(q)cos(q)cos(q) - sin(q)cos(q)cos(q))sin(q))co
q))sin(q) + (-sin(q)cos(q) + sin(q)cos(q))cos(q)cos(q))cos(q) -
                                                    0
s(q) + (-(sin(q)sin(q)sin(q) + sin(q)cos(q)cos(q))sin(q) + cos(q
s(q) + (-(-sin(q)sin(q)cos(q) - cos(q)cos(q)cos(q))sin(q) + sin(q
 (-sin(q)cos(q) + sin(q)cos(q))sin(q)sin(q)

)cos(q))sin(q)   -(((sin(q)sin(q)sin(q) + sin(q)cos(q)cos(q))co
)cos(q))sin(q)  -(((-sin(q)sin(q)cos(q) - cos(q)cos(q)cos(q))c
                                                          -((-sin(q)s

s(q) + sin(q)cos(q))cos(q) + (-sin(q)sin(q)cos(q) + sin(q)sin(q)
os(q) + sin(q)sin(q))cos(q) + (sin(q)cos(q)cos(q) - sin(q)cos(q)
in(q) - cos(q)cos(q))sin(q) + (-sin(q)cos(q) + sin(q)cos(q))cos(q
                                                             0
cos(q))sin(q))sin(q) + (-(sin(q)sin(q)sin(q) + sin(q)cos(q)cos(
cos(q))sin(q))sin(q) + (-(-sin(q)sin(q)cos(q) - cos(q)cos(q)cos
)cos(q))sin(q) - (-sin(q)cos(q) + sin(q)cos(q))sin(q)cos(q)

q))sin(q) + cos(q)cos(q))cos(q)   ((sin(q)sin(q)sin(q) + sin(q)
(q))sin(q) + sin(q)cos(q))cos(q)  ((-sin(q)sin(q)cos(q) - cos(q)
                                                       -(-sin(q)
```

```
cos(q)cos(q))cos(q) + sin(q)cos(q))sin(q) - (-sin(q)sin(q)cos(q

cos(q)cos(q))cos(q) + sin(q)sin(q))sin(q) - (sin(q)cos(q)cos(q

sin(q) - cos(q)cos(q))cos(q) + (-sin(q)cos(q) + sin(q)cos(q))sin(

                                    0

) + sin(q)sin(q)cos(q))cos(q)  0.11345((sin(q)sin(q)sin(q) + sin(

) - sin(q)cos(q)cos(q))cos(q)  0.11345((-sin(q)sin(q)cos(q) - cos

q)cos(q)                                                          -0.11345


q)cos(q)cos(q))cos(q) + sin(q)cos(q))sin(q) - 0.11345(-sin(q)si

(q)cos(q)cos(q))cos(q) + sin(q)sin(q))sin(q) - 0.11345(sin(q)co

(-sin(q)sin(q) - cos(q)cos(q))cos(q) + 0.11345(-sin(q)cos(q) + si


n(q)cos(q) + sin(q)sin(q)cos(q))cos(q) + 0.37092sin(q)sin(q)cos

s(q)cos(q) - sin(q)cos(q)cos(q))cos(q) - 0.37092sin(q)cos(q)cos

n(q)cos(q))sin(q)cos(q) + 0.37092sin(q)sin(q) + 0.37092cos(q)cos

          1

(q) + 0.4209sin(q)sin(q) - 0.37092sin(q)sin(q)cos(q)

(q) - 0.4209sin(q)cos(q) + 0.37092sin(q)cos(q)cos(q)

(q) + 0.4209cos(q) + 0.32871


**************************************************************************
                          Jacobian Matrix (J):
**************************************************************************

-0.11345sin(q)sin(q)sin(q) + 0.4209sin(q)cos(q) + 0.11345sin(q)c
```

```
0.4209sin(q)sin(q) + 0.11345sin(q)sin(q)cos(q)cos(q - q) + 0.113
```

```
os(q)cos(q)cos(q - q) + 0.11345sin(q - q)cos(q)cos(q) + 0.37092s
```

```
45sin(q)sin(q - q)cos(q) + 0.37092sin(q)sin(q - q) + 0.11345sin(q
```

```
         0
         0
         0
         1
```

```
in(q - q)cos(q)  (-0.11345sin(q)sin(q - q)cos(q) + 0.4209cos(q) +
```

```
)sin(q)cos(q)   (0.11345sin(q)sin(q - q)cos(q) - 0.4209cos(q) -
                          -0.4209sin(q) - 0.11345sin(q)cos(q)cos(q - q
                                                                      co
                                                                      si
```

```
 0.11345cos(q)cos(q - q) + 0.37092cos(q - q))sin(q)  (0.11345sin(q
0.11345cos(q)cos(q - q) - 0.37092cos(q - q))cos(q)   (-0.11345sin(q
) - 0.11345sin(q - q)cos(q) - 0.37092sin(q - q)            0.11345si
s(q)
n(q)
 0
)sin(q - q)cos(q) - 0.11345cos(q)cos(q - q) - 0.37092cos(q - q))
```

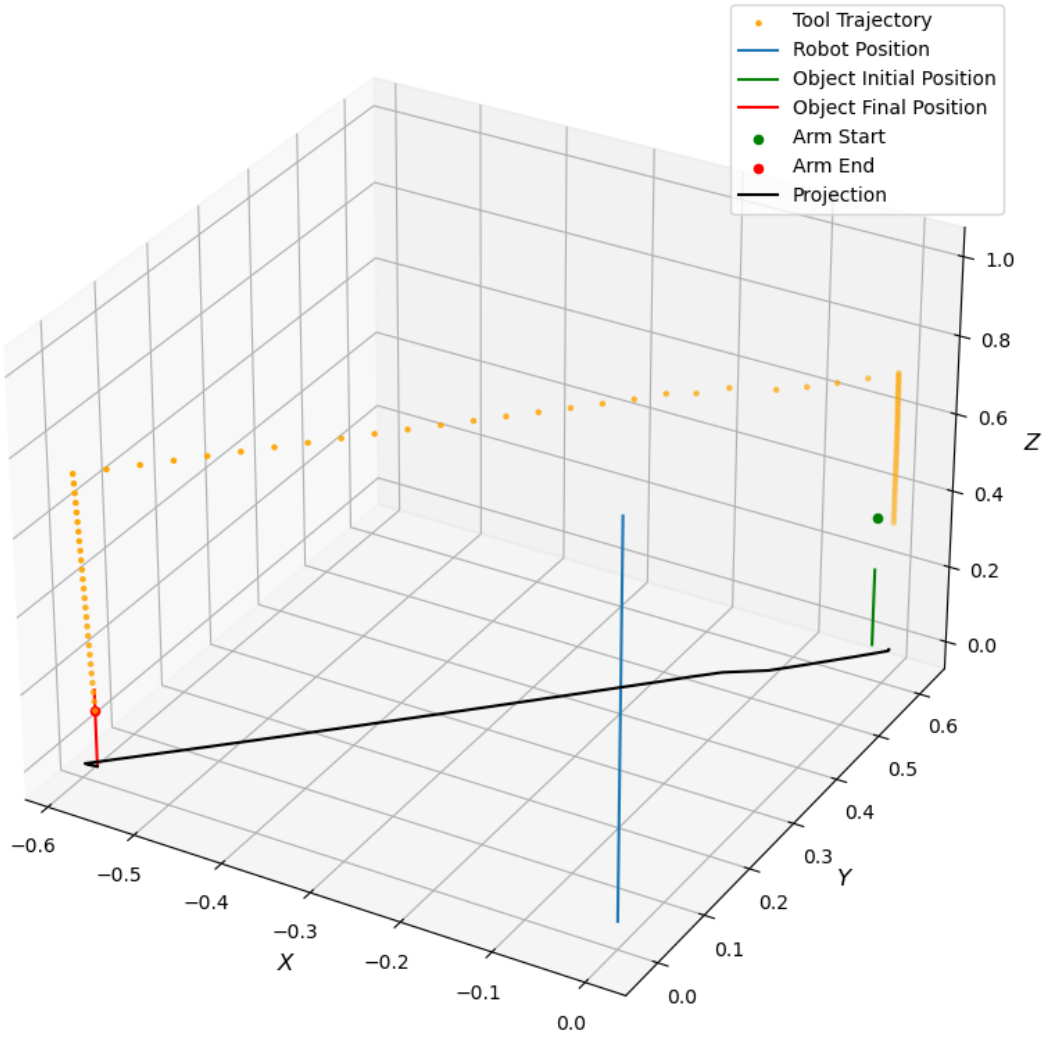
```
)sin(q - q)cos(q) + 0.11345cos(q)cos(q - q) + 0.37092cos(q - q))

n(q)cos(q)cos(q - q) + 0.11345sin(q - q)cos(q) + 0.37092sin(q - q

                            sin(q)sin(q)

                           -sin(q)cos(q)

                               cos(q)

sin(q)   -0.11345(sin(q)sin(q)cos(q - q) - cos(q)cos(q))sin(q)  0

cos(q)  0.11345(sin(q)cos(q) + sin(q)cos(q)cos(q - q))sin(q)   0

)                      0.11345sin(q)sin(q)sin(q - q)

                               sin(q)sin(q - q)

                               -sin(q - q)cos(q)

                                   cos(q - q)

.11345(sin(q)cos(q)cos(q - q) + sin(q)cos(q))cos(q) - 0.11345sin(

.11345(sin(q)sin(q) - cos(q)cos(q)cos(q - q))cos(q) + 0.11345sin(

                -0.11345sin(q)cos(q - q) - 0.11345sin(q - q)cos(q)c

                           -sin(q)sin(q)cos(q - q) + cos(q)cos(q)

                           sin(q)cos(q) + sin(q)cos(q)cos(q - q)

                                        sin(q)sin(q - q)

q)sin(q)sin(q - q)                                              0

q)sin(q - q)cos(q)                                              0

os(q)                                                               0

                          (sin(q)cos(q)cos(q - q) + sin(q)cos(q))sin

                          (sin(q)sin(q) - cos(q)cos(q)cos(q - q))sin

                                       -sin(q)sin(q - q)cos(q) + cos
```

```
(q) + sin(q)sin(q - q)cos(q)

(q) - sin(q - q)cos(q)cos(q)

(q)cos(q - q)
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
```

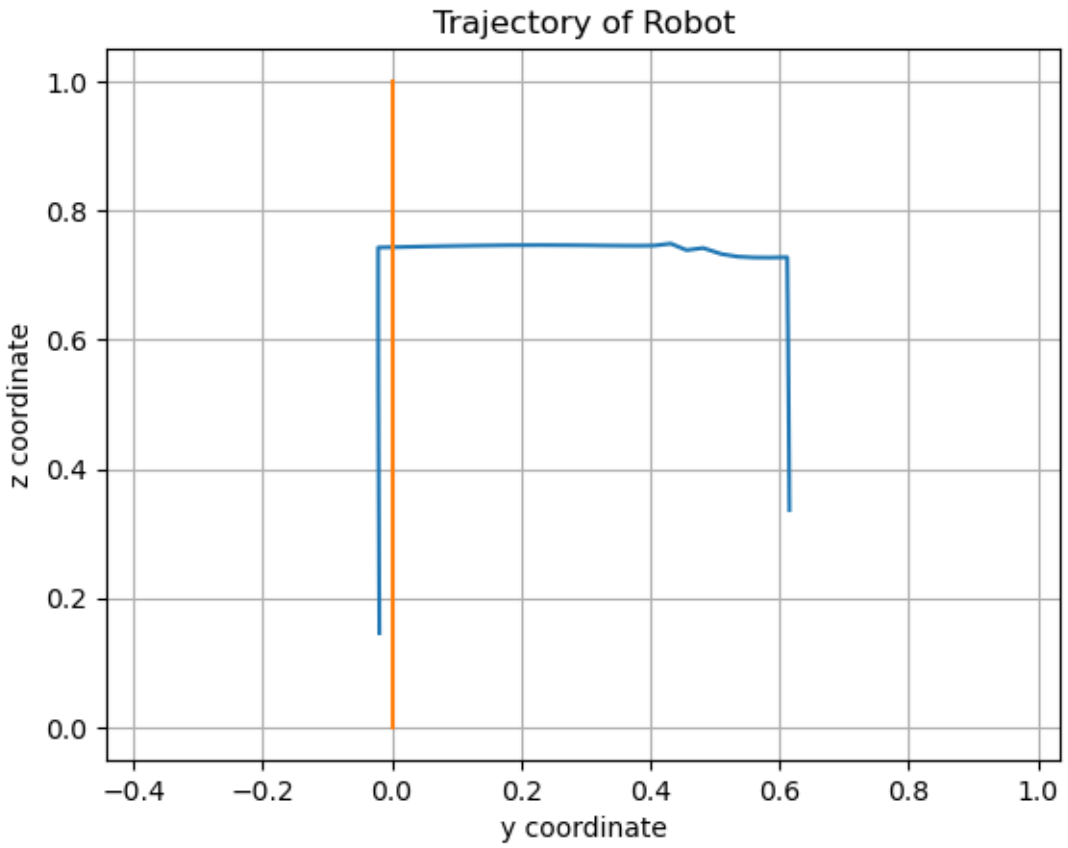
```
87
88
89
90
91
92
93
94
95
96
97
98
99
```

```
<Figure size 640x480 with 0 Axes>
```

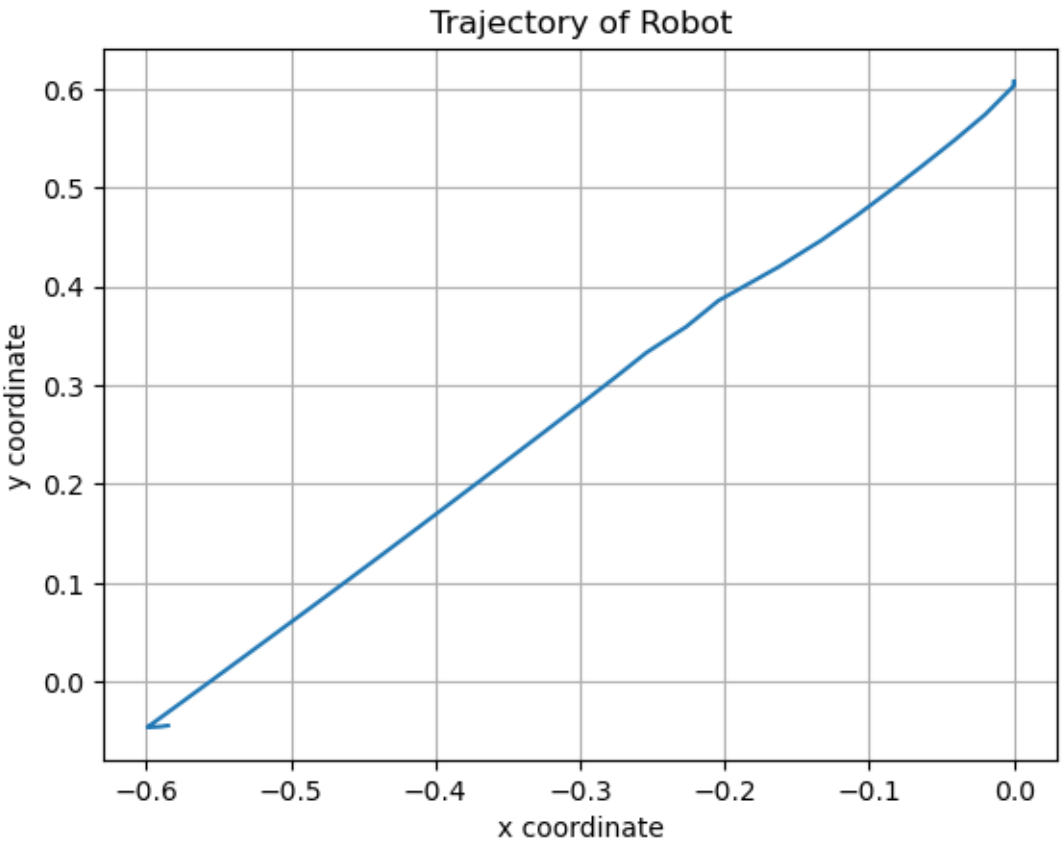
[67]:
```
import numpy as np
ax = plt.figure(figsize=(10,10)).add_subplot(projection='3d')
# ax.axes.set_xlim3d(left=65, right=70)
# ax.axes.set_ylim3d(bottom=-15, top=15)
# ax.axes.set_zlim3d(bottom=62.5, top=82.5)
ax.set_xlabel('$X$', fontsize=12)
ax.set_ylabel('$Y$', fontsize=12)
ax.set_zlabel('$Z$', fontsize=12)
# ax.set_zticks(np.linspace(62.5,82.5,5))
# ax.set_xticks(np.linspace(65,70,5))
# ax.set_yticks(np.linspace(-15,15,5))
ax.scatter(x_tool, y_tool, z_tool,marker='.',color='orange', label='Tool␣
↪Trajectory')
ax.plot(np.full(len(x_tool),0.0), np.full(len(x_tool),0.0), np.
↪linspace(0,1,len(x_tool)), label='Robot Position')
ax.plot(np.full(10,-0.0188), np.full(10,0.6155), np.linspace(0,0.
↪2,10),color='green', label='Object Initial Position')
ax.plot(np.full(10,-0.5702), np.full(10,-0.0193), np.linspace(0,0.2,10),␣
↪color='red',label='Object Final Position')
ax.scatter(np.full(1,-0.0188), np.full(1,0.6155), np.full(1,0.
↪3362),marker='o',color='green', label='Arm Start')
ax.scatter(np.full(1,-0.5702), np.full(1,-0.0193), np.full(1,0.
↪1457),marker='o',color='red', label='Arm End')
ax.plot(x_tool, y_tool, np.full(len(y_tool),0.0),color='black',␣
↪label='Projection')

ax.legend()

plt.show()
```



[62]:
```
import matplotlib.pyplot as plt
plt.plot(y_tool,z_tool)
plt.plot(np.full(len(x_tool),0.0), np.linspace(0,1,len(x_tool)))
# plt.scatter(0,72.5)
plt.xlabel("y coordinate")
plt.ylabel("z coordinate")
plt.axis("equal")
plt.title("Trajectory of Robot")
plt.grid(True)
plt.show()
```



[36]:
```
import matplotlib.pyplot as plt
plt.plot(x_tool,y_tool)
# plt.scatter(0,72.5)
plt.xlabel("x coordinate")
plt.ylabel("y coordinate")
# plt.axis("equal")
plt.title("Trajectory of Robot")
plt.grid(True)
plt.show()
```


Trajectory of Robot
y coordinate
x coordinate
0.6
0.5
0.4
0.3
0.2
0.1
0.0
−0.6
−0.5
−0.4
−0.3
−0.2
−0.1
0.0

[ ]: