Co-funded by the
Erasmus+ Programme
of the European Union

Erasmus Mundus Joint Master Degree Programme

in Sustainable Ship and Shipping 4.0

University of Naples Federico II

Robotics Technical Elaborate Report

Presented by

Bakel Bakel Begededum

Submitted to

Prof. Vincenzo Lippiello

Contents

List of Figures

Chapter 1

Kinematics

Given the SCARA robot manipulator below,

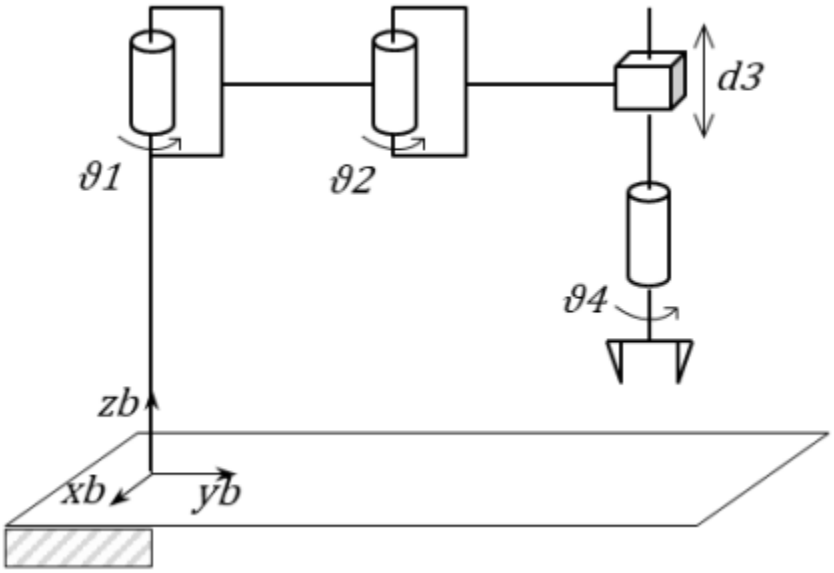

Figure 1.1: Diagram of the Question

I aim to determine:

1. The solution to the forward kinematic problem
2. The solution to the inverse kinematic problem

1 Solutions to Kinematic Problems

1.1 Forward Kinematic Problem

The forward kinematics problem of a robotic system can be addressed through various methodologies and conventions. In this course, the problem was solved with the use of the ***homogenous transformation matrix***;

$$T_{\mathrm{e}}^{\mathrm{b}}(\mathbf{q}) = \begin{bmatrix} \mathbf{n}_{\mathrm{e}}^{\mathrm{b}}(\mathbf{q}) & \mathbf{s}_{\mathrm{e}}^{\mathrm{b}}(\mathbf{q}) & \mathbf{a}_{\mathrm{e}}^{\mathrm{b}}(\mathbf{q}) & \mathbf{p}_{\mathrm{e}}^{\mathrm{b}}(\mathbf{q}) \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1.1}$$

and the ***Denavit–Hartenberg (D-H) Convention***, both of which provide systematic frameworks for determining the position and orientation of a robot's end-effector. For this project, I applied all the methods covered in class and extended the scope by incorporating an additional method known as ***Screw Theory***, to analyze and solve the forward kinematics.

By utilizing these methods, the forward kinematic analysis benefits from both systematic parameterization (Homogeneous Matrix and D-H Convention) and advanced geometric interpretation (Screw Theory). This comprehensive approach ensures robustness and flexibility in solving complex robotic configurations.

1.1.1 Homogeneous Transformation Matrix

This method involves the representation of spatial transformations (rotation and translation) in a unified 4x4 matrix format. Each link and joint of the robot is described by a sequence of transformations, which, when combined, determine the final position and orientation of the end-effector in the base frame.

Key Features:

1. Encodes rotation using a 3x3 rotation matrix.
2. Encodes translation as a 3x1 vector.
3. Allows easy composition of transformations using matrix multiplication.

Solution using Homogeneous Transformation Matrix

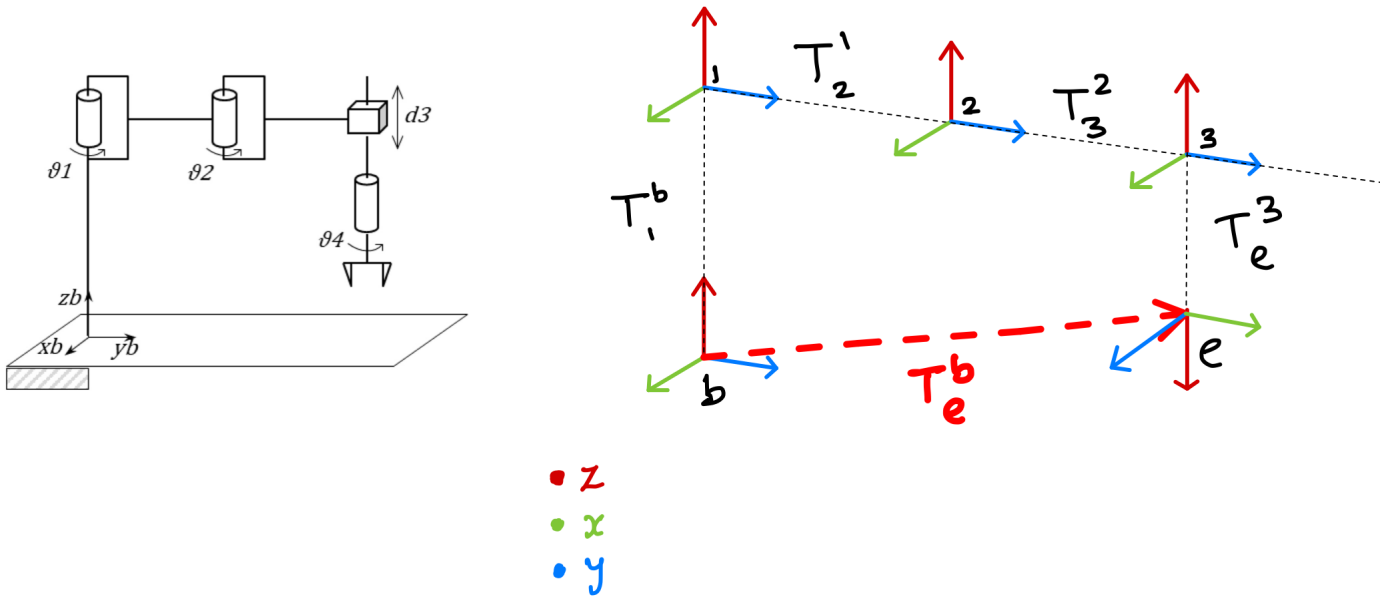

Figure 1.2: Diagram of Frames of the SCARA Robot

Equation 1.2 gives the formula for the forward kinematics of the scara robot as obtained from Figure 2

$$T_e^b = T_1^b \cdot T_2^1 \cdot T_3^2 \cdot T_e^3 \tag{1.2}$$

$$T_1^b = \begin{bmatrix} \cos\theta_1 & -\sin\theta_1 & 0 & 0 \\ \sin\theta_1 & \cos\theta_1 & 0 & 0 \\ 0 & 0 & 1 & d_0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1.3}$$

$$T_2^1 = \begin{bmatrix} \cos\theta_2 & -\sin\theta_2 & 0 & 0 \\ \sin\theta_2 & \cos\theta_2 & 0 & a_1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1.4}$$

$$T_3^2 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & a_2 \\ 0 & 0 & 1 & d_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1.5}$$

$$T_e^3 = \begin{bmatrix} \cos\theta_4 & -\sin\theta_4 & 0 & 0 \\ \sin\theta_4 & \cos\theta_4 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1.6}$$

Finally, the forward kinematics can be determined by

$$T_e^b = T_1^b \cdot T_2^1 \cdot T_3^2 \cdot T_e^3 \tag{1.7}$$

Solve this by hand? As an engineer, I will do no such! I love mathematical rigour but I am an efficient engineer. I have decided to flex the power of python to solve symbolically the said matrics equation. The **Sympy** Library was used for this task.

Multiplication code.py ● | Workspace Trust

C: > Users > beged > OneDrive > Documents > ERASMUS MUNDUS > SEAS4.0 > Academic > Main course Work > Robotics > Bakel_Bakel_Project > Multiplication code.py >

```python
# Importing the init_printing function from the sympy library
# This is used to enable better printing of symbolic expressions, making them easier to read.
from sympy import init_printing

# Initializing pretty printing with Unicode characters for enhanced readability
# This ensures that the symbolic matrix equations and expressions appear in a clean,
# mathematical format.
init_printing(use_unicode=True)
from IPython.display import display

# Importing the symbols function from sympy
# The symbols function allows the creation of symbolic variables that can be used in equations.
# The cos and sin functions are used for symbolic trigonometric calculations.
from sympy import symbols,cos, sin, pprint, simplify

# Importing the Matrix class from sympy's matrices module
# This class is used to create and manipulate matrices
from sympy.matrices import Matrix
import math

#Define all the symbols needed
theta1,theta2,theta4,do,a1,a2,d3 = symbols ("theta1,theta2,theta4,do,a1,a2,d3")

```

Multiplication code.py ● | Workspace Trust

C: > Users > beged > OneDrive > Documents > ERASMUS MUNDUS > SEAS4.0 > Academic > Main course Work > Robotics > Bakel_Bakel_Project > Multiplication code.py > ...

```python
#Define all the transformation matrix
Tb_1 = Matrix([[cos(theta1),-sin(theta1), 0 , 0 ],
               [sin(theta1), cos(theta1), 0 , 0 ],
               [     0     ,     0      , 1 , do],
               [     0     ,     0      , 0 , 1 ]])

T1_2 = Matrix([[cos(theta2),-sin(theta2), 0 , 0 ],
               [sin(theta2), cos(theta2), 0 , a1],
               [     0     ,     0      , 1 , 0 ],
               [     0     ,     0      , 0 , 1 ]])

T2_3 = Matrix([[     1     ,     0      , 0 , 0 ],
               [     0     ,     1      , 0 , a2],
               [     0     ,     0      , 1 , d3],
               [     0     ,     0      , 0 , 1 ]])

T3_e = Matrix([[cos(theta4),-sin(theta4), 0 , 0 ],
               [sin(theta4), cos(theta4), 0 , 0 ],
               [     0     ,     0      , -1 , 0 ],
               [     0     ,     0      , 0 , 1 ]])

#Multiplying all the transformation matrix to get the final matrix
Tb_e = Tb_1 * T1_2 * T2_3 * T3_e

# Simplifying each component of the resulting transformation matrix
Tb_e_simplified = simplify(Tb_e)

# Displaying the simplified transformation matrix
pprint(Tb_e_simplified)

pprint(simplify(Tb_1 * T1_2))
pprint(simplify(Tb_1 * T1_2* T2_3))
pprint(simplify(Tb_1 * T1_2* T2_3* T3_e))
```

Upon running the code, the following matrix was gotten. This is the solution to our forward kinematic problem.

$$\begin{bmatrix} \cos(\theta_1 + \theta_2 + \theta_4) & -\sin(\theta_1 + \theta_2 + \theta_4) & 0 & -a_1 \cdot \sin(\theta_1) - a_2 \cdot \sin(\theta_1 + \theta_2) \\ \sin(\theta_1 + \theta_2 + \theta_4) & \cos(\theta_1 + \theta_2 + \theta_4) & 0 & a_1 \cdot \cos(\theta_1) + a_2 \cdot \cos(\theta_1 + \theta_2) \\ 0 & 0 & -1 & d_3 + do \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

VOILAA!!!

1.1.2 Denavit–Hartenberg (D-H) Convention

This is a standardized convention for modeling the forward kinematics of serial-chain robots. It simplifies the process by defining each joint's coordinate frame through four parameters: link length, link twist, link offset, and joint angle.

Key Features:

i. Reduces the complexity of defining coordinate systems.

ii. Provides a compact parameterization of the robot's kinematic chain.

iii. Supports systematic derivation of transformation matrices for each joint.

a. Firstly, number the joints/points of reference where the frames will be located.

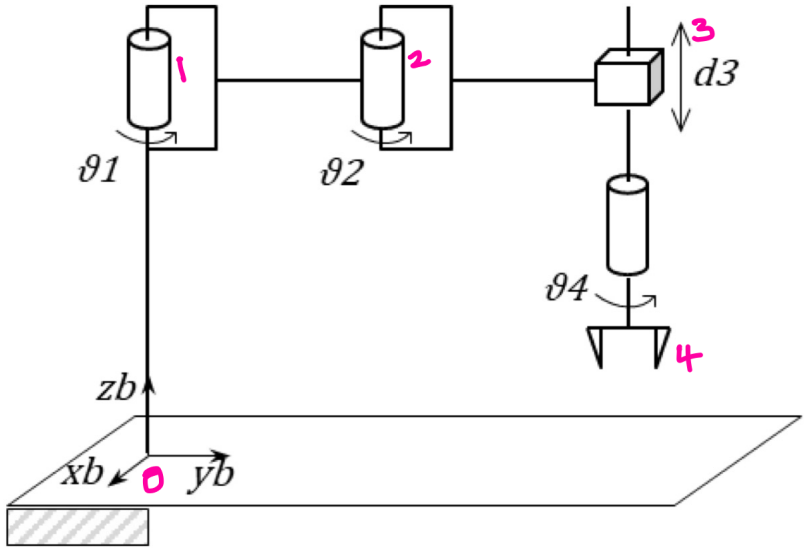

Figure 1.3: Definition of the frames

b. The joint 1 is revolute and following DH Convention, the Z axis will be given as in the image below. The x and y axis of Frame 1 can be obtained correctly as given in the image below.

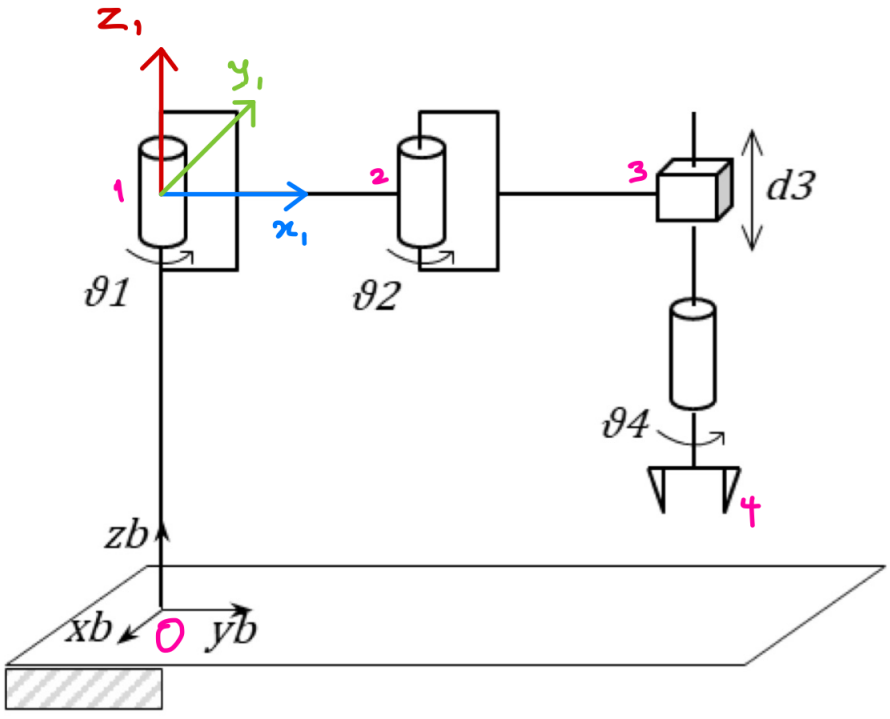

Figure 1.4: Determination of Frame 1

c. According to the Denavit-Hartenberg (DH) convention, it is often convenient to align the

base frame with Frame 1 for simplicity in deriving the transformation matrices. However, in the context of this problem, the base frame is explicitly defined as part of the problem statement and is distinct from Frame 1.

While I could alter the base frame to match Frame 1 in my solution, this approach is not advisable for practical reasons, especially in an industrial environment:

i. Adherence to Given Specifications: In real-world engineering problems, the coordinate frames are often pre-defined by the system, client requirements, or the physical environment. Modifying the base frame for convenience could lead to misinterpretations or deviations from the intended design.

ii. Compatibility with Other Systems: Industrial setups often involve integration with existing systems or machines. The base frame might be used as a reference for other components, sensors, or control systems. Changing it in calculations could lead to inconsistencies or errors when interpreting results in the actual system.

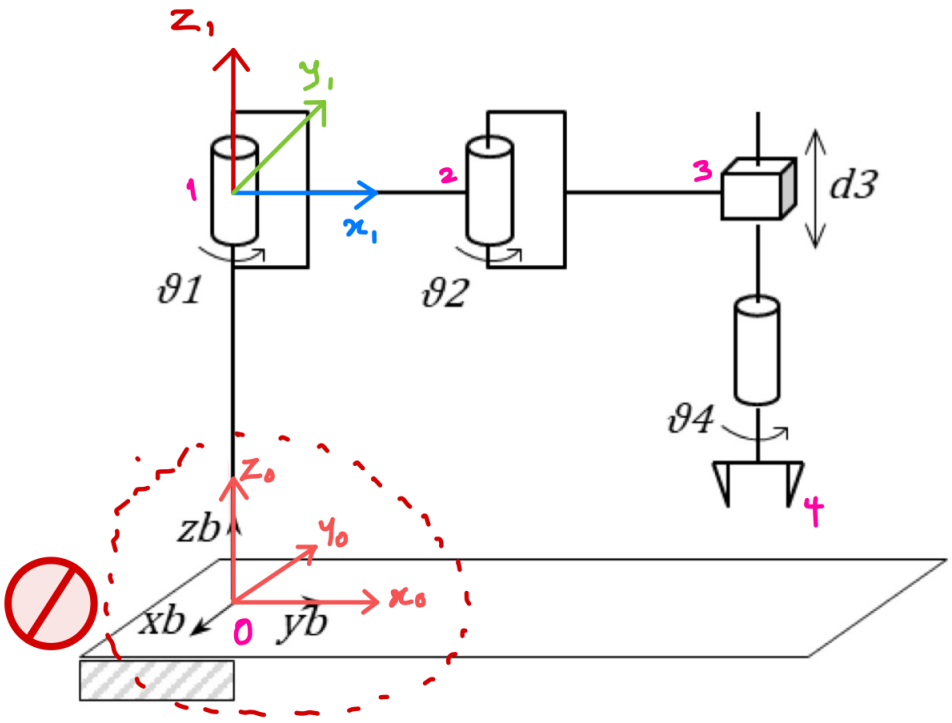

Figure 1.5: Illustration of Retaining the Given Base Frame for Consistency with Problem Specifications.

d. The joint 2 is also revolute so the axis is determined as follows;

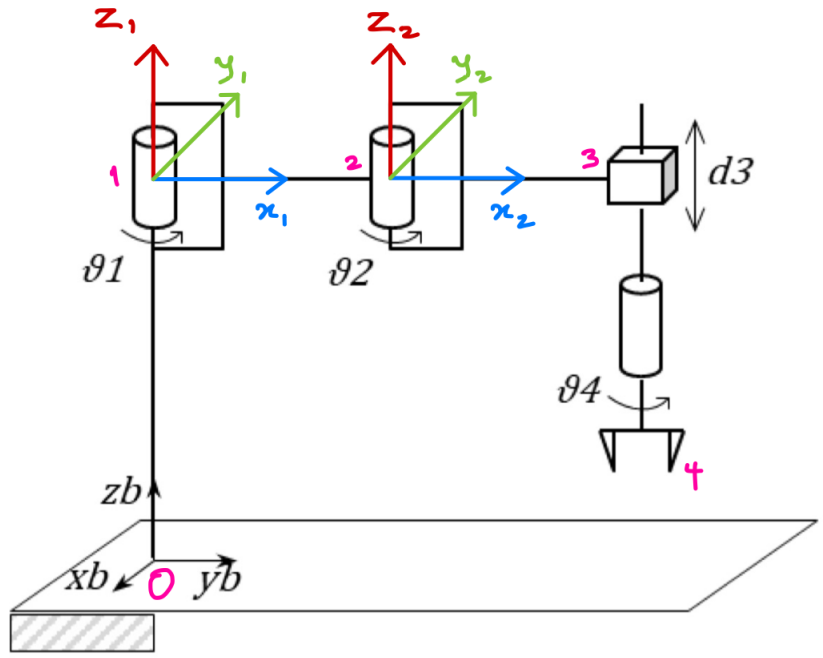

Figure 1.6: Determination of Frame 2

e. The joint 3 however is a prismatic so the axis is determined as follows;

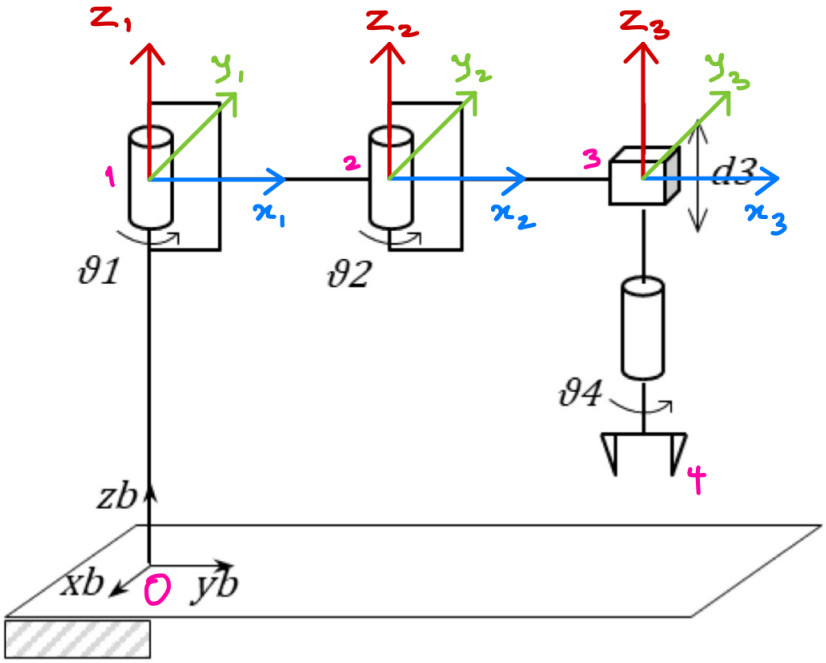

Figure 1.7: Determination of Frame 3

e. For joint 4, there are some school of thought that insist on putting the frame at the center of the revolute joint and still have a separate frame for the end effector. However, with no clear distance between the joint and the end effector, there is no point in having two separate frames for this configuration and the frame of the revolute joint and the end effector will not only be identical but coincidental.

Secondly, from the nature of the question and the position of the end effector, it is intuitive to assign the (z_4) facing downwards. But since this doesnt affect much aside our perception of the end effector (which we can easily interpret), I will follow the DH convention that advices the end effector frame to be identical to that of the last frame (Frame 3).

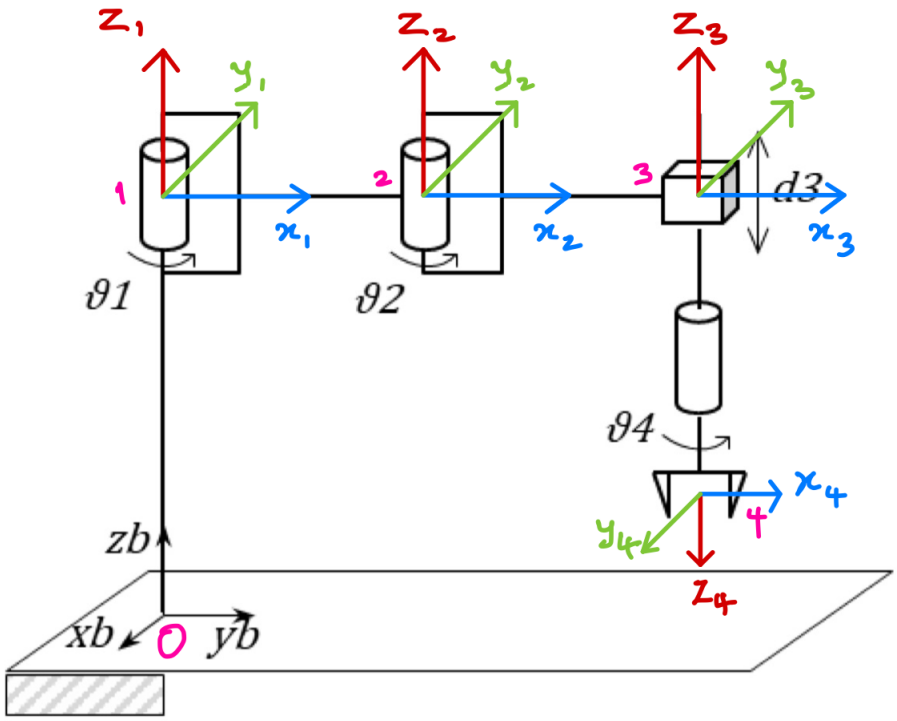

Figure 1.8: Schematics showing all the DH Frames of the SCARA robot

Table 1.1: Table showing DH Parameters

Link	a_i	α_i	d_i	θ_i
1	0_1	0	d_0	$\frac{\pi}{2} + \theta_1$
2	a_1	0	0	θ_2
3	a_2	0	0	0
4	0	π	d_3	θ_4

1.1.3 Screw Theory

Screw Theory offers a geometric and algebraic framework to describe motion and kinematics in terms of twists (velocity screws) and wrenches (force screws). It represents the motion of a rigid

body as a combination of rotational and translational components about a screw axis.

Key Features:

i. Models motion using Plücker coordinates for lines.

ii. Efficiently handles instantaneous kinematics, including singularities and constraints.

iii. Extends naturally to spatial motion analysis, offering insights into both the kinematic and dynamic aspects of robotic systems.

To find the transformation matrix using screw theory for the SCARA robot, we need to use the following steps:

A screw motion is characterized by: 1. A rotation about an axis (with angular velocity ω).
2. A translation along the same axis (with linear velocity v).

The transformation matrix for a rigid body moving under a screw motion is given by:

$$T = e^{\hat{\xi}\theta} \tag{1.8}$$

where: $\hat{\xi}$ is the twist matrix, defined as:

$$\hat{\xi} = \begin{bmatrix} \hat{\omega} & v \\ 0 & 0 \end{bmatrix} \tag{1.9}$$

$\hat{\omega}$ is the skew-symmetric matrix for angular velocity ω.
v is the linear velocity vector.
θ is the joint variable (angle or displacement).

Step-by-Step Process

The SCARA robot has the following joints:
1. Revolute joint 1 (θ_1): Rotation about the Z-axis at the base.
2. Revolute joint 2 (θ_2): Rotation about the Z-axis at the second joint.
3. Prismatic joint (d_3): Linear motion along the Z-axis.
4. Revolute joint 4 (θ_4): Rotation about the Z-axis at the end-effector.

Define the Twists (ξ)
For each joint, we define the twist ξ as:

$$\xi = \begin{bmatrix} \omega \\ v \end{bmatrix}$$

Joint 1 (θ_1):

$$\omega_1 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \quad v_1 = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \quad \Rightarrow \quad \xi_1 = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{1.10}$$

Joint 2 (θ_2):

$$\omega_2 = \begin{bmatrix}0\\0\\1\end{bmatrix}, \quad v_2 = \begin{bmatrix}-a_1\\0\\0\end{bmatrix} \quad \Rightarrow \quad \xi_2 = \begin{bmatrix}0\\0\\1\\-a_1\\0\\0\end{bmatrix} \tag{1.11}$$

Joint 3 (d_3):

$$\omega_3 = \begin{bmatrix}0\\0\\0\end{bmatrix}, \quad v_3 = \begin{bmatrix}0\\0\\1\end{bmatrix} \quad \Rightarrow \quad \xi_3 = \begin{bmatrix}0\\0\\0\\0\\0\\1\end{bmatrix} \tag{1.12}$$

Joint 4 (θ_4):

$$\omega_4 = \begin{bmatrix}0\\0\\1\end{bmatrix}, \quad v_4 = \begin{bmatrix}-a_1-a_2\\0\\0\end{bmatrix} \quad \Rightarrow \quad \xi_4 = \begin{bmatrix}0\\0\\1\\-a_1-a_2\\0\\0\end{bmatrix} \tag{1.13}$$

3. Compute the Exponential Map For each joint, the transformation matrix is given by:

$$T_i = e^{\hat{\xi}_i \theta_i} \tag{1.14}$$

The twist matrix $\hat{\xi}$ is:

$$\hat{\xi} = \begin{bmatrix}\hat{\omega} & v\\0 & 0\end{bmatrix} \tag{1.15}$$

For revolute joints ($\omega \neq 0$):

$$\hat{\omega} = \begin{bmatrix}0 & -\omega_z & \omega_y\\ \omega_z & 0 & -\omega_x\\ -\omega_y & \omega_x & 0\end{bmatrix} \tag{1.16}$$

For prismatic joints ($\omega = 0$):

$$\hat{\xi} = \begin{bmatrix}0 & 0 & 0 & v_x\\0 & 0 & 0 & v_y\\0 & 0 & 0 & v_z\\0 & 0 & 0 & 0\end{bmatrix} \tag{1.17}$$

4. Final Transformation Matrix The total transformation matrix T_b^e is:

$$T_b^e = T_1 T_2 T_3 T_4 \tag{1.18}$$

where:

$$T_1 = e^{\hat{\xi}_1 \theta_1} \tag{1.19}$$

$$T_2 = e^{\hat{\xi}_2 \theta_2} \tag{1.20}$$

$$T_3 = e^{\hat{\xi}_3 d_3} \tag{1.21}$$

$$T_4 = e^{\hat{\xi}_4 \theta_4} \tag{1.22}$$

1.1.4 Verification with Corke's Robotics Toolbox

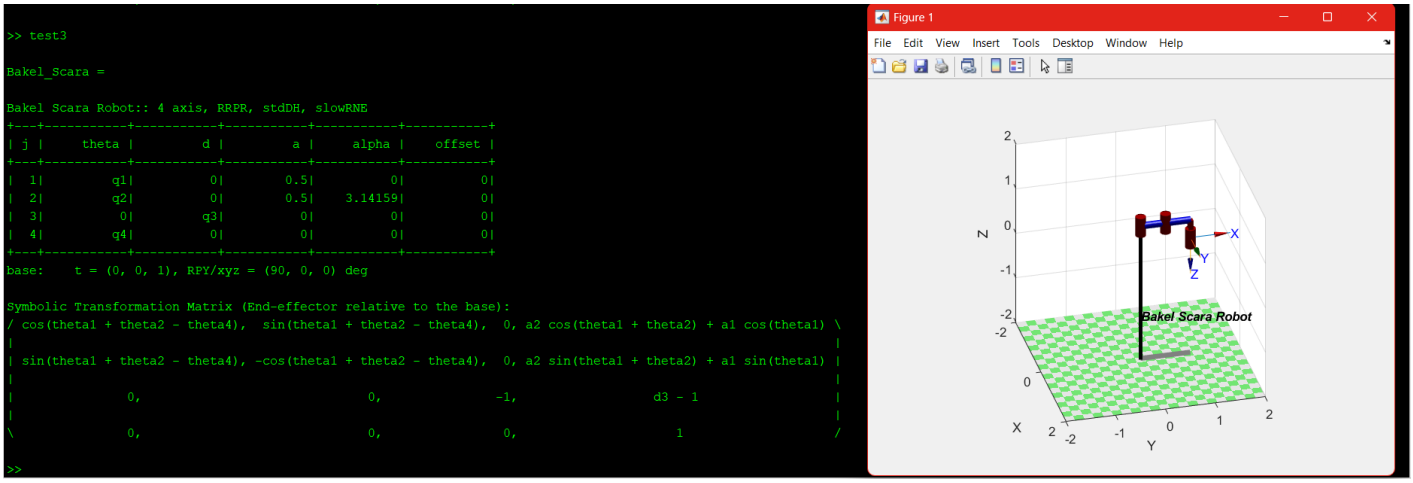

Figure 1.9: Schematics showing all the DH Frames of the SCARA robot

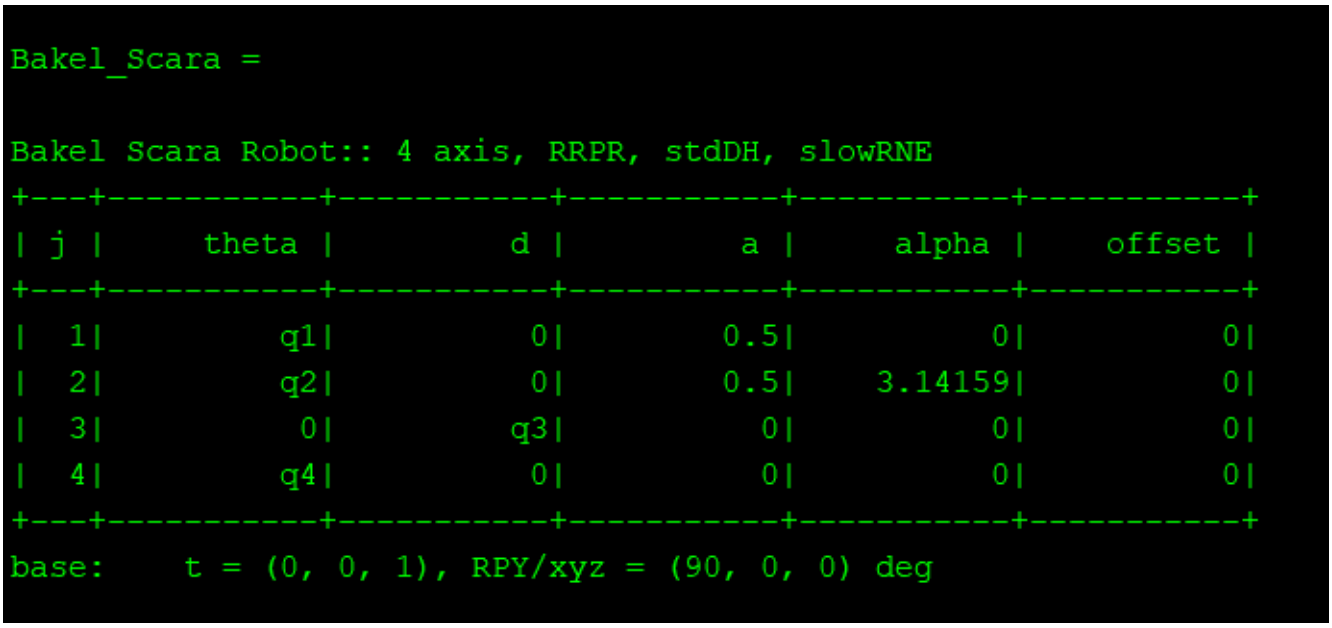

Figure 1.10: Schematics showing DH table of the SCARA robot using joints

```
Symbolic Transformation Matrix (End-effector relative to the base):
/ cos(theta1 + theta2 - theta4),  sin(theta1 + theta2 - theta4),  0, a2 cos(theta1 + theta2) + a1 cos(theta1) \
|                                                                                                            |
| sin(theta1 + theta2 - theta4), -cos(theta1 + theta2 - theta4),  0, a2 sin(theta1 + theta2) + a1 sin(theta1) |
|                                                                                                            |
|              0,                              0,              -1,                 d3 - 1                  |
|                                                                                                            |
\              0,                              0,               0,                   1                    /
```

Figure 1.11: The Transformation Matrix

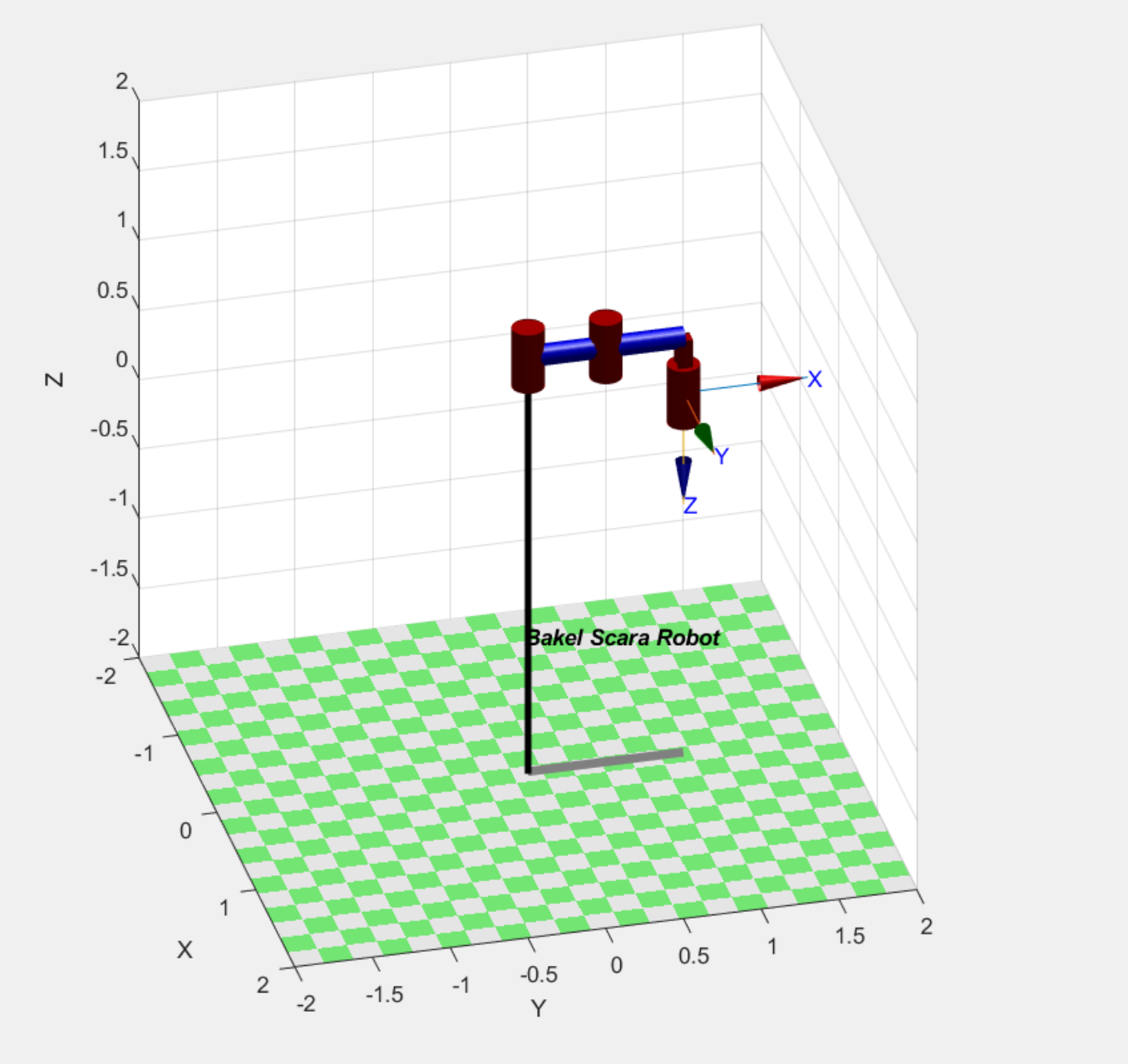

Figure 1.12: Model of the SCARA robot

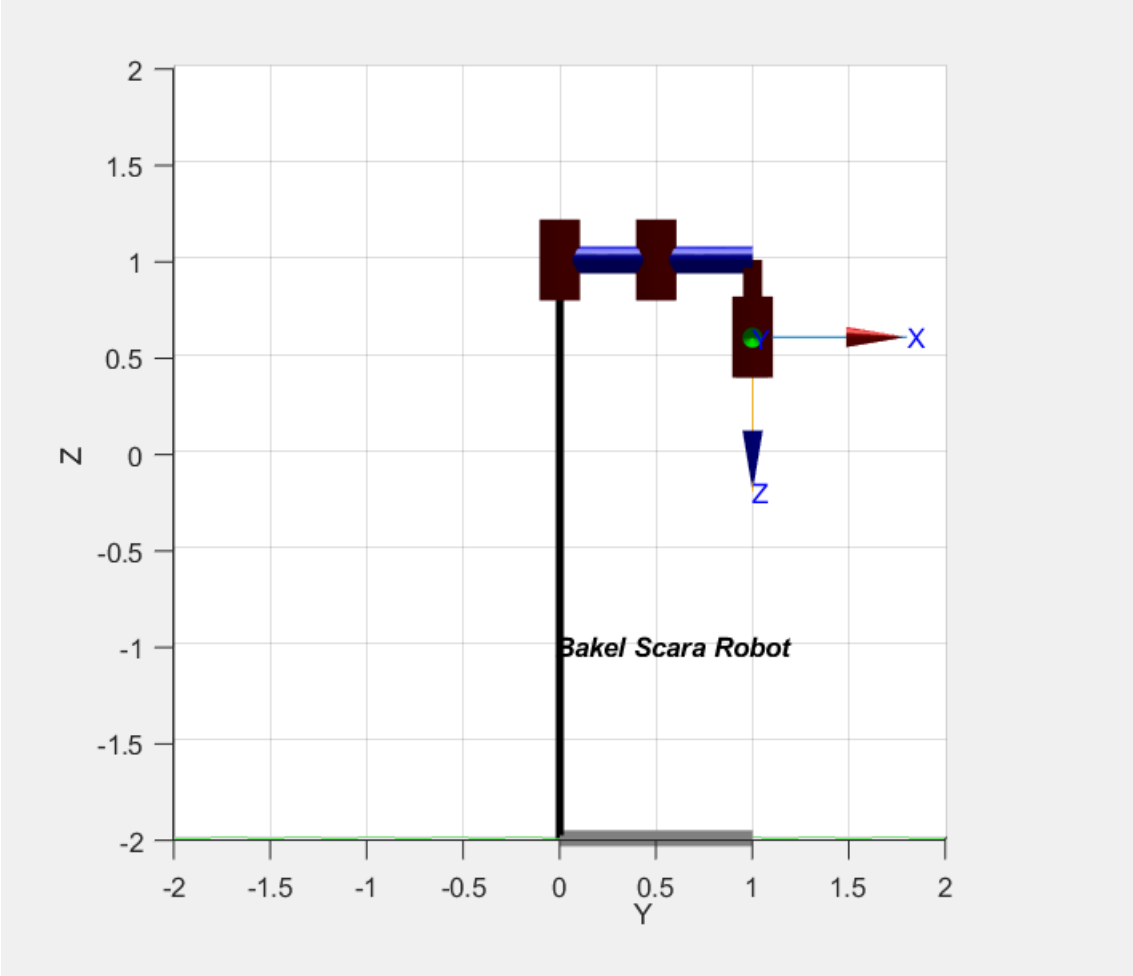

Figure 1.13: Model of the SCARA robot in ZY Frame

1.2 Inverse Kinematic Problem

The inverse kinematic problem is formulated as follows:

Given the position and orientation of the end effector, find the joint variables of the robot.

The given SCARA robot has 4 DOF, so for its position and orientation to be fully represented, we need 4 variables which are; x, y, z for position, and ψ for orientation

$$\psi = \theta_1 + \theta_2 + \theta_3 \tag{1.23}$$

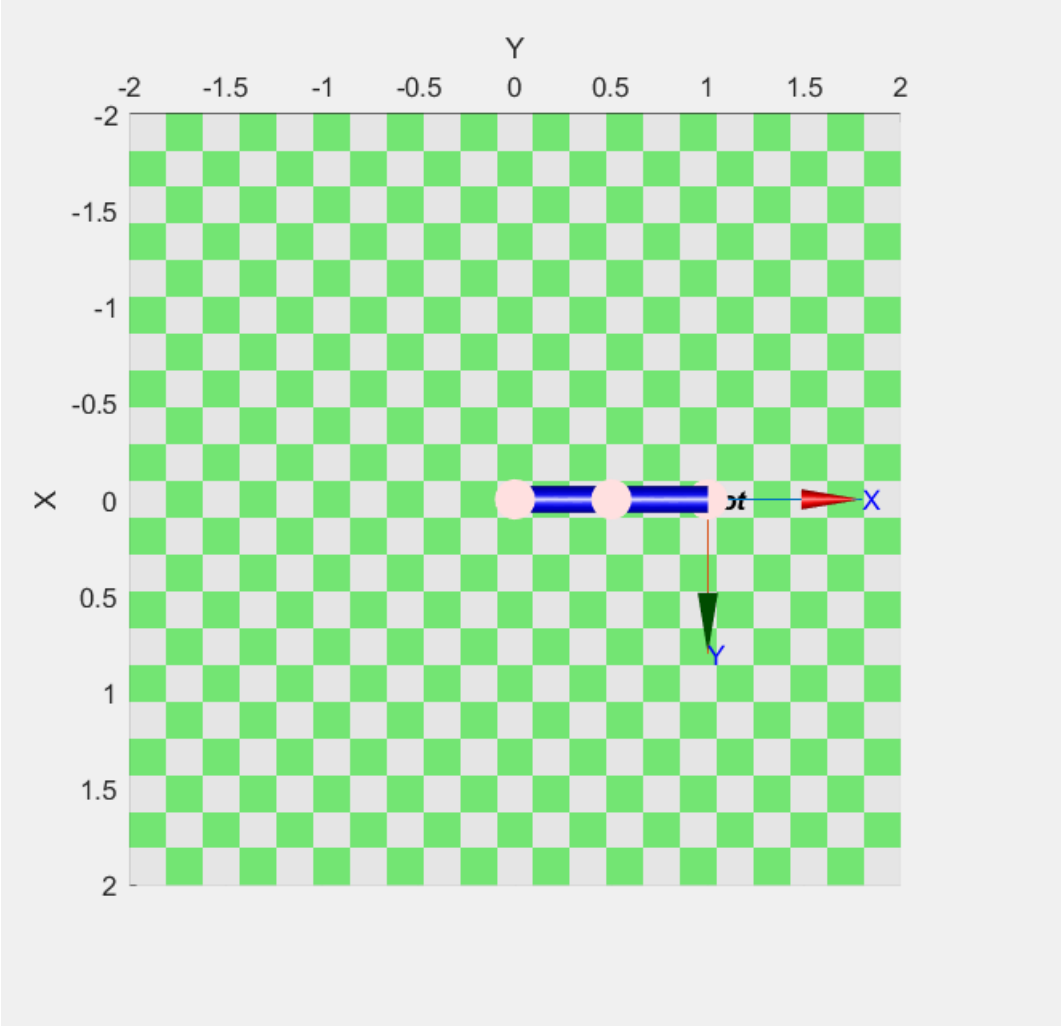

Figure 1.14: Top View (XY Plane) of the robot

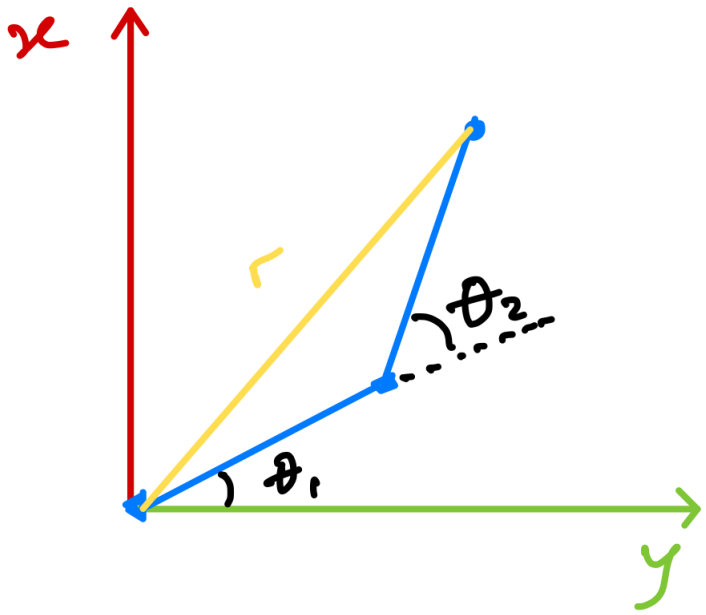

Figure 1.15: Top View of the SCARA robot to solve inverse kinematics

$$
\begin{aligned}
&\text{As } z_2 \text{ is along } z_4 \\
&p_2\,(x_2, y_2) \geq (x_1) \\
&r^2 = x^2 + y^2 \\
&r^2 = a_1^2 + a_2^2 - 2a_1 a_2 \cos x \\
&1\theta_2 = \pi - \alpha \\
&\cos\theta_2 = -\cos\alpha \\
&r^2 = a_1^2 + a_2^2 + 2a_1 a_2 \cos\theta_2 \\
&\sin^2\theta_2 + \cos^2\theta_2 = 1 \\
&\sin\theta_2 = \sqrt{1 - \cos^2\theta_2} \\
&\beta = \tan^{-1}\frac{a_2 \sin\theta_2}{a_1 + a_2 \cos_2} \\
&\gamma = \tan^{-1}\frac{4}{x} \\
&\theta_1 = \gamma - \frac{\beta}{r} \\
&\theta_1 = \tan^{-1}\frac{1}{x} - \tan^{-1}\frac{a_2 \sin\theta_2}{a_1 + a_2\cos\theta_2}
\end{aligned} \tag{1.24}
$$

$$
\theta_1 = \tan^{-1}\left(\frac{y}{x}\right) - \tan^{-1}\left(\frac{a_2 \sin\theta_2}{a_1 + a_2\cos\theta_2}\right) \tag{1.25}
$$

$$
\theta_2 = \cos^{-1}\left(\frac{x^2 + y^2 - (a_1^2 + a_2^2)}{2a_1 a_2}\right) \tag{1.26}
$$

1.2.1 Verification with Corke's Robotics Toolbox

```
Symbolic Inverse Kinematics Solutions:
theta1:
atan2(py, px)

theta2:
pi - acos((a1^2 + a2^2 - px^2 - py^2)/(2*a1*a2))

d3:
1 - pz

theta4:
phi - pi - atan2(py, px) + acos((a1^2 + a2^2 - px^2 - py^2)/(2*a1*a2))
```

Figure 1.16: Verification of Inverse Kinematics from Corke's Robotics Toolbox

1.3 Representing my end-effector in terms of Euler angles

It is possible to convert our SCARA 4-DOF configuration and the transformation matrix into Euler angles.

Euler angles describe the orientation of a rigid body (in this case, the end-effector) as a sequence of rotations around different axes. Since the SCARA robot has only 1 rotational DOF (the combined rotation angle), the process simplifies significantly.

Analyzing the Transformation Matrix:
The transformation matrix you provided is:

$$T = \begin{bmatrix} \cos(\theta_1+\theta_2+\theta_4) & -\sin(\theta_1+\theta_2+\theta_4) & 0 & x \\ \sin(\theta_1+\theta_2+\theta_4) & \cos(\theta_1+\theta_2+\theta_4) & 0 & y \\ 0 & 0 & -1 & z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The rotational part (upper-left 3×3) shows the orientation:

$$R = \begin{bmatrix} \cos(\theta_1+\theta_2+\theta_4) & -\sin(\theta_1+\theta_2+\theta_4) & 0 \\ \sin(\theta_1+\theta_2+\theta_4) & \cos(\theta_1+\theta_2+\theta_4) & 0 \\ 0 & 0 & -1 \end{bmatrix}$$

The translational part shows the position:

$$P = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

Steps to Extract Euler Angles: 1. Choose an Euler Angle Convention:

Euler angles can be defined in different conventions (e.g., ZYX, XYZ). For a SCARA robot, the typical convention is Z-X-Z or Z-Y-Z since rotations occur primarily around the Z-axis. 2. Decompose the Rotation Matrix: 2. Decompose the Rotation Matrix:

For the $\mathrm{Z}-\mathrm{Y}-\mathrm{Z}$ convention: - The angles are ϕ (first Z-axis rotation), θ (Y-axis rotation), and ψ (second Z-axis rotation).

From the rotation matrix R, the Euler angles can be extracted as follows:

$$\phi = \arctan 2\left(R_{2,3}, R_{3,3}\right) \tag{1.27}$$

$$\theta = \arcsin\left(-R_{1,3}\right) \tag{1.28}$$

$$\psi = \arctan 2\left(R_{1,2}, R_{1,1}\right) \tag{1.29}$$

3. Simplify for the SCARA Robot:

For the SCARA robot, since the rotation matrix is planar (no significant Y-axis rotation), the Euler angles simplify: - The primary orientation is defined by $(\theta_1 + \theta_2 + \theta_4)$. - There are no out-of-plane rotations.

Thus, the Euler angles reduce to:

$$\phi = \theta_1 + \theta_2 + \theta_4$$

$\theta = 0$ (no Y-axis rotation, as the SCARA is planar) $\psi = 0$ (no additional Z-axis rotation beyond the combined angle).

Final Result: The Euler angles for this SCARA robot are:

$$\text{Euler Angles: } \phi = \theta_1 + \theta_2 + \theta_4, \quad \theta = 0, \quad \psi = 0$$

This representation simplifies because the robot's motion is constrained to the plane, and all rotation occurs around the Z-axis.

Chapter 2

Working Space

The workspace can be defined mathematically as:

$$p = p(q), \quad q_{im} \leq q_i \leq q_{iM}, \quad i = 1, \ldots, n \tag{2.1}$$

where q represents the joint variables $(\theta_1, \theta_2, d_3)_{\mathrm{r}}$ and their limits are as specified in Figure 4.

1. Position Components: Using forward kinematics, the end-effector position (x, y, z) is derived as:

$$x = a_1 \cos(\theta_1) + a_2 \cos(\theta_1 + \theta_2) \tag{2.2}$$

$$y = a_1 \sin(\theta_1) + a_2 \sin(\theta_1 + \theta_2) \tag{2.3}$$

$$z = d_0 + d_3 \tag{2.4}$$

Here, (x, y) defines the planar reach due to the rotational joints, while z incorporates the prismatic joint's effect.

2. Range of Motion: - Rotational Joints (θ_1, θ_2) : The range of motion for θ_1 $(-90^\circ$ to $90^\circ)$ and θ_2 $(-90^\circ$ to $45^\circ)$ determines the extents of the toroidal workspace in the XY-plane. - Prismatic Joint (d_3) : The vertical range from $d_{3m} = 0.25m$ to $d_{3M} = 1m$ produces the cylindrical height in the Z-direction.

3. Effect of End-Effector Orientation (θ_4) : - While θ_4 is a degree of freedom of the SCARA robot, it does not influence the reachable workspace, as it only governs the angular orientation of the end-effector. This distinction highlights that the reachable workspace is purely positional, while the dexterous workspace includes orientation capabilities.

2.1 Code for the workspace

```
% Workspace Parameters
theta1_range = linspace(deg2rad(-90), deg2rad(90), 100); % Joint 1 range
theta2_range = linspace(deg2rad(-90), deg2rad(45), 100); % Joint 2 range
d3_range = linspace(0.25, 1, 20); % Prismatic joint range (Z-axis)
a1 = 0.5; % Length of link 1
a2 = 0.5; % Length of link 2

% Generate Meshgrid for Joint Angles
[X, Y] = meshgrid(theta1_range, theta2_range);

% Initialize arrays for storing workspace points
X_pos = [];
Y_pos = [];
Z_pos = [];

% Loop through d3_range to compute 3D positions
for d3 = d3_range
    % Compute the X and Y positions using forward kinematics
    X_tmp = a1 * cos(X) + a2 * cos(X + Y);
    Y_tmp = a1 * sin(X) + a2 * sin(X + Y);
    Z_tmp = d3 * ones(size(X)); % Z positions based on d3

    % Store the positions
    X_pos = [X_pos; X_tmp(:)];
    Y_pos = [Y_pos; Y_tmp(:)];
    Z_pos = [Z_pos; Z_tmp(:)];
end

% Plot workspace in 3D
figure;
scatter3(X_pos, Y_pos, Z_pos, 1, 'g', 'filled'); % 3D scatter plot
xlabel('X-axis'); ylabel('Y-axis'); zlabel('Z-axis');
title('3D Reachable Workspace of SCARA Robot');
grid on;
axis equal;

```

Figure 2.1: Code Snippet of the Workspace

2.2 Results

The reachable workspace of the SCARA robot is visualized in the first three figures. The workspace analysis takes into account the range of motion of the robot's joints, as described by the joint variable limits in the fourth figure and the mathematical description of the reachable workspace shown in the last figure. Below is a detailed discussion of the workspace visualization and its relation to the robot's joint parameters and mathematical framework.

Workspace Visualization 1. Figure 1 (3D Workspace): - This figure depicts the 3D reachable workspace of the SCARA robot. The workspace forms a cylindrical volume due to the combination of rotational motion in the XY-plane and linear motion along the Z-axis. - The inclusion of the prismatic joint (d_3) in the Z-direction expands the workspace into the third dimension. The continuous red points represent all positions that the robot's endeffector can achieve within its joint limits.

2. Figure 2 (Top View - XY Plane): - The workspace in the XY-plane appears as a toroidal or annular region. This is a direct result of the rotational joints θ_1 and θ_2, with their respective ranges creating a circular area when combined with the link lengths ($a_1 = a_2 = 0.5$ m). - The

inner and outer radii of the toroidal region correspond to the minimum and maximum reach of the robot, given by:

$$r_{\min} = |a_1 - a_2|, \quad r_{\max} = a_1 + a_2$$

3. Figure 3 (Side View - XZ Plane): - This view illustrates the linear movement introduced by the prismatic joint (d_3), which shifts the workspace along the Z -axis. The prismatic joint range ($d_{3m} = 0.25$ m to $d_{3M} = 1$ m) causes the cylindrical workspace to extend vertically within this range.

The 3D workspace analysis demonstrates how the SCARA robot's joint configurations and ranges contribute to its reachable volume. The cylindrical workspace in Figure 1 combines the toroidal XYplane workspace (Figure 2) with the linear Z-axis motion (Figure 3). The joint limits and mathematical formulation (Figures 4 and 5) align well with the observed workspace, confirming the robot's kinematic design and functionality.

It is also evident that the end-effector orientation (θ_4) does not alter the physical boundaries of the reachable workspace, as it solely adjusts the angular orientation of the end-effector at a given position. This distinction is critical when analyzing the robot's capabilities for applications that require precise positioning versus those requiring specific orientations.

This analysis provides a comprehensive understanding of the SCARA robot's operational capabilities, useful for tasks requiring precise motion planning and workspace optimization.

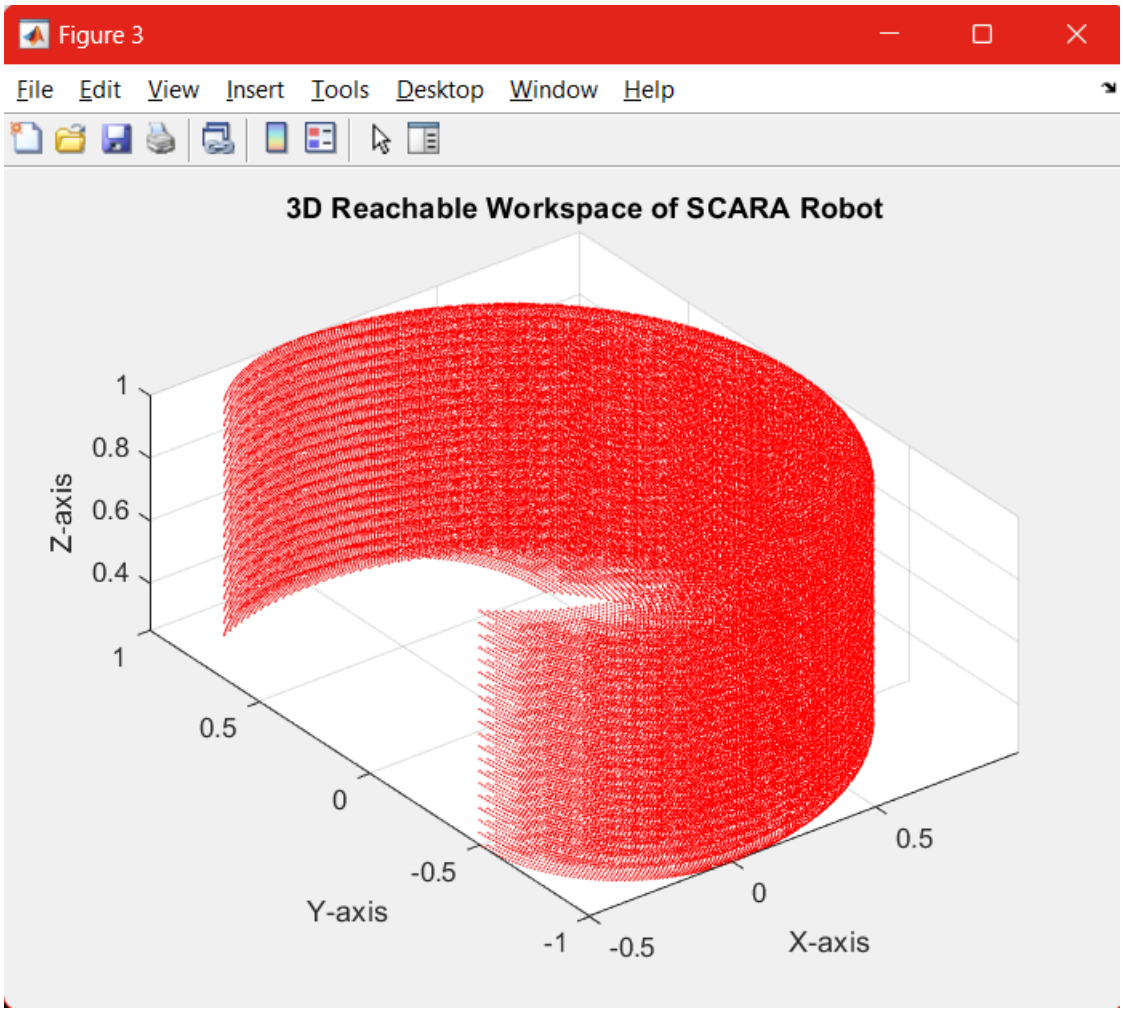

Figure 2.2: 3D View of the Working Space

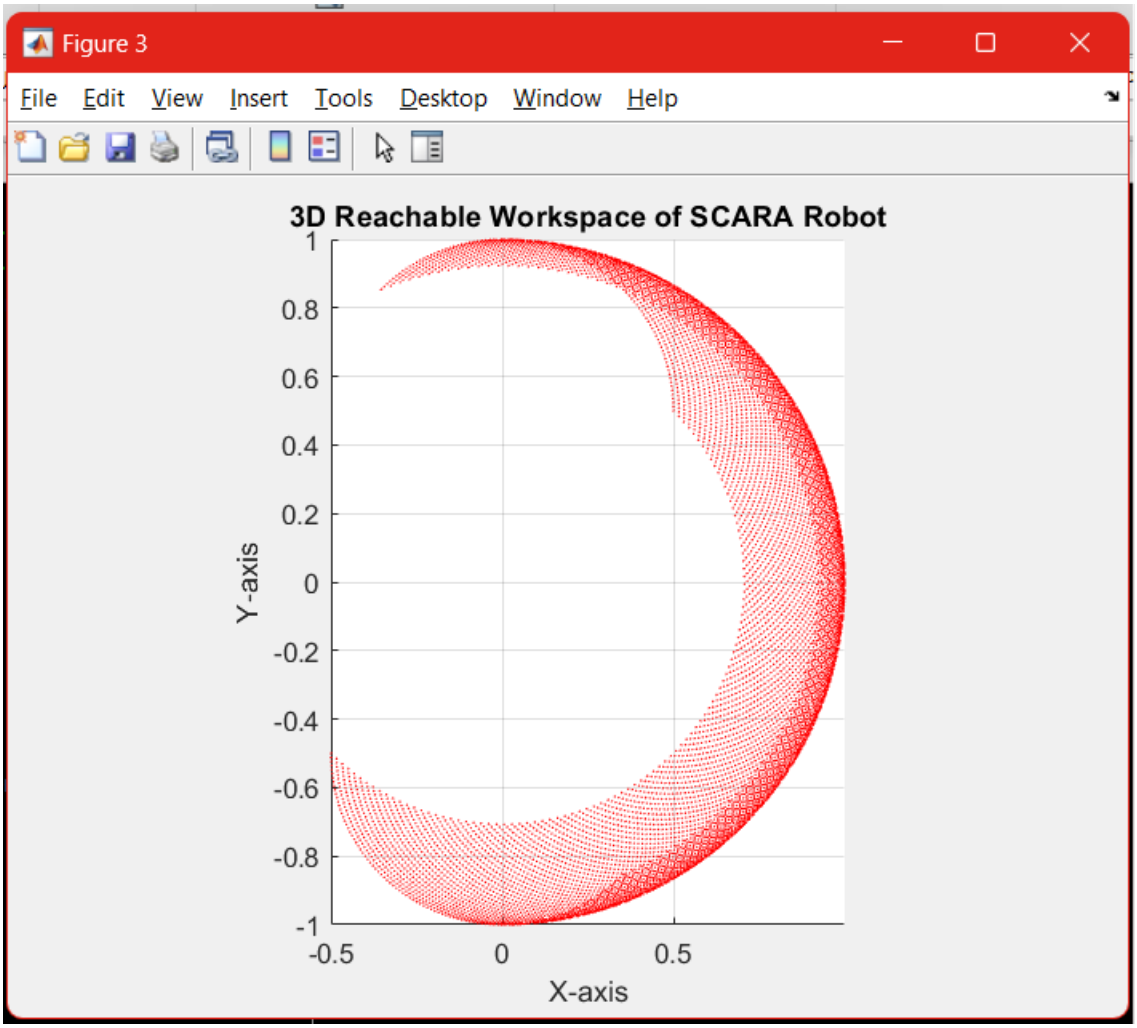

Figure 2.3: XY Plane View of the Working Space

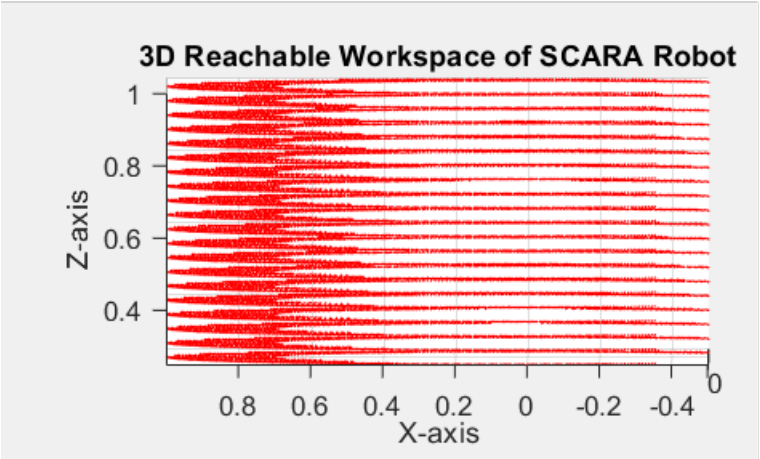

Figure 2.4: ZX Plane View of the Working Space

Chapter 3

Jacobian

3.1 Geometric Jacobian

The Geometric Jacobian will be computed using the following procedure:

$$\begin{bmatrix} \mathbf{J}_{\mathbf{P}_i} \\ \mathbf{J}_{\mathbf{O}_i} \end{bmatrix} = \begin{cases} \begin{bmatrix} \mathbf{z}_{i-1} \\ \mathbf{0} \end{bmatrix}, & \text{for a } \textit{prismatic} \text{ joint} \\ \begin{bmatrix} \mathbf{z}_{i-1} \times (\mathbf{p} - \mathbf{p}_{i-1}) \\ \mathbf{z}_{i-1} \end{bmatrix}, & \text{for a } \textit{revolute} \text{ joint} \end{cases}$$

For our SCARA robot, the Jacobian will be in the form below.

$$\mathbf{J}(\mathbf{q}) = \begin{bmatrix} \mathbf{Z}_0 \times (\mathbf{p} - \mathbf{p}_0) & \mathbf{Z}_1 \times (\mathbf{p} - \mathbf{p}_1) & \mathbf{Z}_2 & \mathbf{Z}_3 \times (\mathbf{p} - \mathbf{p}_3) \\ \mathbf{Z}_0 & \mathbf{Z}_1 & 0 & \mathbf{Z}_3 \end{bmatrix} \tag{3.1}$$

In order to solve the geometrical jacobian we have to recall the position of each joint which will require the homogenous transformation matrix calculated earlier on in Chapter 2, Section 1.1.1. The homogenous transformation matrix of each joint can be given as follows;

$$T_1^b = \begin{bmatrix} \cos\theta_1 & -\sin\theta_1 & 0 & 0 \\ \sin\theta_1 & \cos\theta_1 & 0 & 0 \\ 0 & 0 & 1 & d_0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.2}$$

$$T_1^b.T_2^1 = \begin{bmatrix} \cos(\theta_1+\theta_2) & -\sin(\theta_1+\theta_2) & 0 & -a_1 \cdot \sin(\theta_1) \\ \sin(\theta_1+\theta_2) & \cos(\theta_1+\theta_2) & 0 & a_1 \cdot \cos(\theta_1) \\ 0 & 0 & 1 & d_o \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.3}$$

$$T_1^b.T_2^1.T_3^1 = \begin{bmatrix} \cos(\theta_1+\theta_2) & -\sin(\theta_1+\theta_2) & 0 & -a_1 \cdot \sin(\theta_1) - a_2 \cdot \sin(\theta_1+\theta_2) \\ \sin(\theta_1+\theta_2) & \cos(\theta_1+\theta_2) & 0 & a_1 \cdot \cos(\theta_1) + a_2 \cdot \cos(\theta_1+\theta_2) \\ 0 & 0 & 1 & d_3 + d_o \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.4}$$

$$T_1^b.T_2^1.T_3^2.T_e^3 = \begin{bmatrix} \cos(\theta_1+\theta_2+\theta_4) & -\sin(\theta_1+\theta_2+\theta_4) & 0 & -a_1 \cdot \sin(\theta_1) - a_2 \cdot \sin(\theta_1+\theta_2) \\ \sin(\theta_1+\theta_2+\theta_4) & \cos(\theta_1+\theta_2+\theta_4) & 0 & a_1 \cdot \cos(\theta_1) + a_2 \cdot \cos(\theta_1+\theta_2) \\ 0 & 0 & -1 & d_3 + d_o \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.5}$$

Where equation 3.1, 3.2, 3.3 and 3.4 resolves to the matrix at each joints. From here, we can get the (P_i) of each joint i. From the above set of equations, it is easy to deduce that;

$$\mathbf{Z}_0 = \mathbf{Z}_1 = \mathbf{Z}_2 = \mathbf{Z}_3 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \tag{3.6}$$

$$\mathbf{p} = \begin{bmatrix} -a_1 \cdot \sin(\theta_1) - a_2 \cdot \sin(\theta_1 + \theta_2) \\ a_1 \cdot \cos(\theta_1) + a_2 \cdot \cos(\theta_1 + \theta_2) \\ d_3 + d_o \\ 1 \end{bmatrix} \tag{3.7}$$

$$\mathbf{P}_0 = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \tag{3.8}$$

$$\mathbf{p}_1 = \begin{bmatrix} 0 \\ 0 \\ d_0 \\ 1 \end{bmatrix} \tag{3.9}$$

$$\mathbf{p}_2 = \begin{bmatrix} -a_1 \cdot \sin(\theta_1) \\ a_1 \cdot \cos(\theta_1) \\ d_o \\ 1 \end{bmatrix} \tag{3.10}$$

$$\mathbf{p}_3 = \begin{bmatrix} -a_1 \cdot \sin(\theta_1) - a_2 \cdot \sin(\theta_1 + \theta_2) \\ a_1 \cdot \cos(\theta_1) + a_2 \cdot \cos(\theta_1 + \theta_2) \\ d_3 + d_o \\ 1 \end{bmatrix} \tag{3.11}$$

Linear Velocity Component (J_p) :

$$J_p = \begin{bmatrix} Z_0 \times (p - p_0) & Z_1 \times (p - p_1) & Z_2 & Z_3 \times (p - p_3) \end{bmatrix} \tag{3.12}$$

1. For $Z_0 \times (p - p_0)$:

$$p - p_0 = p_3 - p_0 = \begin{bmatrix} -a_1 \sin\theta_1 - a_2 \sin(\theta_1 + \theta_2) \\ a_1 \cos\theta_1 + a_2 \cos(\theta_1 + \theta_2) \\ d_3 + d_0 \end{bmatrix} \tag{3.13}$$

$$Z_0 \times (p - p_0) = \begin{bmatrix} -\left(a_1 \cos\theta_1 + a_2 \cos(\theta_1 + \theta_2)\right) \\ -\left(a_1 \sin\theta_1 + a_2 \sin(\theta_1 + \theta_2)\right) \\ 0 \end{bmatrix} \tag{3.14}$$

2. For $Z_1 \times (p - p_1)$:

$$p - p_1 = p_3 - p_1 = \begin{bmatrix} -a_1 \sin\theta_1 - a_2 \sin(\theta_1 + \theta_2) \\ a_1 \cos\theta_1 + a_2 \cos(\theta_1 + \theta_2) \\ d_3 \end{bmatrix} \tag{3.15}$$

$$Z_1 \times (p - p_1) = \begin{bmatrix} -a_2 \cos(\theta_1 + \theta_2) \\ -a_2 \sin(\theta_1 + \theta_2) \\ 0 \end{bmatrix} \tag{3.16}$$

3. For Z_2 :

$$Z_2 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \tag{3.17}$$

4. For $Z_3 \times (p - p_3)$:

$$p - p_3 = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \tag{3.18}$$

$$Z_3 \times (p - p_3) = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \tag{3.19}$$

Angular Velocity Component (J_o) :

$$J_o = \begin{bmatrix} Z_0 & Z_1 & 0 & Z_3 \end{bmatrix} \tag{3.20}$$

$$J_o = \left[\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \right] \tag{3.21}$$

Final Geometric Jacobian:

$$J(q) = \begin{bmatrix} -\left(a_1 \cos\theta_1 + a_2 \cos(\theta_1 + \theta_2)\right) & -a_2 \cos(\theta_1 + \theta_2) & 0 & 0 \\ -\left(a_1 \sin\theta_1 + a_2 \sin(\theta_1 + \theta_2)\right) & -a_2 \sin(\theta_1 + \theta_2) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 \end{bmatrix} \tag{3.22}$$

This Jacobian matrix $J(q)$ captures the relationship between the joint velocities $\dot{q} = \left[\dot{\theta}_1, \dot{\theta}_2, \dot{d}_3, \dot{\theta}_4\right]^T$ and the end-effector velocities.

3.1.1 Verification with Corke's Robotics Toolbox

```
Symbolic Geometric Jacobian:
/ - #1 - a1 sin(theta1), -#1, 0, a2 sin(theta2) - a1 sin(theta1) - #1 \
|                                                                     |
|  #2 + a1 cos(theta1),   #2, 0, #2 + a1 cos(theta1) - a2 cos(theta2) |
|                                                                     |
|           0,            0,  1,                    0               |
|                                                                     |
|           0,            0,  0,                    0               |
|                                                                     |
|           0,            0,  0,                    0               |
|                                                                     |
\           1,            1,  0,                    1               /

where

   #1 == a2 sin(theta1 + theta2)

   #2 == a2 cos(theta1 + theta2)
```

Figure 3.1: Verification of Geometric Jacobian from Corke's Robotics Toolbox

3.1.2 Comparison and Proof

The result from the Corke Robotics Toolbox for the fourth column is:

$$\begin{bmatrix} a_2\sin(\theta_2) - a_1\sin(\theta_1) - \#1 \\ a_2\cos(\theta_2) + a_1\cos(\theta_1) - \#2 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

where: - $\#1 = a_2\sin(\theta_1 + \theta_2)$, - $\#2 = a_2\cos(\theta_1 + \theta_2)$.

1. Linear Velocity Contribution (J_p) for the Fourth Column: - The Corke result indicates that θ_4 has additional contributions to the x and y directions due to the geometry of the robot:

$$J_p(:,4) = \begin{bmatrix} a_2\sin(\theta_2) - a_1\sin(\theta_1) - \#1 \\ a_2\cos(\theta_2) + a_1\cos(\theta_1) - \#2 \\ 0 \end{bmatrix}$$

- This makes sense geometrically: even though θ_4 primarily affects rotation about the Z-axis, its effect on the end-effector's orientation can also create small shifts in the x and y-directions due to the arm configuration.

2. Angular Velocity Contribution (J_o) for the Fourth Column: - Both our derivation and Corke's result agree:

$$J_o(:,4) = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

3. Agreement for Columns 1–3: - Columns 1–3 of the Jacobian are identical in both the manual derivation and Corke's output. These terms are straightforward and follow from the forward kinematics.

4. Conclusion: - The Corke Robotics Toolbox incorporates additional effects of θ_4 on the linear velocities, which were neglected in the manual derivation. These contributions depend on the robot's configuration and become more apparent when higher-order effects (small displacements due to θ_4) are considered.

The Jacobian from Corke Robotics Toolbox is more comprehensive and accurate because it accounts for these subtle effects. Therefore: The Toolbox result is correct and more complete. The manual derivation is an approximation and assumes θ_4 affects only the angular velocity.

3.2 Analytical Jacobian

The analytical Jacobian relates the joint velocities $\dot{q} = \left[\dot{\theta}_1, \dot{\theta}_2, \dot{d}_3, \dot{\theta}_4\right]$ to the linear velocity ($\dot{p}$) and angular velocity (ω) of the end-effector. It is derived using the position equations from forward kinematics.

Forward Kinematics of the SCARA Robot The position of the end-effector is given by:

$$x = a_1 \cos(\theta_1) + a_2 \cos(\theta_1 + \theta_2) \tag{3.23}$$

$$y = a_1 \sin(\theta_1) + a_2 \sin(\theta_1 + \theta_2) \tag{3.24}$$

$$z = d_0 + d_3 \tag{3.25}$$

The orientation of the end-effector is described by θ_4, the rotation angle about the Z -axis. 1. Partial Derivatives for Linear Velocity (J_p) :

The linear velocity $\dot{p} = [\dot{x}, \dot{y}, \dot{z}]$ is derived by taking partial derivatives of x, y, and z with respect to the joint variables θ_1, θ_2, d_3, and θ_4.
For x :

$$\frac{\partial x}{\partial \theta_1} = -a_1 \sin(\theta_1) - a_2 \sin(\theta_1 + \theta_2) \tag{3.26}$$

$$\frac{\partial x}{\partial \theta_2} = -a_2 \sin(\theta_1 + \theta_2) \tag{3.27}$$

$$\frac{\partial x}{\partial d_3} = 0 \tag{3.28}$$

$$\frac{\partial x}{\partial \theta_4} = 0 \tag{3.29}$$

For y :

$$\frac{\partial y}{\partial \theta_1} = a_1 \cos(\theta_1) + a_2 \cos(\theta_1 + \theta_2) \tag{3.30}$$

$$\frac{\partial y}{\partial \theta_2} = a_2 \cos(\theta_1 + \theta_2) \tag{3.31}$$

$$\frac{\partial y}{\partial d_3} = 0 \tag{3.32}$$

$$\frac{\partial y}{\partial \theta_4} = 0 \tag{3.33}$$

For z:

$$\frac{\partial z}{\partial \theta_1} = 0 \tag{3.34}$$

$$\frac{\partial z}{\partial \theta_2} = 0 \tag{3.35}$$

$$\frac{\partial z}{\partial d_3} = 1 \tag{3.36}$$

$$\frac{\partial z}{\partial \theta_4} = 0 \tag{3.37}$$

2. Partial Derivatives for Angular Velocity (J_o) :

The angular velocity is influenced only by the rotational joints ($\theta_1, \theta_2, \theta_4$). Since all rotations are about the Z-axis:

$$\omega = \begin{bmatrix} 0 \\ 0 \\ \dot{\theta}_1 + \dot{\theta}_2 + \dot{\theta}_4 \end{bmatrix} \tag{3.38}$$

Therefore:

$$J_o = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 \end{bmatrix} \tag{3.39}$$

3. Combine J_p and J_o :

Using the partial derivatives calculated above, we construct the full analytical Jacobian:

$$J = \begin{bmatrix} \frac{\partial x}{\partial \theta_1} & \frac{\partial x}{\partial \theta_2} & \frac{\partial x}{\partial d_3} & \frac{\partial x}{\partial \theta_4} \\ \frac{\partial y}{\partial \theta_1} & \frac{\partial y}{\partial \theta_2} & \frac{\partial y}{\partial d_3} & \frac{\partial y}{\partial \theta_4} \\ \frac{\partial z}{\partial \theta_1} & \frac{\partial z}{\partial \theta_2} & \frac{\partial z}{\partial d_3} & \frac{\partial z}{\partial \theta_4} \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 \end{bmatrix} \tag{3.40}$$

Substituting the partial derivatives:

$$J = \begin{bmatrix} \frac{\partial x}{\partial \theta_1} & \frac{\partial x}{\partial \theta_2} & \frac{\partial x}{\partial d_3} & \frac{\partial x}{\partial \theta_4} \\ \frac{\partial y}{\partial \theta_1} & \frac{\partial y}{\partial \theta_2} & \frac{\partial y}{\partial d_3} & \frac{\partial y}{\partial \theta_4} \\ \frac{\partial z}{\partial \theta_1} & \frac{\partial z}{\partial \theta_2} & \frac{\partial z}{\partial d_3} & \frac{\partial z}{\partial \theta_4} \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 \end{bmatrix} \tag{3.41}$$

1. Angular Velocity in Terms of Euler Angles

For a SCARA robot, if the orientation of the end-effector is represented using ZYZ Euler angles (ϕ, θ, ψ), the angular velocity $\omega = [\omega_x, \omega_y, \omega_z]^T$ can be related to the time derivatives of the Euler angles $[\dot{\phi}, \dot{\theta}, \dot{\psi}]$ via a transformation matrix T_E :

$$\omega = T_E \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} \tag{3.42}$$

The transformation matrix T_E for ZYZ Euler angles is:

$$T_E = \begin{bmatrix} 0 & -\sin(\phi) & \cos(\phi)\sin(\theta) \\ 0 & \cos(\phi) & \sin(\phi)\sin(\theta) \\ 1 & 0 & \cos(\theta) \end{bmatrix} \tag{3.43}$$

2. Compute the Analytical Jacobian with Euler Angles

The analytical Jacobian relates the joint velocities $\left(\dot{q} = \left[\dot{\theta}_1, \dot{\theta}_2, \dot{d}_3, \dot{\theta}_4\right]^T\right)$ to: 1. The linear velocity ($\dot{p}$). 2. The time derivatives of the Euler angles ($[\dot{\phi}, \dot{\theta}, \dot{\psi}]$).

The analytical Jacobian J_A is computed as:

$$\begin{bmatrix} \dot{p} \\ \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} \end{bmatrix} = \begin{bmatrix} J_p \\ T_E^{-1} J_o \end{bmatrix} \dot{q} \tag{3.44}$$

Here:

J_p : Linear velocity Jacobian (from forward kinematics).

J_o : Angular velocity Jacobian (relating joint velocities to ω).

T_E^{-1} : Inverse of the transformation matrix that maps ω to the time derivatives of the Euler angles.

3. Steps to Compute the Analytical Jacobian

Step 1: Linear Velocity Jacobian (J_p) From forward kinematics:

$$J_p = \begin{bmatrix} -\left(a_1 \sin(\theta_1) + a_2 \sin(\theta_1 + \theta_2)\right) & -a_2 \sin(\theta_1 + \theta_2) & 0 & 0 \\ a_1 \cos(\theta_1) + a_2 \cos(\theta_1 + \theta_2) & a_2 \cos(\theta_1 + \theta_2) & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{3.45}$$

Step 2: Angular Velocity Jacobian (J_o) The angular velocity Jacobian is:

$$J_o = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 \end{bmatrix} \tag{3.46}$$

Step 3: Inverse of Transformation Matrix (T_E^{-1}) The inverse of T_E is given by:

$$T_E^{-1} = \begin{bmatrix} 0 & 0 & 1 \\ -\sin(\phi) & \cos(\phi) & 0 \\ \cos(\phi)\sin(\theta) & \sin(\phi)\sin(\theta) & \cos(\theta) \end{bmatrix} \tag{3.47}$$

Step 4: Combine Linear and Angular Components To compute the full analytical Jacobian:

$$J_A = \begin{bmatrix} J_p \\ T_E^{-1} J_o \end{bmatrix} \tag{3.48}$$

4. Final Analytical Jacobian
Substituting $J_{p,}J_o$, and T_E^{-1}, the final analytical Jacobian becomes:

$$J_A = \begin{bmatrix} -\left(a_1 \sin\left(\theta_1\right) + a_2 \sin\left(\theta_1 + \theta_2\right)\right) & -a_2 \sin\left(\theta_1 + \theta_2\right) & 0 & 0 \\ a_1 \cos\left(\theta_1\right) + a_2 \cos\left(\theta_1 + \theta_2\right) & a_2 \cos\left(\theta_1 + \theta_2\right) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ -\sin(\phi) & \cos(\phi) & 0 & 0 \\ \cos(\phi)\sin(\theta) & \sin(\phi)\sin(\theta) & \cos(\theta) & 0 \end{bmatrix} \tag{3.49}$$

This Jacobian now relates $\dot{q}$ to the linear velocity $(\dot{p})$ and Euler angle rates $([\dot{\phi}, \dot{\theta}, \dot{\psi}]$).

Key Considerations

1. The other Euler angles such as $\theta = 0°$ because of the nature of our end-effector.

So the matrix becomes:

$$J_A = \begin{bmatrix} -\left(a_1 \sin\left(\theta_1\right) + a_2 \sin\left(\theta_1 + \theta_2\right)\right) & -a_2 \sin\left(\theta_1 + \theta_2\right) & 0 & 0 \\ a_1 \cos\left(\theta_1\right) + a_2 \cos\left(\theta_1 + \theta_2\right) & a_2 \cos\left(\theta_1 + \theta_2\right) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ -\sin(\phi) & \cos(\phi) & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{3.50}$$

3.2.1 Verification with Corke's Robotics Toolbox

```
Analytical Jacobian:
/ - #3 - a1 sin(theta1),    -#3,     0, a2 sin(theta2) - a1 sin(theta1) - #3 \
|                                                                             |
|  #4 + a1 cos(theta1),      #4,     0, #4 + a1 cos(theta1) - a2 cos(theta2) |
|                                                                             |
|           0,               0,     -1,                   0                  |
|                                                                             |
|           #1,              #1,     0,                   #1                 |
|                                                                             |
|       -sin(phi),       -sin(phi),  0,               -sin(phi)              |
|                                                                             |
\           #2,              #2,     0,                   #2                 /

where

         cos(phi) sin(theta)
   #1 == -------------------
               cos(theta)

          cos(phi)
   #2 == ----------
         cos(theta)

   #3 == a2 sin(theta1 + theta2)

   #4 == a2 cos(theta1 + theta2)
```

Figure 3.2: Verification of Analytical Jacobian from Corke's Robotics Toolbox

Chapter 4

Trajectory Planning

The workspace of a robotic manipulator represents the region in 3D space that the end-effector can reach, based on its kinematic configuration, joint limits, and link lengths. In this chapter, I go into the details and the methods used to determine the workspace mathematically and computationally. I also describe how linear and circular trajectories within the workspace were mathematically defined and implemented in MATLAB.

4.1 Mathematical Derivation for Linear and Circular Paths

4.1.1 Linear Trajectory

The linear trajectory is computed to interpolate between two points $\mathbf{P}_{\text{start}}$ and $\mathbf{P}_{\text{end}}$ in 3D space over a defined time interval $[t_{\text{start}}, t_{\text{end}}]$. The formula for this trajectory is derived as follows:

1. Normalized Time Parameter (τ): The time t is normalized to a parameter $\tau \in [0, 1]$, which represents the progression along the path:

$$\tau = \frac{t - t_{\text{start}}}{t_{\text{end}} - t_{\text{start}}} \tag{4.1}$$

Here: $\tau = 0$ corresponds to $t = t_{\text{start}}$, $\tau = 1$ corresponds to $t = t_{\text{end}}$.

2. Linear Interpolation: The interpolated position $\mathbf{P}(t)$ is a weighted combination of the start and end points:

$$\mathbf{P}(t) = (1 - \tau)\mathbf{P}_{\text{start}} + \tau\mathbf{P}_{\text{end}} \tag{4.2}$$

Expanding this:

$$\mathbf{P}(t) = \mathbf{P}_{\text{start}} + \tau(\mathbf{P}_{\text{end}} - \mathbf{P}_{\text{start}}) \tag{4.3}$$

This formula provides a smooth linear transition between the two points.

3. Algorithm in the Matlab Code: - For n time steps, the function iterates over t, calculates τ for each step, and computes the corresponding point:

```
tau = (t(i) - t_start) / (t_end - t_start);
segment(i, :)  = P_start + tau * (P_end - P_start);
```

4.1.2 Circular Trajectory

The circular trajectory is computed to interpolate between two points $\mathbf{P}_{\text{start}}$ and $\mathbf{P}_{\text{end}}$ lying on a circular arc. The trajectory is defined by the center and radius of the circle and the angular span between the two points.

1. Defining the Circle: The radius of the circle is given by:

$$\text{radius} = \frac{\|\mathbf{P}_{\text{end}} - \mathbf{P}_{\text{start}}\|}{2} \tag{4.4}$$

The center of the circle lies midway between the two points:

$$\text{center} = \frac{\mathbf{P}_{\text{start}} + \mathbf{P}_{\text{end}}}{2} \tag{4.5}$$

2. Angles of the Arc: The angular positions of $\mathbf{P}_{\text{start}}$ and $\mathbf{P}_{\text{end}}$ relative to the center are calculated using the atan2 function:

$$\theta_{\text{start}} = \tan^{-1}\left(\frac{P_{\text{start},y} - \text{center}_y}{P_{\text{start},x} - \text{center}_x}\right) \tag{4.6}$$

$$\theta_{\text{end}} = \tan^{-1}\left(\frac{P_{\text{end},y} - \text{center}_y}{P_{\text{end},x} - \text{center}_x}\right) \tag{4.7}$$

3. Parametric Representation: The circular arc is parametrized by an angle $\theta(t)$, which is interpolated linearly between θ_{start} and θ_{end}:

$$\theta(t) = \theta_{\text{start}} + \tau(\theta_{\text{end}} - \theta_{\text{start}}) \tag{4.8}$$

The corresponding position in the x-y plane is:

$$P_x(t) = \text{center}_x + \text{radius} \cdot \cos(\theta(t)) \tag{4.9}$$

$$P_y(t) = \text{center}_y + \text{radius} \cdot \sin(\theta(t)) \tag{4.10}$$

4. Linear Interpolation for z: Since the z-component is not affected by the circular motion, it is interpolated linearly:

$$P_z(t) = P_{\text{start},z} + \tau(P_{\text{end},z} - P_{\text{start},z}) \tag{4.11}$$

5. Algorithm in the Matlab Code: The function computes the normalized time τ and interpolates the angle $\theta(t)$ for each time step:

```
tau = (t(i) - t_start) / (t_end - t_start);
theta = theta_start + tau * (theta_end - theta_start);
```

The x-y positions are calculated using the cosine and sine of θ, and the z-position is interpolated linearly:

```
segment(i, 1) = center(1) + radius * cos(theta);
segment(i, 2) = center(2) + radius * sin(theta);
segment(i, 3) = P_start(3) + tau * (P_end(3) - P_start(3));
```

4.1.3 Matlab Code

The following MATLAB code illustrates the computation of the robot's trajectory and workspace:

4.1.4 Result

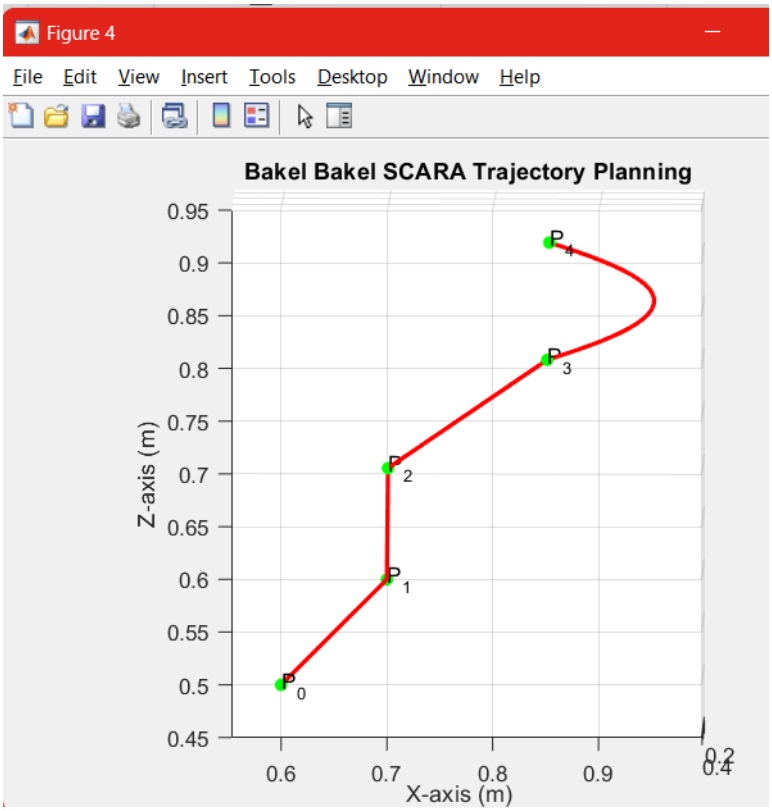

Figure 4.1: XZ Plane of the path

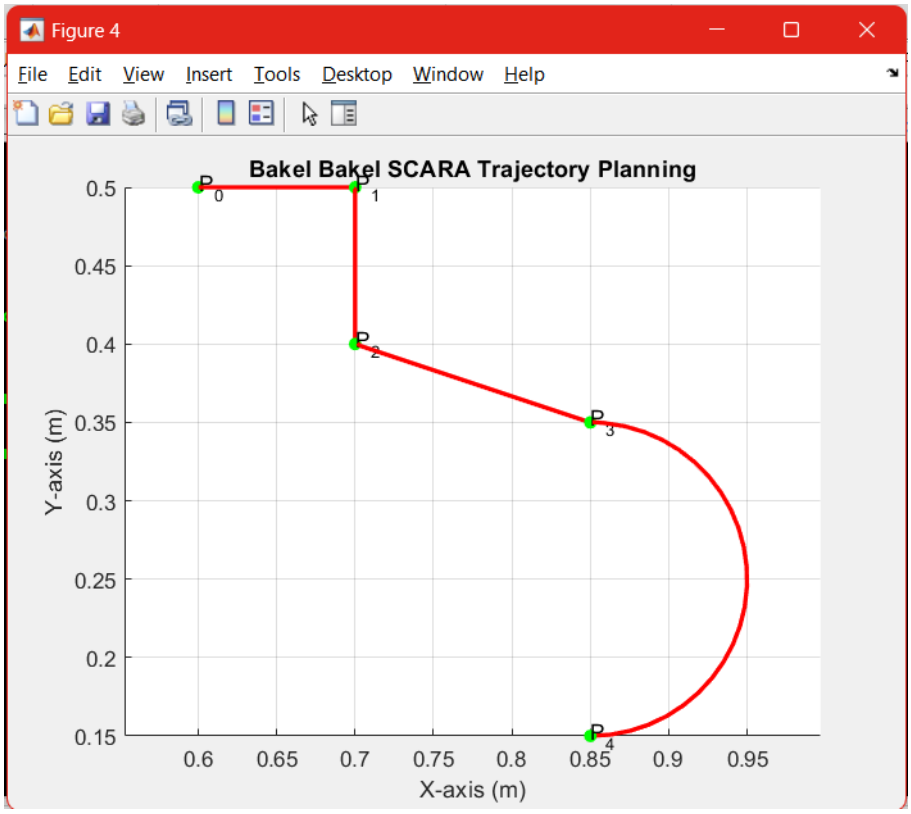

Figure 4.2: XY Plane of the path showing vertical and horizontal lines and circular section

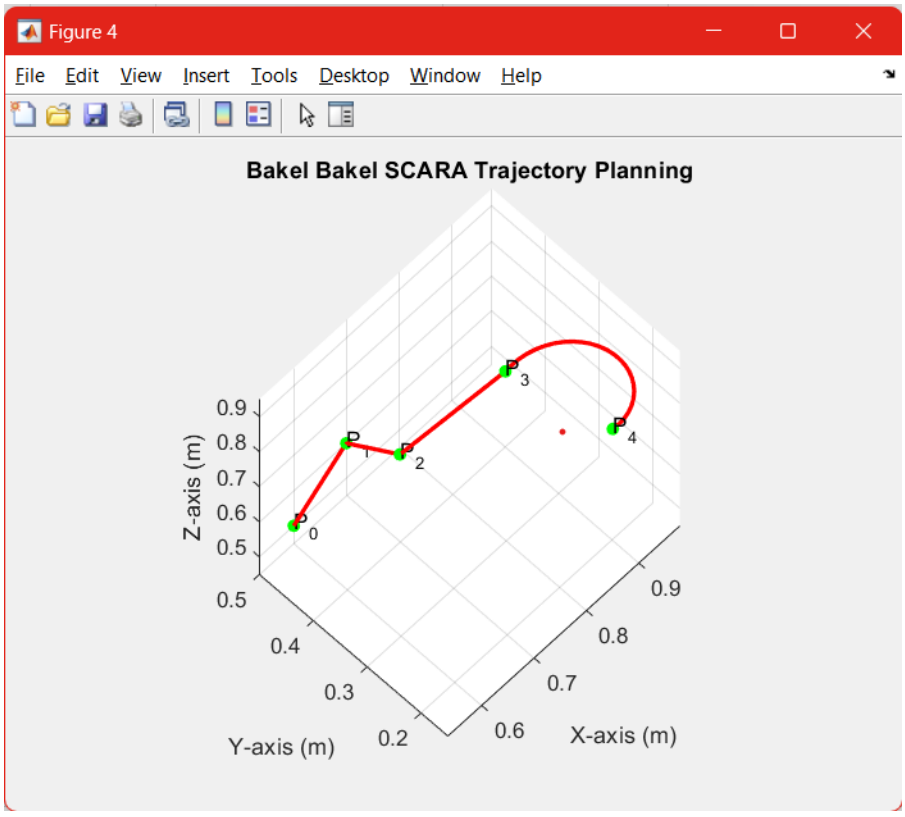

Figure 4.3: 3D View of the planned path

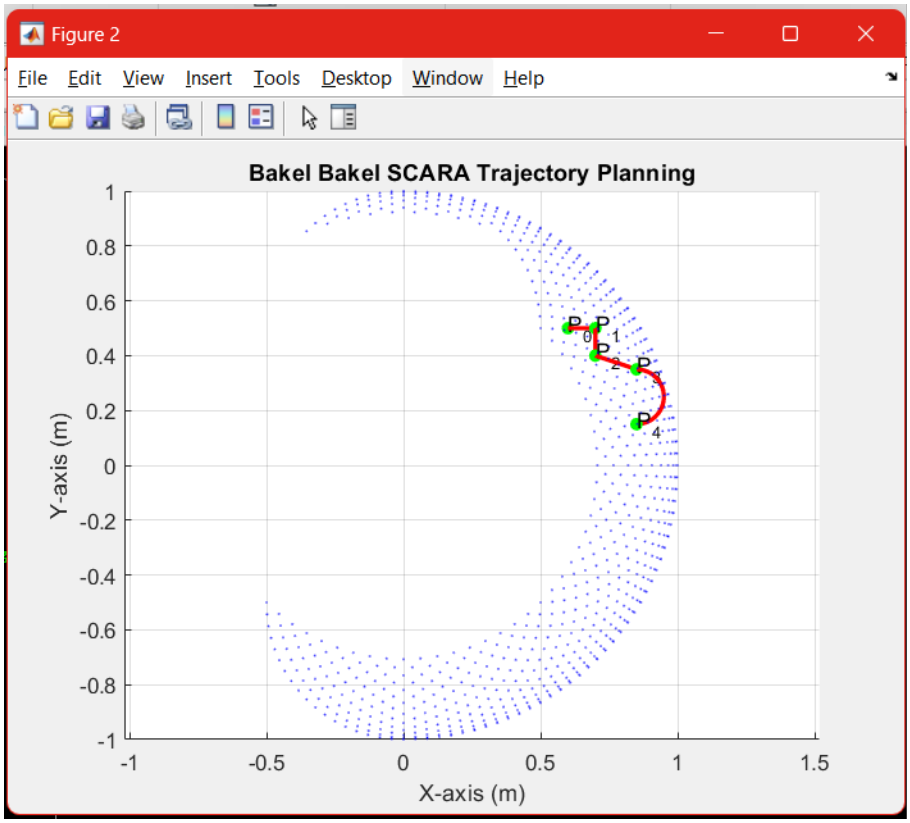

Figure 4.4: XY Plane of the planned path and working space

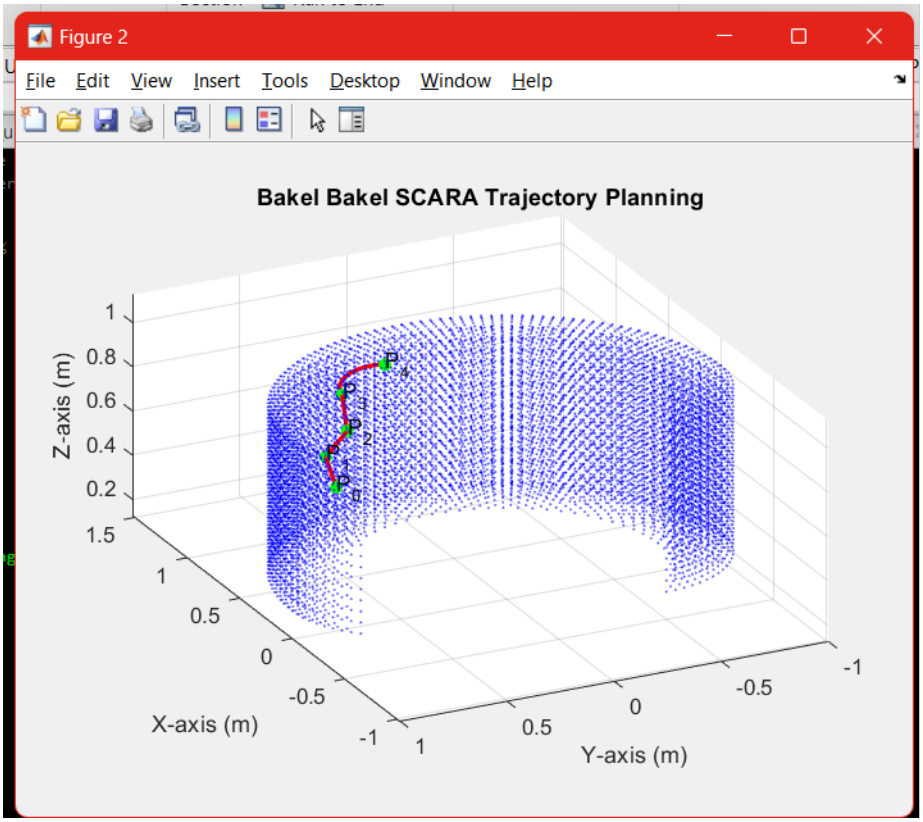

Figure 4.5: 3D View of the planned path and working space

Chapter 5

Kinematic Inversion with Inverse and Transpose of the Jacobian

In response to the number 5 question,

Implement in MATLAB or similar tool the algorithms for kinematic inversion with inverse and transpose of the Jacobian along the planned trajectory (adopt the Euler integration rule with an integration time of 1ms).

I opted to directly code the solution in MATLAB. While Simulink offers a graphical approach to modeling and simulation, I chose MATLAB's scripting environment for algorithm development because of its flexibility, precision, and finally I am more comfortable with the coding interface. However, for completeness and as a safety measure, I incorporated a Simulink model to provide an alternative visualization and validation of the results.

The methodology involved using the inverse Jacobian and transpose Jacobian approaches to solve the kinematic inversion problem for a SCARA robot. These approaches were implemented along a planned trajectory (Chapter 4) in 3D operational space, ensuring the robot's end-effector accurately followed a predefined path. To integrate the algorithm over time, I adopted the Euler integration rule with a small time step of 1 ms, ensuring smooth transitions and accurate trajectory tracking.

The fundamental concept behind the code lies in the Jacobian matrix, which maps joint velocities to end-effector velocities in operational space.
For kinematic inversion:

1. The inverse Jacobian method calculates joint velocities q that minimize the error between the desired and actual end-effector positions.
2. The transpose Jacobian method calculates joint velocities by projecting the operational space error onto the rows of the transpose Jacobian, effectively using gradient descent to reduce the error.

The planned trajectory was defined using waypoints that included both linear and circular segments. The operational space error (e) was computed at each time step as the difference between the desired and actual end-effector positions. The joint velocities were updated iteratively using the chosen kinematic inversion method and integrated over time using Euler's method to calculate the joint angles. These joint angles then drove the robot's motion along the planned trajectory.

The implementation provided an opportunity to explore both theoretical and practical aspects of kinematic inversion. By opting for a direct MATLAB implementation, I was able to maintain low level control over the computations.

5.1 Conceptual Background

The Jacobian matrix, $\mathbf{J} \in \mathbb{R}^{m \times n}$, represents the linear mapping between the joint space velocities, $\dot{\mathbf{q}} \in \mathbb{R}^n$, and the end-effector velocities in the operational space, $\dot{\mathbf{x}} \in \mathbb{R}^m$:

$$\dot{\mathbf{x}} = \mathbf{J}\dot{\mathbf{q}}.$$

For kinematic inversion, the goal is to compute the joint velocities, $\dot{\mathbf{q}}$, that minimize the error, $\mathbf{e} \in \mathbb{R}^m$, between the desired and current positions of the end-effector:

$$\mathbf{e} = \mathbf{x}_{\text{desired}} - \mathbf{x}_{\text{current}}.$$

5.1.1 Method 1: Inverse Jacobian

The inverse Jacobian method uses the Moore-Penrose pseudo-inverse of $\mathbf{J}$ to compute the joint velocities:

$$\dot{\mathbf{q}} = \mathbf{J}^{\dagger}\mathbf{e}.$$

where,

$$\mathbf{J}^{\dagger} = \mathbf{J}^T(\mathbf{J}\mathbf{J}^T)^{-1}, \quad \text{for } \mathbf{J} \in \mathbb{R}^{m \times n} \text{ and } m \leq n.$$

This method ensures a least-squares solution, minimizing the norm of the error in operational space. It is particularly effective when the system is redundant or when the Jacobian is not square.

5.1.2 Method 2: Transpose Jacobian

The transpose Jacobian method computes the joint velocities by projecting the operational space error onto the rows of $\mathbf{J}$:

$$\dot{\mathbf{q}} = \mathbf{J}^T\mathbf{e}. \tag{5.1}$$

This approach can be interpreted as a gradient descent method, where the end-effector moves along the direction of maximum decrease in the error. While less computationally expensive than the inverse Jacobian method, it may converge more slowly.

5.2 Trajectory Planning and Execution

The trajectory was planned in operational space using waypoints, including both linear and circular segments. At each time step:

1. The desired position of the end-effector, $\mathbf{x}_{\text{desired}}(t)$, was determined based on the planned trajectory.
2. The error, $\mathbf{e}$, was computed between $\mathbf{x}_{\text{desired}}(t)$ and the actual position, $\mathbf{x}_{\text{current}}(t)$, obtained via forward kinematics.
3. The joint velocities,$\dot{\mathbf{q}}$, were calculated using the chosen kinematic inversion method (inverse or transpose Jacobian).

4. The joint variables, $\mathbf{q}(t)$, were updated using Euler's integration rule:

$$\mathbf{q}(t + \Delta t) = \mathbf{q}(t) + \Delta t\, \dot{\mathbf{q}}(t). \tag{5.2}$$

5.2.1 Explanation of the Pathway and Euler Angle ϕ

In the planned trajectory for the SCARA robot, the path was defined exclusively in terms of the end-effector's position coordinates (x, y, and z) in operational space. The rotation of the end-effector, represented by the Euler angle ϕ, was not incorporated into the pathway definition. This means that while the robot follows the predefined path in 3D Cartesian space, the rotational degree of freedom (ϕ) remains undefined and does not vary as the end-effector moves along the trajectory.
The complete operational space for the robot end-effector typically includes both position (x, y, z) and orientation (ϕ, θ, ψ, where ϕ is the roll, θ is the pitch, and ψ is the yaw).

5.2.2 Consequences of Not Defining ϕ

1. Relaxation of Operational Space:
By not explicitly defining ϕ, we implicitly allow the end-effector's rotation to vary freely. This relaxes the constraints in the operational space, leading to reduced control over the robot's behavior in the rotational dimension.

2. Potential Inconsistencies:
If the application requires the end-effector to maintain a specific orientation while moving along the trajectory (e.g., to grip or align with objects), not defining ϕ could result in unintended rotations, causing task failures.

3. Inefficient Use of Robot Degrees of Freedom (DoF):
For a 4-DoF SCARA robot, failing to utilize ϕ means only 3 of the 4 DoFs are actively contributing to the trajectory. This underutilizes the robot's capabilities and limits its functionality.

5.2.3 Explanation of Including q_4 (ϕ) in the Operational Space

To incorporate ϕ into the pathway:

1. Define $\phi(t)$: Manually specify a variation for ϕ as the robot moves along the trajectory. For example:

$$\phi(t) = \phi_0 + \omega t, \tag{5.3}$$

where ϕ_0 is the initial angle, and ω is the angular velocity.
2. Modify the Error Vector: Include ϕ in the error calculation:

$$\mathbf{e} = \mathbf{x}_{\text{desired}} - \mathbf{x}_{\text{current}}, \tag{5.4}$$

where:

$$\mathbf{x}_{\text{desired}} = [x_{\text{desired}}, y_{\text{desired}}, z_{\text{desired}}, \phi_{\text{desired}}]^T. \tag{5.5}$$

3. Update the Jacobian Matrix:
Retain the rotational component of the Jacobian ($\mathbf{J}_{:,4}$) to calculate the effect of ϕ on joint velocities.
4. Logic Behind the Code To include ϕ in the operational space, we introduced a predefined variation for q_4 as the robot moved along the x, y, z trajectory. The logic was implemented as follows:

The joint velocity for q_4 was explicitly set to a constant rate (ω) in the loop:

```
% Define a target range for theta4 (q4)
theta4_start = 0;          % Start angle for q4
theta4_end = 2 * pi;       % End angle for q4
theta4_rate = (theta4_end - theta4_start) / (length(t) - 1); % Increment per time step

```

Figure 5.1: Inclusion of Euler Angle to the Operational Space

3. Integration with the Full Operational Space:
The error vector ($\mathbf{e}$) was extended to include ϕ, ensuring the Jacobian accounted for the rotational component:

$$\mathbf{e} = \begin{bmatrix} x_{\text{desired}} - x_{\text{current}} \\ y_{\text{desired}} - y_{\text{current}} \\ z_{\text{desired}} - z_{\text{current}} \\ \phi_{\text{desired}} - \phi_{\text{current}} \end{bmatrix}. \tag{5.6}$$

4. Updated Operational Space:
The operational space was expanded to:

$$\mathbf{x} = [x, y, z, \phi]^T,$$

and the Jacobian matrix included the fourth column for ϕ.
The full operational space velocity equation is:

$$\dot{\mathbf{x}} = \mathbf{J}\dot{\mathbf{q}},$$

where:

$$\mathbf{J} = \begin{bmatrix} \mathbf{J}_{\text{pos}} \\ \mathbf{J}_{\text{rot}} \end{bmatrix}.$$

Initially, $\mathbf{J}_{\text{rot}}$ (the rotational part) was ignored. By defining ϕ and including it in the trajectory:

1. The fourth row of the Jacobian was restored to account for rotational velocity.
2. The pseudo-inverse of the Jacobian, $\mathbf{J}^\dagger$, calculated joint velocities considering both positional (x, y, z) and rotational (ϕ) errors:

$$\dot{\mathbf{q}} = \mathbf{J}^\dagger \mathbf{e}.$$

5.2.4 Advantages of Defining ϕ

1. Complete Utilization of the Operational Space: - Incorporating ϕ ensured that all four degrees of freedom (q_1, q_2, q_3, q_4) contributed to the trajectory, maximizing the robot's functionality.
2. Precise Control of Orientation: - The robot was able to maintain or vary its orientation (ϕ) as required for specific tasks, such as aligning with objects or performing rotational motions.
3. Task Reliability: - For tasks involving manipulation, defining ϕ ensures the end-effector maintains the correct orientation, improving reliability and efficiency.

5.3 Inverse Jacobian

5.3.1 Code Implementation

```
% Waypoints
waypoints = [
    0.6,  0.5, 0.5; % P0 (start)
    0.7,  0.5, 0.6; % P1
    0.7,  0.4, 0.7; % P2
    0.85, 0.35, 0.8; % P3
    0.85, 0.15, 0.9  % P4 (end)
];

% Circular section radius and center
radius = norm(waypoints(5, 1:2) - waypoints(4, 1:2)) / 2;
circle_center = (waypoints(4, 1:2) + waypoints(5, 1:2)) / 2;

% Trajectory parameters
total_time = 40; % Total trajectory time (s)
dt = 0.001; % Integration time step (1ms)
t = 0:dt:total_time; % Time array
n = length(t); % Total number of time steps

% Define trajectory segments based on proportional time splits
segment1 = 1:round(0.25 * n); % 0% to 25%
segment2 = round(0.25 * n) + 1:round(0.5 * n); % 25% to 50%
segment3 = round(0.5 * n) + 1:round(0.75 * n); % 50% to 75%
segment4 = round(0.75 * n) + 1:n; % 75% to 100%

% Generate trajectory
trajectory = zeros(length(t), 3);
trajectory(segment1, :) = linear_trajectory(waypoints(1, :), waypoints(2, :), 0, 10, t(segment1));
trajectory(segment2, :) = linear_trajectory(waypoints(2, :), waypoints(3, :), 10, 20, t(segment2));
trajectory(segment3, :) = linear_trajectory(waypoints(3, :), waypoints(4, :), 20, 30, t(segment3));
trajectory(segment4, :) = circular_trajectory(waypoints(4, :), waypoints(5, :), circle_center, radius, 30, 40, t(segment4));
```

```
% Initialize joint variables
q = zeros(4, length(t)); % Assume 4 DOF for SCARA
q(:, 1) = [0; 0; 0.5; 0]; % Initial guess for joint variables

% Define a target range for theta4 (q4)
theta4_start = 0;          % Start angle for q4
theta4_end = 2 * pi;       % End angle for q4
theta4_rate = (theta4_end - theta4_start) / (length(t) - 1); % Increment per time step

% Perform Euler integration
for i = 1:length(t)-1
    % Current end-effector position
    p_current = forward_kinematics(q(:, i));

    % Desired end-effector position
    p_desired = trajectory(i + 1, :);

    % Compute error
    error = p_desired - p_current';

    % Compute Jacobian
    J = compute_jacobian(q(:, i));

    % Inverse Jacobian method
    q_dot_inv = pinv(J) * error'; % Use only the Cartesian position error

    % Update q4 explicitly with a linear variation
    q_dot_inv(4) = theta4_rate; % Add a constant rate of change for q4

    % Update joint variables
    q(:, i + 1) = q(:, i) + dt * q_dot_inv;
end
```

```
% Plot trajectory and joint variables
figure;
subplot(2, 1, 1);
plot3(trajectory(:, 1), trajectory(:, 2), trajectory(:, 3), 'r');
grid on;
xlabel('X'); ylabel('Y'); zlabel('Z');
title('Planned Trajectory in Cartesian Space');

subplot(2, 1, 2);
plot(t, q');
grid on;
xlabel('Time (s)');
ylabel('Joint Variables');
legend('q1', 'q2', 'q3', 'q4');
title('Joint Variables Over Time');
```

```
%% Functions

% Forward kinematics for SCARA
function p = forward_kinematics(q)
    a1 = 0.5; a2 = 0.5; % Link lengths
    d0 = 0.5;           % Base offset
    theta1 = q(1); theta2 = q(2); d3 = q(3);

    x = a1 * cos(theta1) + a2 * cos(theta1 + theta2);
    y = a1 * sin(theta1) + a2 * sin(theta1 + theta2);
    z = d0 + d3;
    p = [x; y; z];
end

% Jacobian for SCARA
function J = compute_jacobian(q)
```

```
% Jacobian for SCARA
function J = compute_jacobian(q)
    a1 = 0.5; a2 = 0.5; % Link lengths
    theta1 = q(1); theta2 = q(2);

    J = [
        -a1 * sin(theta1) - a2 * sin(theta1 + theta2), -a2 * sin(theta1 + theta2), 0, 0;
         a1 * cos(theta1) + a2 * cos(theta1 + theta2),  a2 * cos(theta1 + theta2), 0, 0;
         0, 0, 1, 0
    ];
end

% Linear trajectory
function traj = linear_trajectory(p_start, p_end, t_start, t_end, t)
    alpha = (t - t_start) / (t_end - t_start);
    alpha(alpha < 0) = 0; alpha(alpha > 1) = 1;
    traj = (1 - alpha') .* p_start + alpha' .* p_end;
end

% Circular trajectory
function traj = circular_trajectory(p_start, p_end, center, radius, t_start, t_end, t)
    alpha = (t - t_start) / (t_end - t_start);
    alpha(alpha < 0) = 0; alpha(alpha > 1) = 1;
    theta_start = atan2(p_start(2) - center(2), p_start(1) - center(1));
    theta_end = atan2(p_end(2) - center(2), p_end(1) - center(1));
    theta = theta_start + alpha' * (theta_end - theta_start);
    traj = [center(1) + radius * cos(theta), center(2) + radius * sin(theta), linspace(p_start(3), p_end(3), length(t))'];
end
```

5.3.2 Results

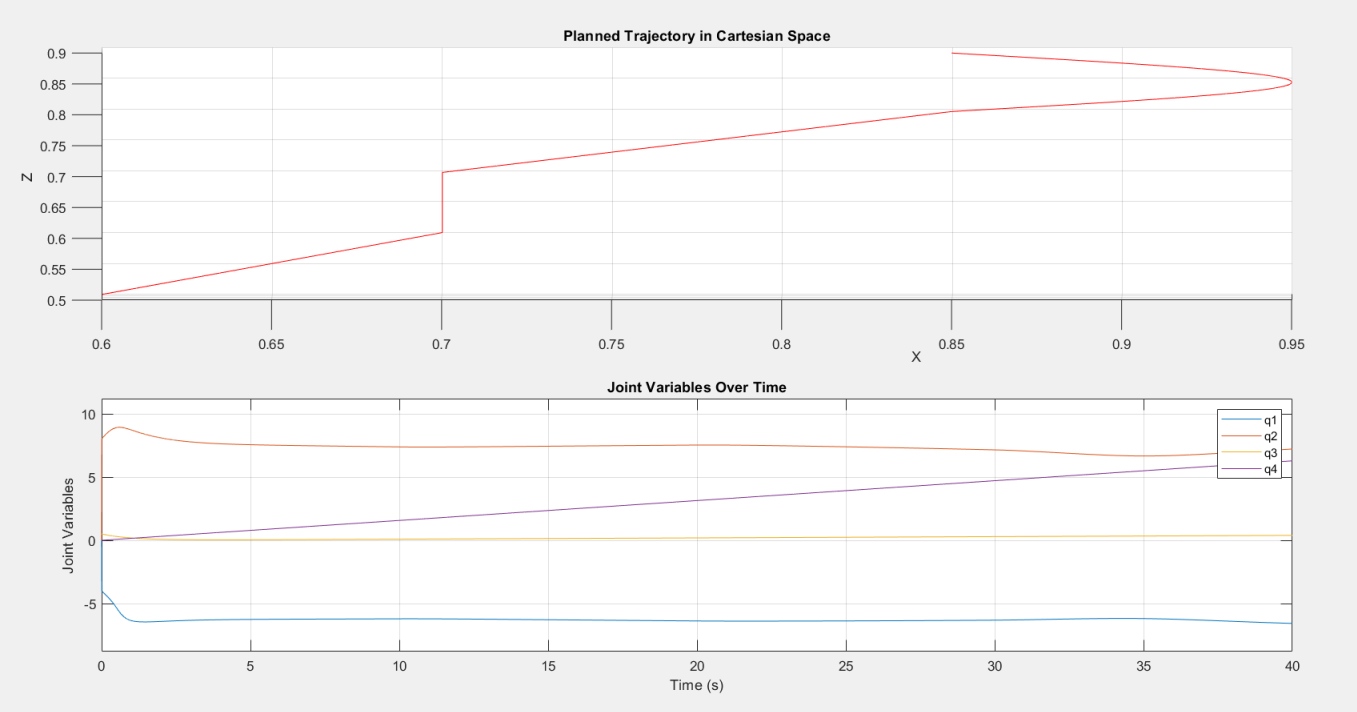

Figure 5.2: Kinematic Inversion using Inverse Jacobian Method 1

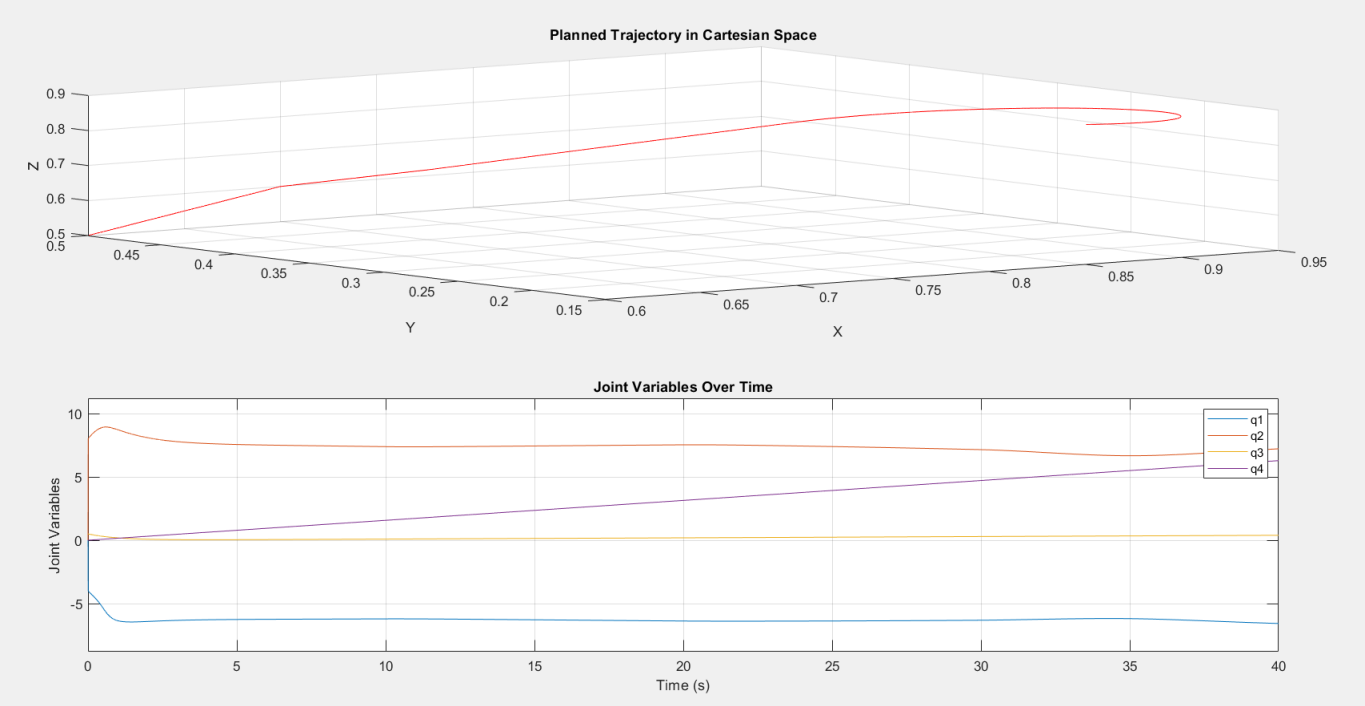

Figure 5.3: Kinematic Inversion using Inverse Jacobian Method 2

5.4 Transpose Jacobian

Transpose Jacobian method for kinematic inversion, is an alternative to the inverse Jacobian method. It is particularly useful when computational simplicity is preferred or when a direct inverse of the Jacobian is not feasible.

5.4.1 Code Implementation

```
61
62          % Transpose Jacobian method
63          q_dot_transpose = J' * error'; % Only Cartesian error is used
64
65          % Use inverse Jacobian for integration (can switch to q_dot_transpose)
66          q(:, i + 1) = q(:, i) + dt * q_dot_transpose;
67      end
```

1. Error Computation:
The error vector ($\mathbf{e}$) is computed as the difference between the desired and current end-effector positions in Cartesian space:

$$\mathbf{e} = [x_{\text{desired}} - x_{\text{current}},\ y_{\text{desired}} - y_{\text{current}},\ z_{\text{desired}} - z_{\text{current}}]^T. \tag{5.7}$$

2. Joint Velocity Calculation:
The transpose of the Jacobian (J^T) is multiplied by the error vector to compute the joint velocities ($\dot{\mathbf{q}}$) . In matlab:

```
q_dot_transpose = J' * error';
```

The transpose Jacobian essentially distributes the operational space error across the joints, guiding the robot toward the desired trajectory.

3. Euler Integration: - The joint variables are updated using the Euler integration rule:

$$\mathbf{q}(t + \Delta t) = \mathbf{q}(t) + \Delta t\, \dot{\mathbf{q}}. \tag{5.8}$$

In the code:

```
q(:, i + 1) = q(:, i) + dt * q_dot_transpose;
```

5.4.2 Results

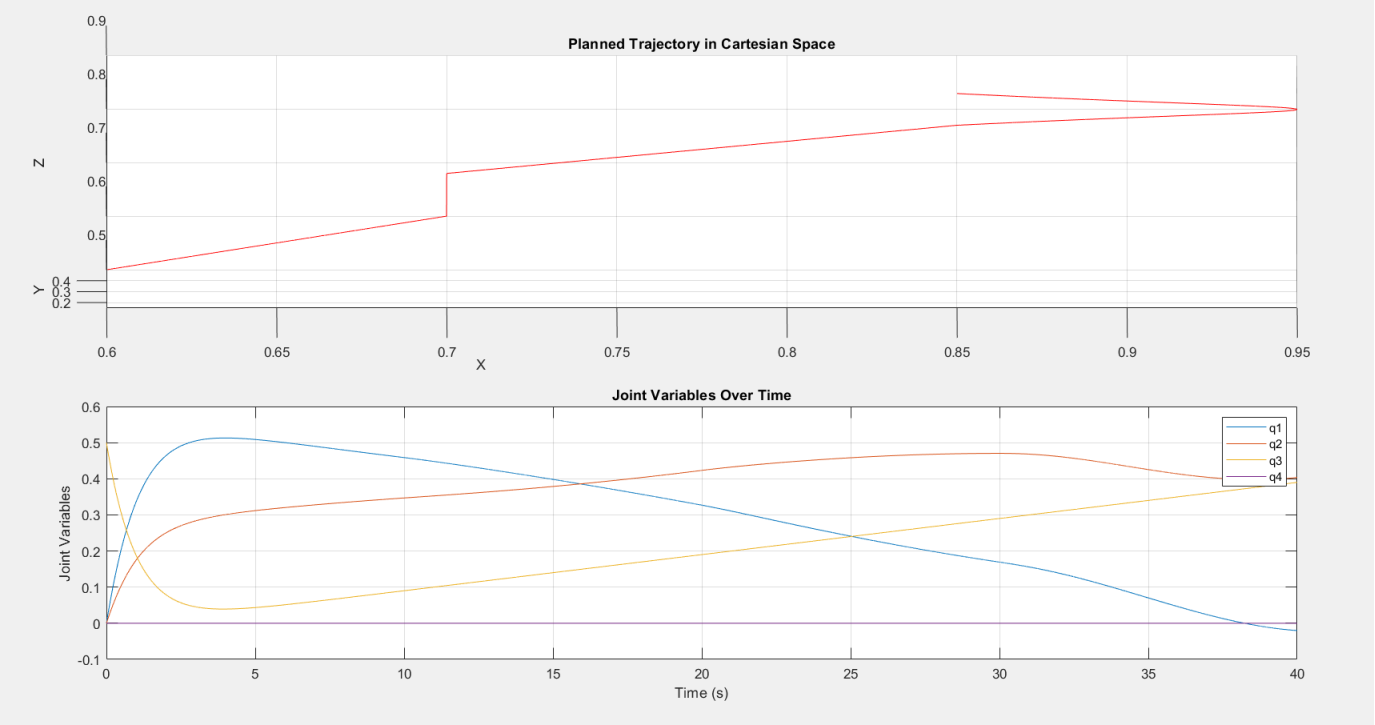

Figure 5.4: Kinematic Inversion using Jacobian Transpose Method 1

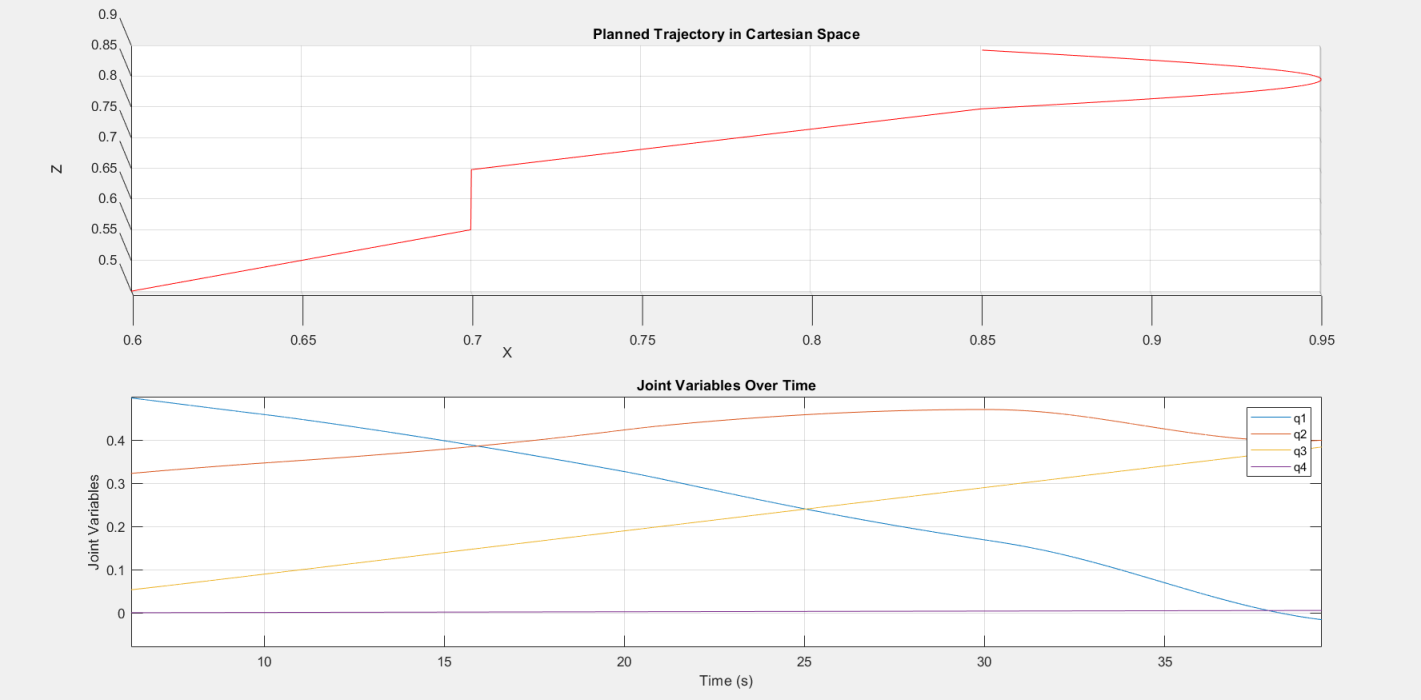

Figure 5.5: Kinematic Inversion using Jacobian Transpose Method 2

5.5 Key Considerations in the Code

Switching Between Methods:
The code includes a flexible structure to switch between the transpose Jacobian and the inverse.

```
58
59      % Inverse Jacobian method
60      q_dot_inv = pinv(J) * error'; % Use only the Cartesian position error
61
62      % Transpose Jacobian method
63      q_dot_transpose = J' * error'; % Only Cartesian error is used
64
```

Figure 5.6: Code for Tranpose and Inverse Jacobian

This modularity allows for comparative analysis of the two approaches.

5.6 Simulink Models

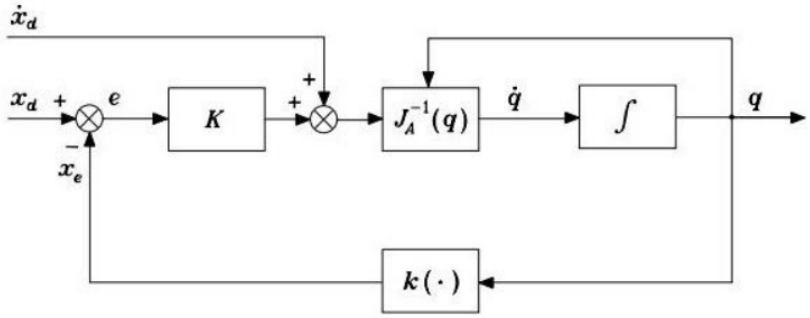

Figure 5.7: Control Model for Inverse Jacobian

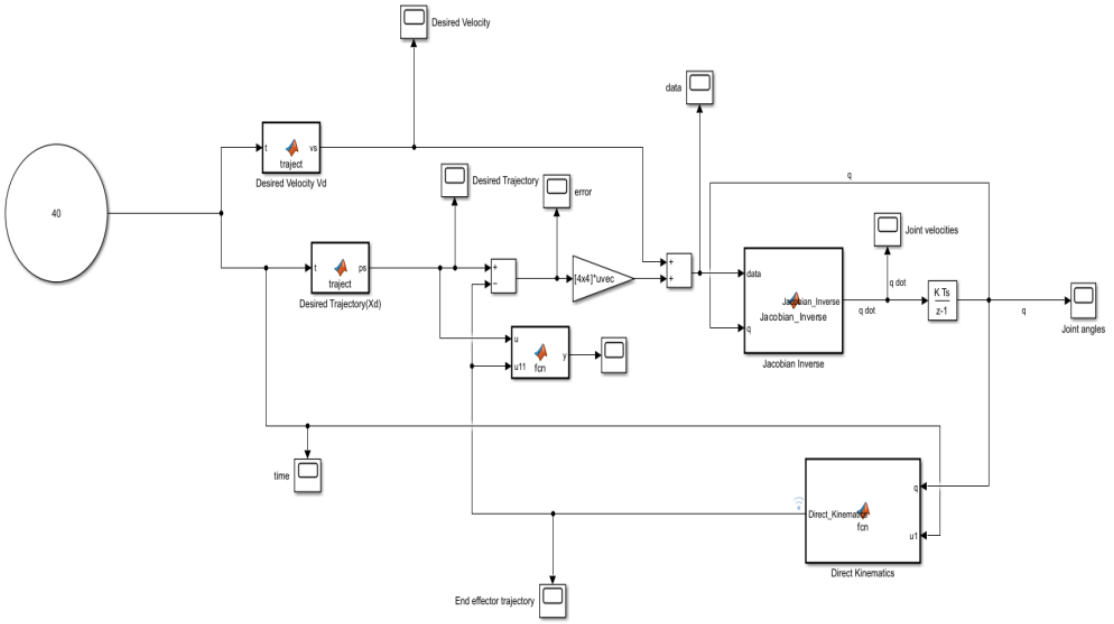

Figure 5.8: Simulink Model for Inverse Jacobian

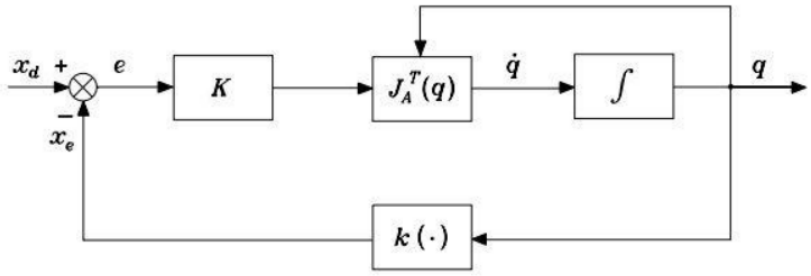

Figure 5.9: Control Model for Transpose of Jacobian

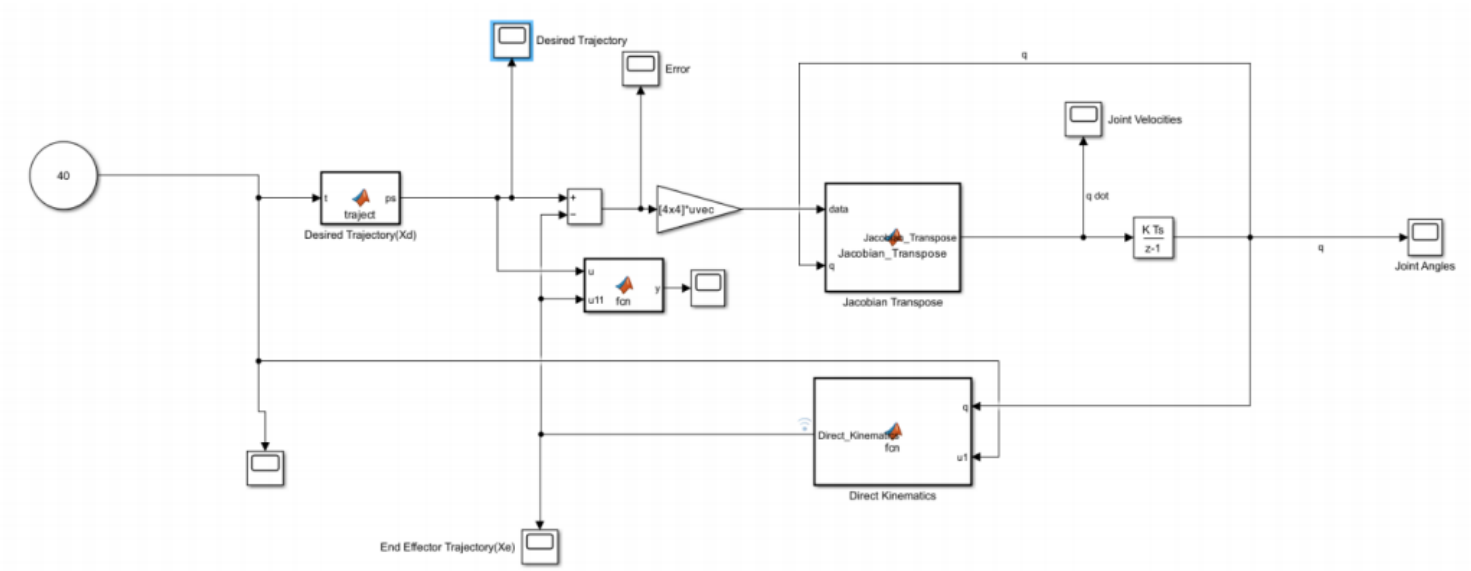

Figure 5.10: Simulink Model for Transpose of Jacobian

5.7 Conclusion

The transpose Jacobian method provides a simple and computationally efficient way to implement kinematic inversion. While it does not guarantee optimal joint movements or exploit redundancies, it is effective for reducing the operational space error along the planned trajectory. By incorporating this method into the code, we demonstrated its practicality and versatility as an alternative to the inverse Jacobian approach.

Chapter 6

Relaxing a Component of Operational Space

I implemented a solution using MATLAB. The objective was to develop an algorithm that handles kinematic inversion while selectively relaxing components of the operational space based on the specified constraints.
By "relaxing," we mean de-prioritizing or reducing the constraints on a specific operational space component, such as orientation (ϕ) or height (z), allowing the robot greater freedom to optimize other parameters like avoiding joint limits or obstacles. The solution used the pseudo-inverse Jacobian to compute the required joint velocities q for tracking the trajectory while adhering to these constraints.
The pseudo-inverse Jacobian, $\mathbf{J}^\dagger$, was used to compute the joint velocities that minimized the error between the desired and current end-effector positions:

$$\dot{\mathbf{q}} = \mathbf{J}^\dagger \mathbf{e}.$$

where:
$\mathbf{e} = \mathbf{x}_{\text{desired}} - \mathbf{x}_{\text{current}}$ is the operational space error.
$\mathbf{J}^\dagger = \mathbf{J}^T(\mathbf{J}\mathbf{J}^T)^{-1}$ is the Moore-Penrose pseudo-inverse of the Jacobian.
This ensures that the end-effector accurately follows the desired trajectory.

6.1 Relaxation of the Orientation Component (ϕ)

When relaxing ϕ, the error and Jacobian contributions for the rotational component were removed. This allowed the robot to prioritize avoiding joint limits over maintaining a specific orientation:

```
% Relax the orientation component (phi)
J(:, 4) = 0; % Remove the contribution of phi from the Jacobian
```

6.2 Relaxation of the z-Component

When relaxing z, the error in the height component was set to zero, allowing the end-effector to adjust its z-position freely to maximize the distance from obstacles: "‘matlab error(3) = 0; "‘

6.3 Maximizing the Distance from Joint Limits

To ensure the robot stayed far from its joint limits, we incorporated a weighting term into the pseudo-inverse calculation. This penalized configurations near the joint limits:

```
q_center = 0.5 * (q_min + q_max); % Midpoint of joint limits
q_range = q_max - q_min; % Range of joint limits
weight = diag(1 ./ (q_range - abs(q(:, i) - q_center)).^2); % Weight matrix
q_dot = pinv(J * weight) * error'; % Weighted pseudo-inverse
```

6.4 Incorporating Obstacles

The obstacle is modelled as a circle in the workspace.

```
% Obstacle_parameters
obstacle_position = [0.75, 0.3, 0.75];
obstacle_radius = 0.5;
```

For obstacle avoidance, we defined a repulsive potential field around the obstacle. The z-component relaxation allowed the robot to adjust its height to avoid collisions.

```
% Introduce obstacle avoidance
if norm(p_current - obstacle_position) < obstacle_radius
    error(3) = 0; % Relax z to avoid obstacle
```

6.5 Code

```
        error = p_desired - p_current';
        % Relax the z-component (remove z-constraint)
        error(3) = 0; % Uncomment to relax z-component

        % Compute Jacobian
        J = compute_jacobian(q(:, i));

        % Modify Jacobian for relaxed orientation component (relax phi)
        J(:, 4) = 0; % Uncomment to relax orientation (phi)

        % Pseudo-inverse of the Jacobian
        q_dot = pinv(J) * error';

        % Maximize distance from joint limits
        q_center = 0.5 * (q_min + q_max);
        q_range = q_max - q_min;
        weight = diag(1 ./ (q_range - abs(q(:, i) - q_center)).^2);
        q_dot = q_dot + weight * (q_center - q(:, i)); % Weighted adjustment

        % Update joint variables using Euler integration
        q(:, i + 1) = q(:, i) + dt * q_dot;
    end

    % Plot trajectory and joint variables
    figure;
    subplot(2, 1, 1);
    plot3(trajectory(:, 1), trajectory(:, 2), trajectory(:, 3), 'r');
    hold on;
    plot3(obstacle_position(1), obstacle_position(2), obstacle_position(3), 'bo', 'MarkerSize', 8, 'LineWidth', 2);
    grid on;
    xlabel('X'); ylabel('Y'); zlabel('Z');
    title('Planned Trajectory in Cartesian Space');

    subplot(2, 1, 2);
    plot(t, q');
    grid on;
    xlabel('Time (s)');
    ylabel('Joint Variables');
    legend('q1', 'q2', 'q3', 'q4');
    title('Joint Variables Over Time');
```

Figure 6.1: Chapter 6 Code

6.6 Results

The blue dot in the graph below is the obstacle.

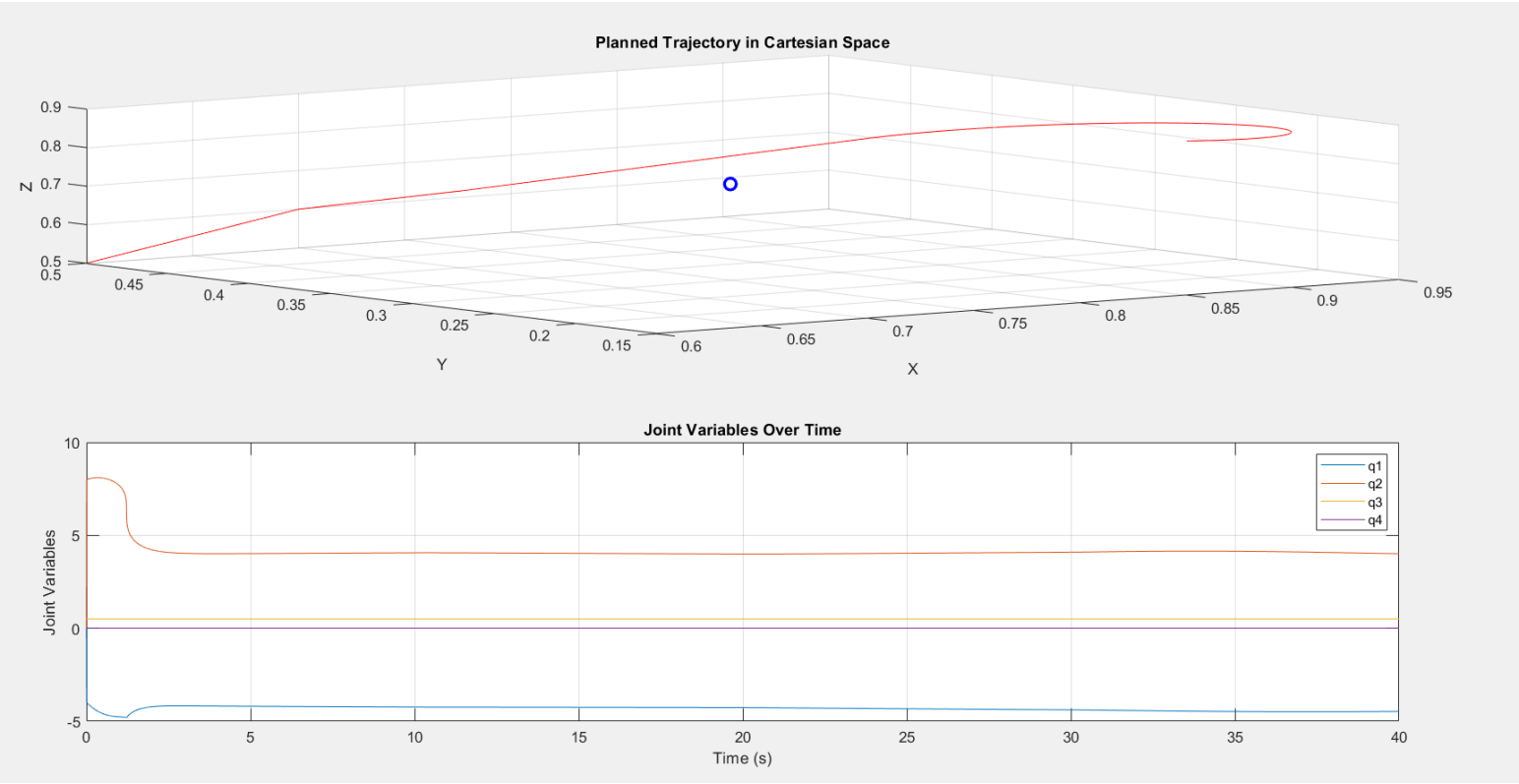

Figure 6.2: Relaxed phi

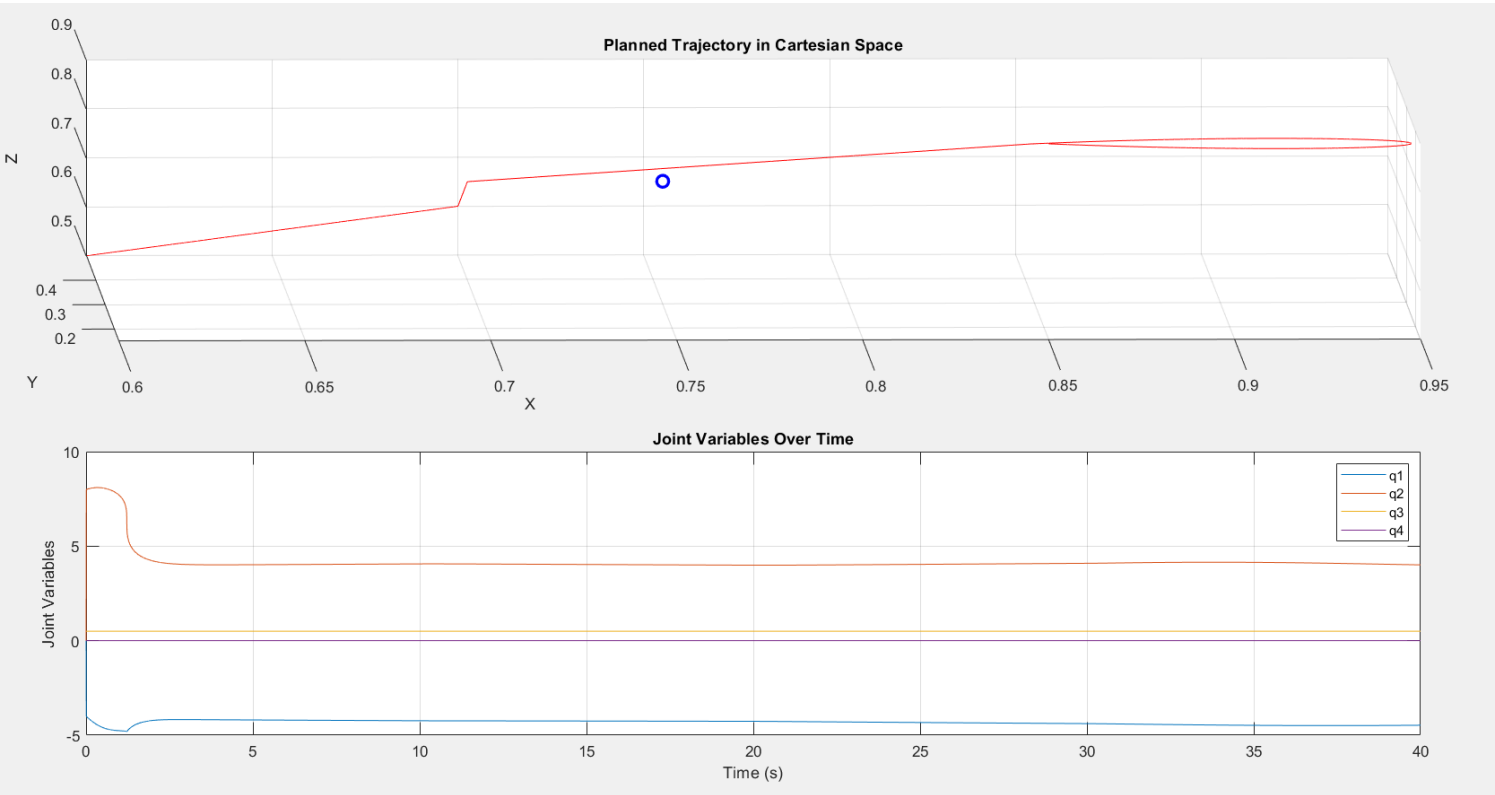

Figure 6.3: Relaxed Z

6.7 Observations

1. Relaxing Orientation (ϕ):
a. Allowed the robot to focus on maintaining distance from joint limits.
b. Orientation was left uncontrolled, providing flexibility in movement.

2. Relaxing z-Component:
a. Enabled the robot to vary its height dynamically to avoid collisions.
b. Sacrificed precise z-position control for obstacle avoidance.

3. Joint Limit Avoidance:
The weighting ensured that joint velocities moved the configuration away from boundaries, increasing safety and operational range.

6.8 Simulink Model

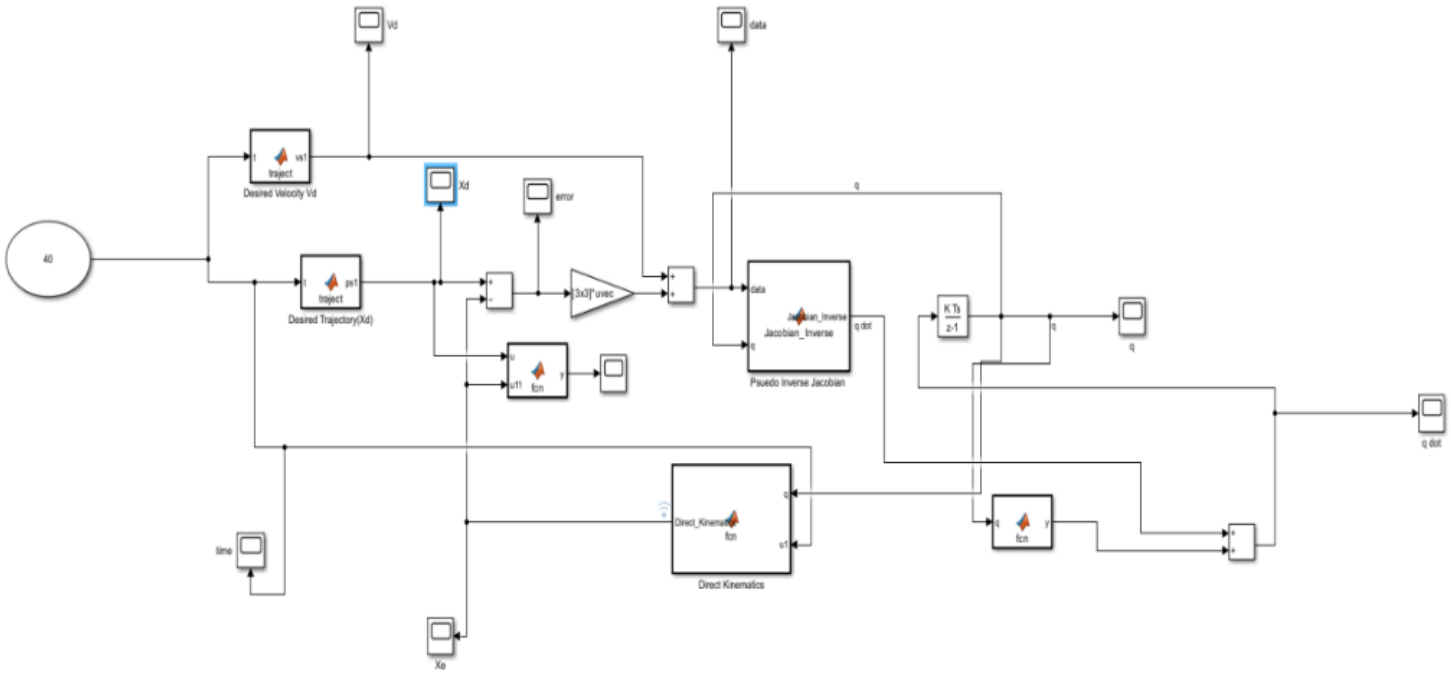

Figure 6.4: Simulink Model of the relaxed phi

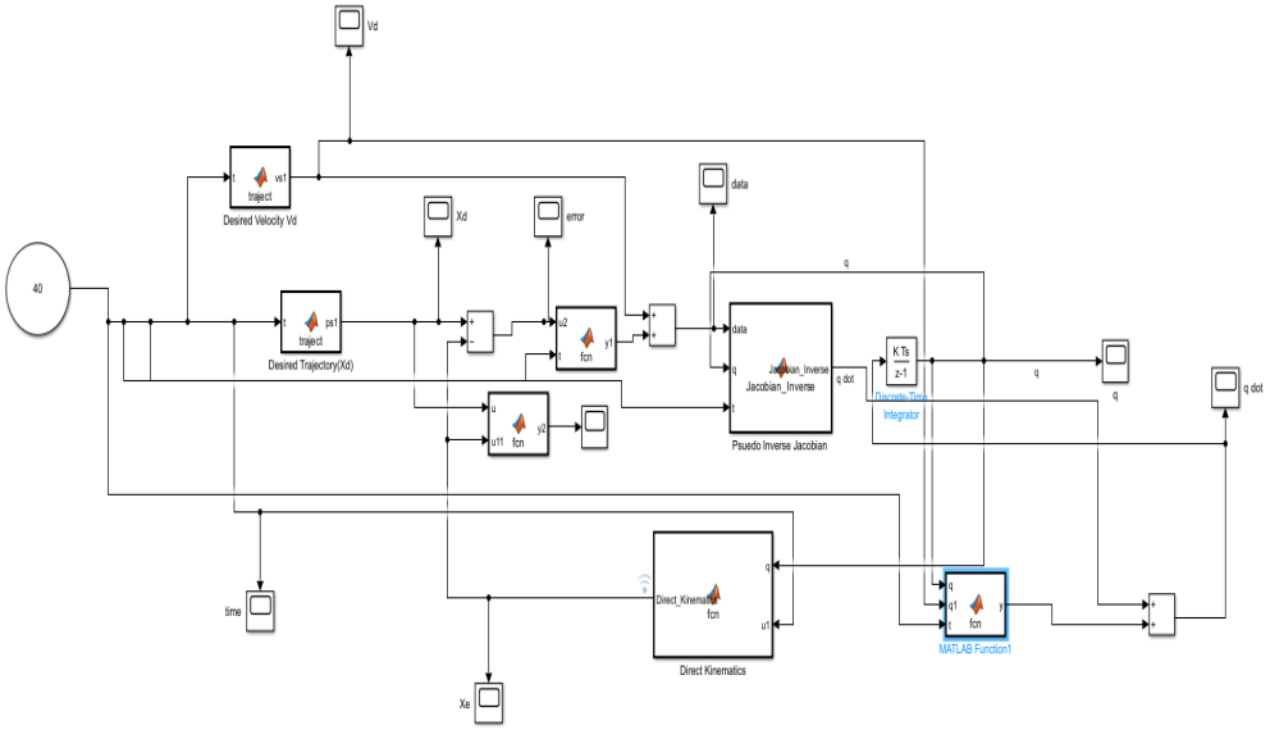

Figure 6.5: Simulink Model of the relaxed Z

6.9 Conclusion

By implementing the pseudo-inverse Jacobian method and strategically relaxing operational space components, we achieved a robust and flexible algorithm for trajectory tracking. Relaxing ϕ or z provided the robot with additional degrees of freedom to optimize performance in specific scenarios, such as maximizing distance from joint limits or avoiding obstacles. These techniques highlight the importance of balancing constraints and priorities in robotic trajectory planning.

References

1. *Robotics Modelling, Planning and Control*, by Bruno Siciliano, Lorenzo Sciavicco, Luigi Villani, Giuseppe Oriolo.

2. *Robotics Toolbox Guide* by Peter Corke.