ME5402/EE5106 ADVANCED ROBOTICS

NATIONAL UNIVERSITY OF SINGAPORE

DEPARTMENT OF MECHANICAL ENGINEERING

Part 1 CA Robot Simulation

Author:

Niu Dixiao (A0284913X)

Ming Hao (A0285252A)

Jiang Cenxi (A0289684A)

Lecturer

Prof. CHUI Chee Kong

April 15, 2024

Abstract

With the development of artificial intelligence and machine learning technology, robotic arms have begun to incorporate these advanced technologies to achieve more complex operations and a higher level of automation. By establishing an accurate robotic arm model, we can improve the operating accuracy and efficiency of the robotic arm and further expand its applications in manufacturing, medical, service and other fields.

The robotic arm is an important component in the field of automation and robotics. It provides theoretical basis and technical support for the control and operation of the robotic arm. Robotic arm models usually include dynamics modeling, kinematics modeling and trajectory tracking design. In this paper, commonly used methods include the Denavit-Hartenberg (DH) parameter method, which simplifies the description of the geometric relationship between the manipulator joints and links in a systematic manner, thereby achieving accurate calculation of the manipulator configuration. The dynamic model involves the calculation of forces and moments, providing the necessary dynamic characteristics analysis for precise control of the robotic arm. In addition, the design of the control strategy is the key to ensuring that the robotic arm can complete the intended task, involving path planning, attitude adjustment, response speed and other aspects. In the experimental part, the trajectory tracking of eight preset points was tested through simulation and actual operation. In actual operation, the robotic arm was able to accurately reach every predetermined point, which verified the effectiveness and practicability of the model.

Contents

Chapter 1

Introduction

1.1 Development status

With the development of science and technology and the rapid development of social economy, machines are gradually becoming an important source of replacement for manual labor. In the fields of automation control, industrial manufacturing, aerospace and other fields, it has gradually penetrated into people's daily life, and also occupied a place in medical care, food manufacturing, entertainment, services, etc. There is no doubt that robotic arms are gradually improving people's quality of life.

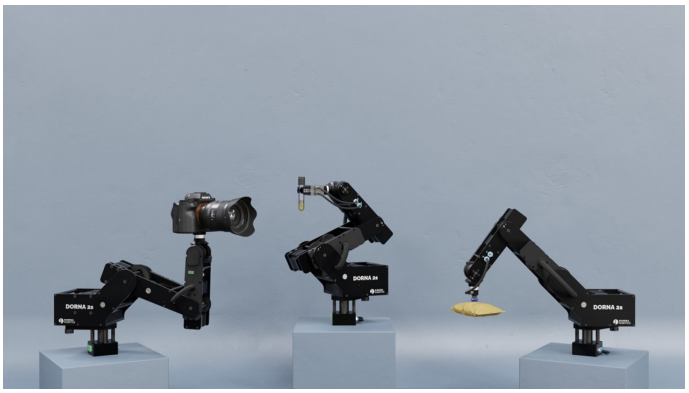


Figure 1.1: Robotic arm

1.2 Application areas

For factories in the manufacturing industry, they only need to use programs to execute codes, and then they can hand over repetitive or dangerous tasks to robotic arms. These

tasks can save a lot of human resources and replace mechanical repetitive work. At the same time, a basic completion rate can be guaranteed. For the aerospace field, space missions

Figure 1.2: Robotic arm in manufacturing industry

can be performed under extreme conditions during the aircraft and during operation. Such as the construction and maintenance of the space station, deployment, maintenance and repair of satellites, assisting astronauts in space walks and repair tasks, and operating extravehicular equipment. It can also be used to assemble and repair spacecraft, repair damaged sky telescopes or satellites.

Figure 1.3: Robotic arm in aerospace field

Robotic arms are also widely used in medical treatment. Robotic arms have a variety of roles and applications in the medical field. Robotic arms can assist doctors in performing precise, minimally invasive surgeries. Through the robotic arm, doctors can perform more precise operations during surgery, reducing surgical risks and complications. It can also be used for rehabilitation treatment to help patients perform sports rehabilitation training. Robotic arms are also used to operate medical equipment. Robotic arms can help doctors accurately locate the location required for surgery.

Figure 1.4: Robotic arm in medical treatment

1.3 Mainly applied technologies

The robotic arm can achieve basic acquisition, movement, transportation of targets, trajectory tracking and other actions, and has a wide range of application prospects. At present, neural networks are gradually leading future development, and there are also certain research results on the application of machine learning in path planning of robotic arms. Google's Deep Mind team improved the DQN (Deep Q-Network) method and proposed a method with a competitive network structure. Gu et al. proposed a normalized advantage function for continuous control problems. In order to solve the high-dimensional path planning problem, Prianto et al. proposed a path planning algorithm based on Soft Actor-Critic. Finn et al. of the University of California used inverse optimal control to train robotic arm path planning. Since imitation learning was proposed, it has also been used for robotic arm path planning. By combining it with deep learning, the optimal strategy for path planning can be obtained through training.

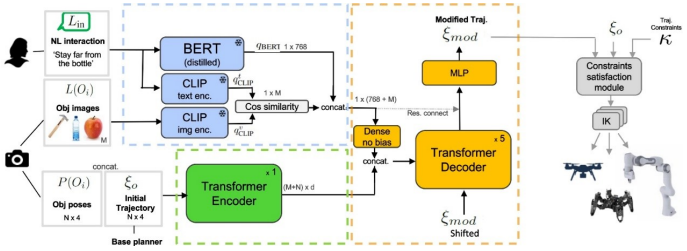


Figure 1.5: Robotic arm controlled by neural network

Currently, there are many simulation platforms for robotic arm research, among which ROS (Robot Operating System) and Adams are two commonly used platforms.

ROS (Robot Operating System): ROS is an open source robot operating system that provides a series of tools, libraries and packages for building robot applications. ROS has powerful simulation functions and supports the control and path planning of robotic arms.

Adams: Adams is a commercial multi-body dynamics simulation software developed by MSC Software. It is mainly used to simulate and analyze the dynamic behavior of mechanical systems. Adams can model complex mechanical systems and simulate them under various working conditions. In robotic arm research, it can be used for design optimization, kinematic analysis and dynamics simulation.

Each platform has its own characteristics. ROS is an open source and flexible system suitable for rapid prototyping and experimentation. Adams encapsulates a commercial-level simulation software with powerful dynamic analysis and simulation capabilities, suitable for detailed modeling and analysis of complex mechanical systems.

MATLAB can also perform some basic simulation operation steps, and has a variety of built-in functions and algorithms for calculating trajectories for operator use.

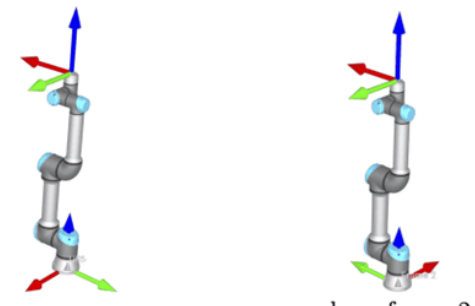


Figure 1.6: Robotic arm controlled by neural network

Regarding how to solve the problem of robotic arm path planning more efficiently, reliably

and quickly, researchers continue to apply cutting-edge technologies to current research content and have achieved certain results. These cutting-edge methods and advanced technologies are changing the future of mankind step by step. What this article will do is to do the basic work before that, build a basic UR5e robotic arm model, and simulate trajectory planning, so as to better prepare for future research.

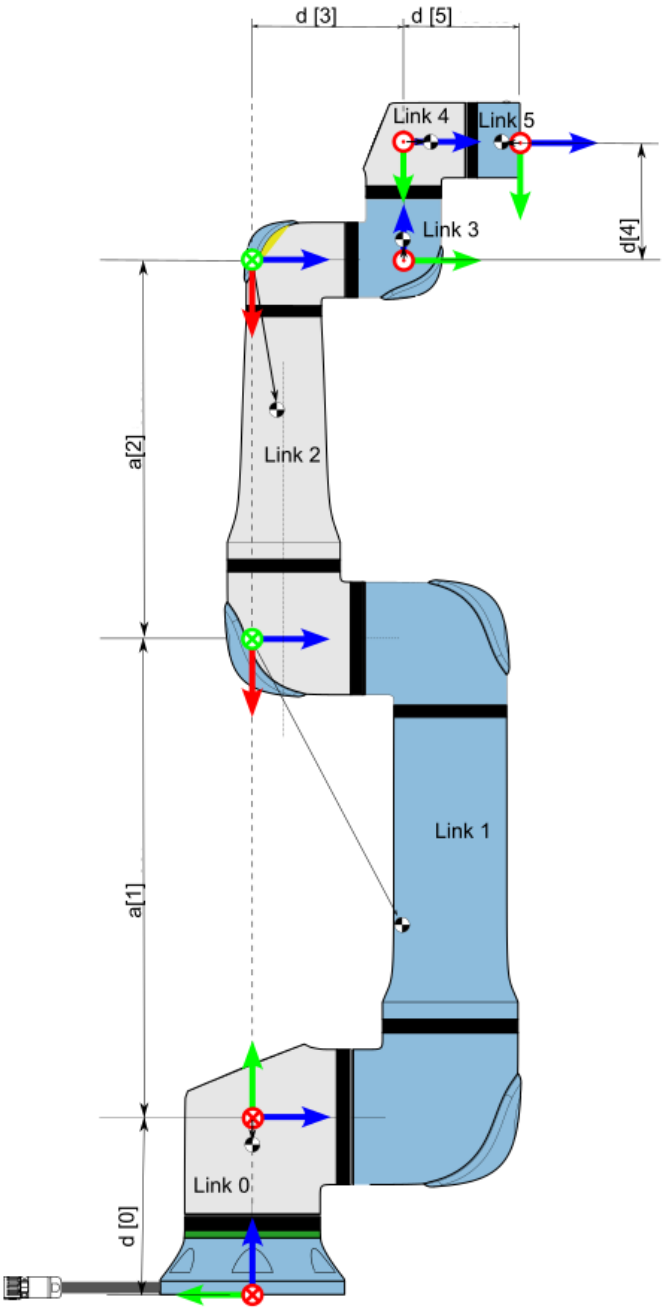


Figure 1.7: Robotic arm modeling

Determining DH Parameters: First, it is necessary to determine the DH parameters based on the geometric structure and joint configuration of the robotic arm. DH parameters

include the distance between the rotation axes of adjacent joints, the rotation angle between joints, the distance between adjacent joint axes, and the possible displacement when connecting adjacent joints .

Next, define joint symbols: In MATLAB, it is essential to define symbolic variables for each joint. For example, use syms to define symbolic joint angle variables.

Then, create a robotic arm object: Use the robotic arm object class provided by the Robotics Toolbox to create a robotic arm object. When creating the object, specify the DH parameters of the robotic arm.

Subsequently, set the range of motion for the robotic arm: The range of motion for each joint can be set based on the actual construction and limitations of the robotic arm.

Then, perform forward kinematics: Use the robotic arm object to perform forward kinematics calculations, i.e., calculate the position and orientation of the end effector based on the given joint angles.

Finally, perform inverse kinematics: Inverse kinematics calculations can be performed using the robotic arm object to compute the required joint angles based on the given position and orientation of the end effector.

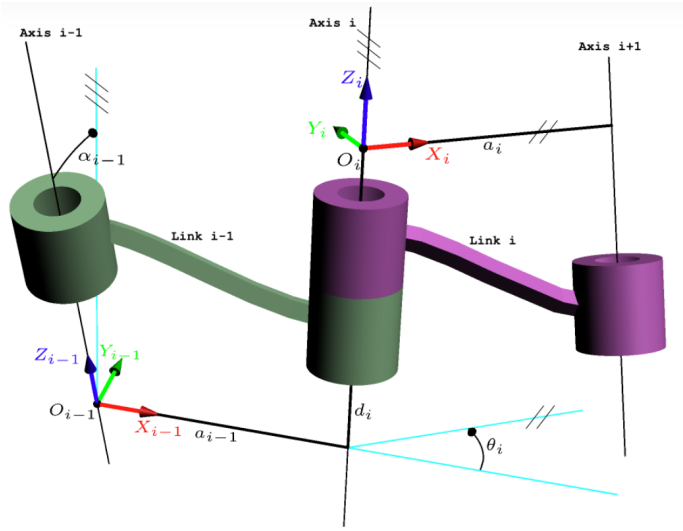


Figure 1.8: DH modeling diagram

To express the transformation relationship between two coordinate systems, a rotation matrix R and a displacement matrix P are usually required, with a total of 12 parameters. However, for conventional manipulators, as long as the following two constraints are met, the transformation relationship of the coordinate system can be clearly expressed with four parameters.

The two constraints of the DH method are:

The x-axis of the latter joint coordinate system should be perpendicular to the z-axis of the previous coordinate system.

The x-axis of the latter joint coordinate system must intersect the z-axis of the previous coordinate system.As shown below.

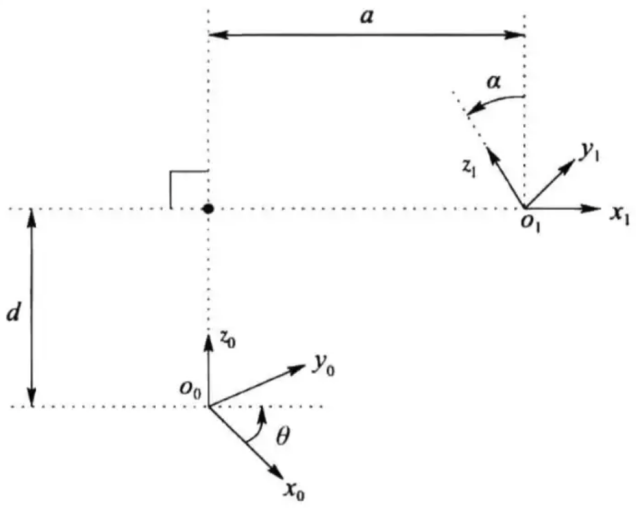


Figure 1.9: Schematic diagram

Import the robotic arm into matlab using the DH modeling method to build a basic model.

- a**(Link length)**: The vertical distance from the Z axis of the previous joint to the Z axis of the current joint.
- α**(Connecting rod torsion angle)**: The rotation angle from the Z axis of the previous joint to the Z axis of the current joint around the X axis of the previous joint.
- d **(Link offset)**: The distance from the X-axis of the previous joint to the X-axis of the current joint along the Z-axis of the previous joint.
- θ**(Joint angle)**: The rotation angle from the X-axis of the previous joint to the X-axis of the current joint around the Z-axis of the previous joint.

Based on the defined DH parameters, the coordinate transformation matrix from one joint to the next can be calculated. This matrix is obtained through the following four transformations:

Rotate around x-axis by α angle

$$R_{x,\alpha} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha & 0 \\ 0 & \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Translate α distance along the x-axis

$$T_{x,a} = \begin{bmatrix} 1 & 0 & 0 & a \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Translate d distance along the z-axis

$$T_{z,d} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & d \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotate around the z-axis by θ angle

$$R_{z,\theta} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

By combining these four transformations, the complete transformation matrix from one joint to the next can be obtained:

$$T = T_{x,\alpha} \cdot T_{x,a} \cdot T_{z,d} \cdot R_{z,\theta} \tag{1.1}$$

The DH modeling method is very useful in robotic arm control and path planning. By

using these DH parameters, engineers can calculate the global position and orientation of any joint of the robotic arm, and then accurately control the robotic arm to perform precise actions, such as welding, assembly, painting, etc.

DH parameter modeling not only enhances the operability of the robotic arm, but also simplifies the design and analysis process of complex mechanical systems, making it more intuitive and easy to understand. Therefore, despite the existence of some alternative methods, DH modeling is still one of the cornerstones of learning and applying robot kinematics.

Chapter 2

Forward Kinematics

This project uses UR5e as the manipulator. It is a robotic arm with 6 revolute joints with a reach of 0.85 meter, which makes it flexible enough to perform the task of piercing. All the joints have a working range of 360 degrees. The force accuracy of the manipulator is around 4N. The movements of the arm are achieved by Robotics System Toolbox of MATLAB.

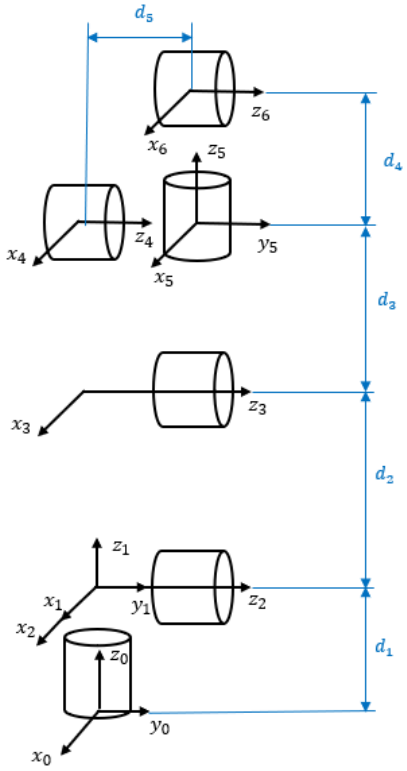


Figure 2.1: A simplified diagram of UR5e joints

To describe the kinetics of the manipulator, the Denavit-Hartenberg (D-H) parameters are used to show how the joints are related to each other. The plot below shows how each parameter is chosen, where link length a is the distance between z-axes along the common normal line. Link twist alpha is the angle between joint axes and joint angle theta is the angle between two subsequent common normal lines. Link offset d is the distance between two consecutive common normal lines.

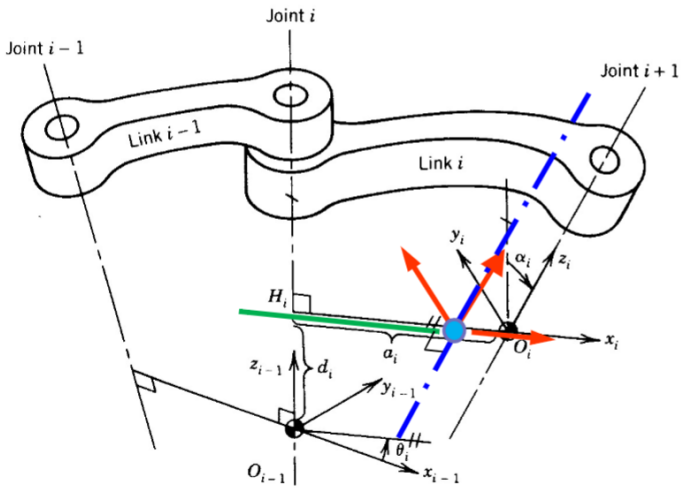


Figure 2.2: A plot that shows how each D-H parameter is defined [1].

By using the technical specification of UR5e that lists the exact dimensions of the manipulator, the D-H parameter table can be easily formulated.

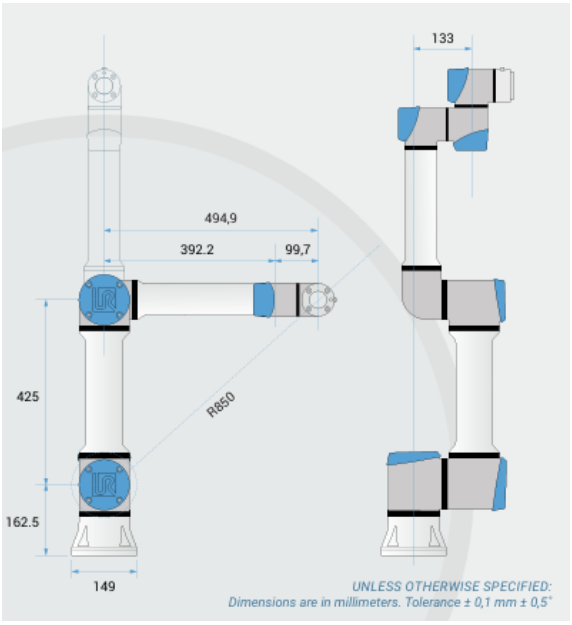


Figure 2.3: Technical drawings of UR5e in different views.

Joints	θ (rad)	α (rad)	a (m)	d (m)
1	θ_1	π/2	0	0.1625
2	θ_2	0	-0.425	0
3	θ_3	0	-0.3922	0
4	θ_4	π/2	0	0.1333
5	θ_5	-π/2	0	0.0997
6	θ_6	0	0	0.0996

Table 2.1. D-H parameter table of the UR5e manipulator

After obtaining the D-H table, a 4 by 4 matrix between two consecutive frames can be calculated based on the equation:

$$ {}^{i-1}_{i}A = \begin{bmatrix} cos\theta_i & -sin\theta_i cos\alpha_i & sin\theta_i sin\alpha_i & a_i cos\theta_i \\ sin\theta_i & cos\theta_i cos\alpha_i & -cos\theta_i sin\alpha_i & a_i sin\theta_i \\ 0 & sin\alpha_i & cos\alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.1} $$

This transformation matrix shows how frame i is relative to frame i-1. There are a few intermediate steps to obtain this result. The frame i-1 is first rotated about z-axis by an angle theta. Then the frame is moved by a distance of value d in the z-axis. Next, the frame travels a distance a in the x-direction. Finally, it is rotated by a angle alpha around x-axis. By multiplying all the matrices, the overall transformation matrix T can be obtained. This matrix shows the locations of the end effect relative to the base frame as a function of the joint angles.

$$ {}^{0}_{6}T = {}^{0}_{1}A(\theta_1){}^{1}_{2}A(\theta_2){}^{2}_{3}A(\theta_3){}^{3}_{4}A(\theta_4){}^{4}_{5}A(\theta_5){}^{5}_{6}A(\theta_6) \tag{2.2} $$

The pseudo-code of Foward Kinematics in Appendix (Foward Kinematics Solver). shows how to calculate the transformation matrix T based on D-H parameters.

Chapter 3

Inverse Kinematics

Inverse Kinematics (IK) is an important concept in robotics that is primarily concerned with calculating the required angles of the robot's joints, as opposed to positive kinematics, given the target position and orientation of an end-effector (e.g., a hand grasp of a robot arm). Positive kinematics is the calculation of end-effector position and orientation with known joint angles. IK is particularly critical when implementing complex movement control and dynamic environment interaction, where IK algorithms can be used to efficiently control the robot to move in a natural and expected manner to meet specific task requirements.

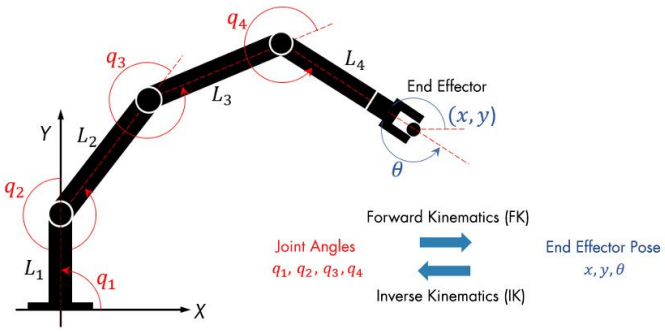


Figure 3.1: Inverse Kinematics Schematic

3.1 Methodology for Inverse Kinematic

There are usually two main types of methods for IK solution: analytical and numerical methods. In terms of solution methods, analytical methods may vary depending on the structure of the robotic arm, whereas numerical methods can usually have a uniform

solution. Generally, we need to choose the appropriate solution method according to the scenario of the robotic arm.

3.1.1 Analytical Methods

Analytical methods use mathematical formulas in IK to directly solve for the angles of robot joints. This method usually relies on the specific geometry of the robot arm to derive a closed-form solution. A closed-form solution is an exact solution that can be obtained by direct computation without iteration or numerical approximation. The advantages of the analytical method are that the computation is fast, the solution is instantly available, and the solution is exact and free from numerical errors. However, the limitation of this method is that closed-form solutions are not available for all robotic arm structures, especially when the robotic arm structure is more complex or has multiple degrees of freedom.

A typical example of an analytic method is for a simple two dimensional planar robot arm (shown in Fig.3.2), which is assumed to consist of two rotational joints and two segments of connecting rods of lengths l_1 and l_2, respectively. To get the end of the robot arm to a point(x, y), we can use trigonometric functions to solve for the angles θ_1 and θ_2 of the two joints.

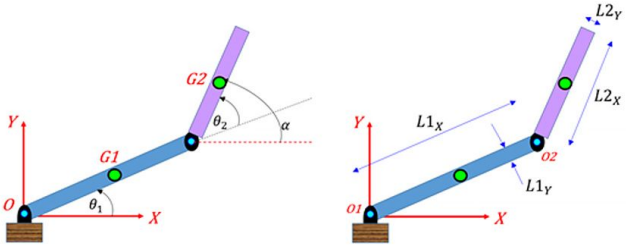


Figure 3.2: Analytical Methods

Taking θ_2 as an example, the cosine theorem shows that:

$$\theta_2 = \cos^{-1}\left(\frac{x^2 + y^2 - l_1^2 - l_2^2}{2l_1 l_2}\right) \tag{3.1}$$

Using the value of θ_2, θ_1 can be calculated as:

$$\theta_1 = \tan^{-1}\left(\frac{y}{x}\right) - \tan^{-1}\left(\frac{l_2 \sin(\theta_2)}{l_1 + l_2 \cos(\theta_2)}\right) \tag{3.2}$$

Note that these formulas assume that the arm can reach the point (x, y) and that there is a real solution to the IK problem.

Analytical methods in IK offer significant advantages in terms of computation speed and accuracy. They can process solutions in microseconds, which is essential for real-time applications, and they provide high precision in determining the necessary joint angles.

However, the main drawback of analytical methods lies in their limited applicability. These methods are not universally applicable to all robotic structures; they work best with specific configurations that satisfy certain criteria, such as the Pieper criterion. This lack of generality means that while they are highly effective for some designs, they cannot be used as a one-size-fits-all solution across different types of robotic systems.

3.1.2 Numerical methods

Numerical methods for IK involve iterative algorithms that incrementally adjust the joint angles to converge to the desired position of the end effector. Common numerical methods include the use of the Jacobian matrix, which represents the partial derivatives of the end effector's position with respect to the joint angles.

One such numerical method is the Jacobian inverse technique, where the change in joint angles $\Delta\boldsymbol{\theta}$ required to achieve a small change in the end effector position $\Delta\mathbf{x}$ is given by:

$$\Delta\boldsymbol{\theta} = \mathbf{J}^{-1}\Delta\mathbf{x} \tag{3.3}$$

Here, $\mathbf{J}^{-1}$ represents the inverse of the Jacobian matrix. However, when the Jacobian is not square or invertible, the pseudoinverse is used, computed as $\mathbf{J}^{+}$. The resulting equation with the pseudoinverse is:

$$\Delta\boldsymbol{\theta} = \mathbf{J}^{+}\Delta\mathbf{x} \tag{3.4}$$

where $\mathbf{J}^{+}$ denotes the Moore-Penrose pseudoinverse of the Jacobian matrix. This method allows for a general approach to solve IK for robotic arms with complex structures and multiple degrees of freedom.

3.2 Inverse Kinematics in MATLAB

As the ur5e is a 6DOF robotic arm, solving its IK through analytical methods is too complex and cannot be automated using a computer. Therefore, numerical methods are used instead.

Specifically, the IK problem is posed as the following optimization problem:

$$\min_{\theta} \|T(\theta) - P_{\text{target}}\| \tag{3.5}$$

where $T(\theta)$ denotes the transformation matrix calculated by the FK function, parameterized by the joint angles θ.

The transformation matrix $T(\theta)$ is then converted into a numeric function via MATLAB's `'matlabFunction'`. This conversion is essential for evaluating the transformation matrix efficiently during optimization. The error function, which quantifies the discrepancy between the current and desired positions of the end-effector, is defined as the norm of the difference between the numeric transformation matrix and P_{target}.

Optimization is performed using MATLAB's fmincon, an algorithm suitable for constrained nonlinear optimization. The initial guesses for the joint angles are set to zeros, and joint limits are specified to ensure physically feasible solutions. The fmincon function minimizes the error function under these constraints, utilizing the Sequential Quadratic Programming (SQP) algorithm. This method iteratively approximates the problem as a quadratic programming problem, solving these approximations to converge on an optimal set of joint angles.

The pseudo-code of IK can be seen in Appendix (Inverse Kinematics Solver).

A simultaneous test was conducted to verify the accuracy of the forward and IK procedure. Firstly, the end positions were determined using forward kinematics based on the angles of each joint. Then, the IK function was used to invert the joint angles and the resulting angles were compared with the initial joint angles.

We set the initial value of a as

$$\theta = [-0.0000,\ 1.1638,\ -0.2425,\ 0.0658,\ -0.0000,\ 0.0601]$$

The transfer matrix T from the initial point to the end point can be derived through

positive kinematics as

$$T = \begin{bmatrix} 0.5000 & -0.8660 & 0 & -0.3222 \\ 0.0000 & 0.0000 & -1.0000 & -0.1433 \\ 0.8660 & 0.5000 & 0 & -0.5951 \\ 0 & 0 & 0 & 1.0000 \end{bmatrix}$$

The inverse kinematic function receives the transfer matrix T to obtain the optimized solution, as shown in Fig. 3.3.

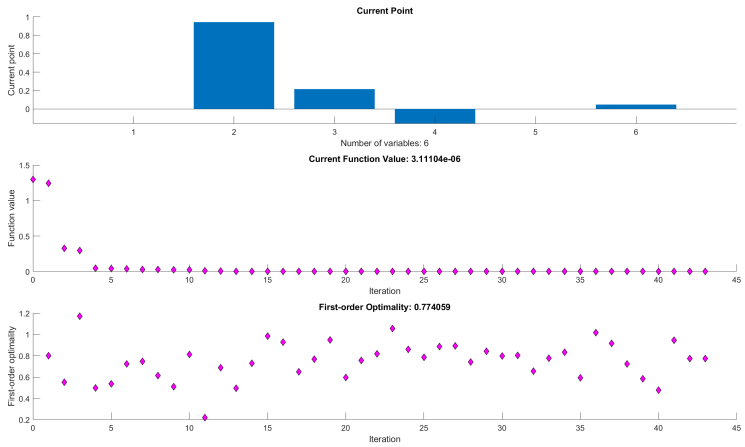


Figure 3.3: Inverse Kinematics Solving Process

The objective function value reaches a small range at step 5 during the iteration process. Finally, the function returns the value of θ after the 45th iteration step.

$$\theta_{solved} = [0.0000,\ 0.9436,\ 0.2162,\ -0.1604,\ -0.0000,\ 0.0478]$$

The value of θ_{solved} is different from inital θ. It is because multiple solutions. When using optimization methods to solve the IK problem for robotic arms, encountering multiple solutions is a common issue. This phenomenon arises primarily because the IK problem is inherently non-unique for many robotic configurations, particularly for robots with six or more degrees of freedom. Each configuration of joint angles that achieves the same end-effector position and orientation is considered a valid solution, leading to the possibility of multiple solutions.

Chapter 4

Jacobian

When controlling the motions of a manipulator, the end-effector is usually required to reach certain points in Cartesian space. Therefore, it is necessary to convert the angular velocities of joint angles between each link into linear velocities at the end effector. In the case of UR5e, there are a total of three functions of linear displacements in terms of 6 joint angles.

$$\begin{aligned} x &= f_1(\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6) \\ y &= f_2(\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6) \\ z &= f_3(\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6) \end{aligned} \tag{2.3}$$

The Jacobian is a matrix of partial derivatives that calculates Cartesian velocities of the end-effector based on joint velocities.

$$J = \begin{bmatrix} \frac{\partial x}{\partial \theta_1} & \frac{\partial x}{\partial \theta_2} & \cdots & \frac{\partial x}{\partial \theta_6} \\ \frac{\partial y}{\partial \theta_1} & \frac{\partial y}{\partial \theta_2} & \ddots & \frac{\partial y}{\partial \theta_6} \\ \frac{\partial z}{\partial \theta_1} & \frac{\partial z}{\partial \theta_2} & \cdots & \frac{\partial x}{\partial \theta_6} \end{bmatrix} \tag{2.4}$$

In MATLAB, the Jacobian matrix can be easily calculated after obtaining the transformation matrix. The position vector from the last column is extracted and the partial derivatives of it with respect to each joint variable are computed.

Besides the measurements of displacements and velocities, force control is also important, especially in the case of body piercing. If the force implemented is not enough, the piercings will not be effective. In contrast, a large piercing force can result in severe damage to human body. Therefore, the torque exerted by each joint need to be related to the endpoint force by applying the Jacobian matrix.

$$\tau = J^T F \tag{2.5}$$

The pseudo-code in Appendix (Find Jacobian Matrix) shows how to obtain Jacobian matrix. First extracting position vector from the last column of transformation matrix. Next, the partial derivatives of the position vector with respect to each joint variable are calculated based on the diff() function in MATLAB.

4.1 Dynamics of the End-Effector

The previous part of the report has already demonstrated how to convert from joint velocities to Cartesian velocities. However, sometimes it is necessary to change the reference frame from the base to other parts of the manipulator. In this case, the velocity vector P of an object measured in frame B with respect to frame A can be calculated by

$$^{A}V_P = {}^{A}V_{B\_ORIGIN} + {}^{A}_{B}R\,{}^{B}V_P + {}^{A}\Omega_B \times {}^{A}_{B}R\,{}^{B}P \tag{2.6}$$

where R is the rotational matrix that moves frame A to frame B. The omega term is the rotational velocity in frame B with respect to A.

Next, the linear acceleration can be derived by differentiating the equation of velocity.

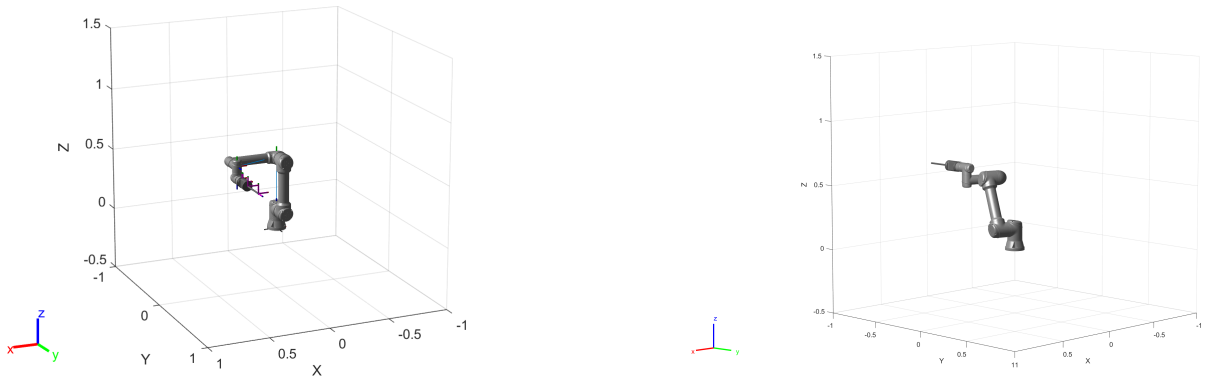
$$^{A}\dot{V}_P = {}^{A}\dot{V}_{B\_ORIGIN} + {}^{A}\Omega_B \times ({}^{A}\Omega_B \times {}^{A}_{B}R\,{}^{B}P) + {}^{A}\dot{\Omega}_B \times {}^{A}_{B}R\,{}^{B}P \tag{2.7}$$

Chapter 5

Trajectory planning

A trajectory describes the motion of a robotic arm in a multidimensional space, including the position, velocity, and acceleration of each degree of freedom over time. The robot's trajectory may be determined by the task, such as tracking a moving object with an end-effector, or simply moving from one position to another within a specified time frame.

It is generally expected that the operating arm moves smoothly. To achieve this, a smooth function that is continuous and has first-order continuous derivatives must be defined. Occasionally, it may also be desirable for the second-order derivative to be continuous. Constraints on the spatial and temporal properties of the path need to be added between intermediate points to ensure smooth motion. Rapid and clumsy motion increases wear on the mechanism and provokes resonance in the operating arm.



(a) Init Position of Ur5e

(b) One Working Positions of ur5e

Figure 5.1: MATLAB Simulation of UR5e

5.1 Methodology for Trajectory Planning

5.1.1 Polynomial Trajectory Planning

Polynomial trajectory planning is a commonly used technique in robotics and automated control systems. It utilises polynomial functions to describe the position, velocity, and acceleration of a robot or robot joint over a time series, resulting in smooth and continuous paths. This method is ideal for ensuring the continuity and controllability of robot motion.

In polynomial trajectory planning, the trajectory is typically represented using a polynomial function of time t. For example, a common form is the cubic polynomial:

$$p(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3 \tag{5.1}$$

Here, $p(t)$ represents the position at time t. The coefficients a_0, a_1, a_2, and a_3 are determined based on boundary conditions, such as the initial and final positions and the initial and final velocities.

For situations requiring higher smoothness and control over additional boundary conditions like initial and final accelerations, a quintic polynomial can be used:

$$p(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3 + a_4 t^4 + a_5 t^5 \tag{5.2}$$

These polynomial functions ensure the trajectory is smooth and continuous, with continuous derivatives up to the required degree, which is crucial for the mechanical systems to operate smoothly.

The pseudo-code of Polynomial Trajectory Planning can be seen in Appendix(Polynomial Trajectory Planning).

5.1.2 Inverse Jacobian Method

Polynomial trajectory planning may not be suitable when a defined trajectory for robotic arm is required. This is because the method can only determine the initial and final states of the robotic arm movement and cannot control the process of the robotic arm movement. In such cases, trajectory planning using inverse Jacobian method becomes a vital alternative.

Inverse Jacobian based trajectory generation involves defining a trajectory, discretizing

it into points, and calculating the Jacobian matrix at each point to relate joint velocities to end effector velocities. The inverse of the Jacobian is then used to determine joint velocities which is expressed as:

$$\dot{\theta} = J^{-1}\dot{x} \tag{5.3}$$

where$\dot{\theta}$represents the joint velocities, and $\dot{x}$ represents the desired velocity of the end effector in task space.

With the joint velocities $\dot{\theta}$ known, the joint displacements over time can be calculated by integrating these velocities. This can be done numerically if the path and timing requirements are discretized. The formula used is:

$$\theta(t) = \theta(t_0) + \int_{t_0}^{t} \dot{\theta}\, dt \tag{5.4}$$

This method offers precise control over the robot's movement, aligning the end effector's motion directly with the desired trajectory.

The pseudo-code of Inverse Jacobian Trajectory Planning can be seen in Appendix(Inverse Jacobian Trajectory Planning).

5.2 Trajectory Description

As the task for this assignment involves identifying 8 target points on the plane of a human body and writing a program to move the distal end of the manipulator to insert a needle onto each point, we assume the problem as a process of using a robotic arm to perform traditional Chinese acupuncture physiotherapy on the back of a human body.

Chinese acupuncture physiotherapy is a traditional Chinese medicine technique that involves the insertion of fine needles into specific points on the body. These points, known as acupoints, are selected based on the flow of Qi (vital energy) through meridians. The practice aims to balance the body's energy flow, alleviate pain, and support overall health and well-being. The acupuncture points on the back of the human body are shown in Fig.5.2

Sanjiaoshu (BL 22), Qihaishu (BL 24), Shangliao (BL 31), Guanyuanshu (BL 26), C.liao (BL 32), Zhongliao (BL 33), Xiaochangshu (BL 27), and Pangguangshu (BL 28) have been selected as the target points for robotic arm acupuncture. These points are

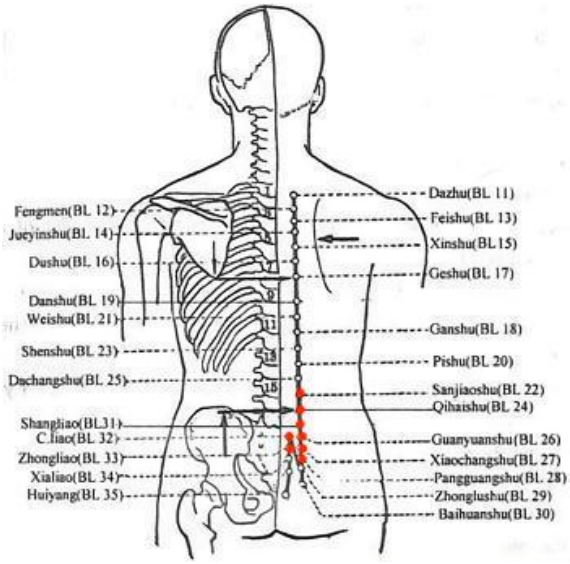


Figure 5.2: Acupuncture Points On the Back

highlighted in red in Fig. 5.2.

For simplicity, we assume that the human body's back is a flat plane. The eight target points, as shown in Figure 1, consist of two points on the left and six points on the right, distributed vertically. The coordinates for each point are in Table 5.1.

Table 5.1: Positions for Specific Acupoints

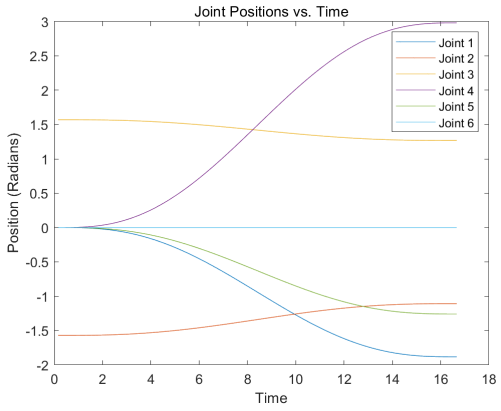
Acupoint	**X_Position**	**Y_Position**	**Z_Position**	**roll**	**pitch**	**yaw**
Sanjiaoshu (BL 22)	0.05	0.6	0.58	$-\frac{\pi}{2}$	0	0
Qihaishu (BL 24)	0.05	0.6	0.55	$-\frac{\pi}{2}$	0	0
Shangliao (BL 31)	0.05	0.6	0.52	$-\frac{\pi}{2}$	0	0
Guanyuanshu (BL 26)	0.05	0.6	0.4	$-\frac{\pi}{2}$	0	0
C.liao (BL 32)	0.08	0.6	0.4	$-\frac{\pi}{2}$	0	0
Zhongliao (BL 33)	0.05	0.6	0.35	$-\frac{\pi}{2}$	0	0
Xiaochangshu (BL 27)	0.08	0.6	0.35	$-\frac{\pi}{2}$	0	0
Pangguangshu (BL 28)	0.05	0.6	0.3	$-\frac{\pi}{2}$	0	0

5.3 Trajectory Planning for Robot Arm

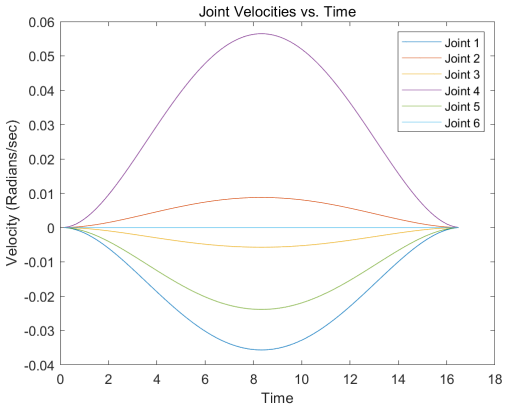
The planning of trajectories for UR5e can be divided into two stages. The first stage involves moving the end of the robot arm to the corresponding position of the acupoint. The second stage involves controlling the needle at the end of the robot arm to insert into the corresponding acupoint in a uniform straight line and return with the same path.

It is important to note that the requirements for these two phases are different. The initial phase only requires the robotic arm to reach the desired end position from the starting

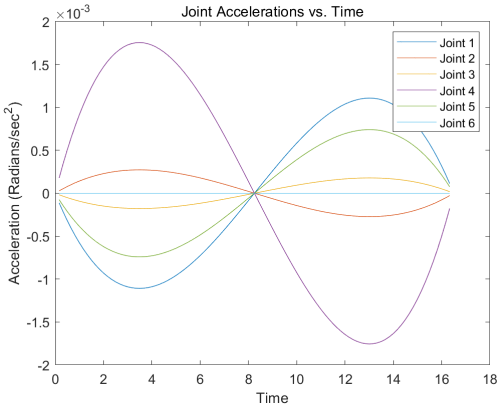
point, without any explicit requirement for the arm's trajectory during movement. In the second stage, the robotic arm must move strictly along a straight line trajectory, maintaining a constant speed throughout. Polynomial Trajectory Planning can be employed for the first stage, depending on the specific requirements of the robotic arm's movement. A fifth degree polynomial is used to ensure smooth motion of the arm at all times. In the second stage, the Inverse Jacobian Method is employed to guarantee absolute control of the end-of-arm operation.



(a) Joint Positions vs. Time



(b) Joint Velocities vs. Time



(c) Joint Accelerations vs. Time

Figure 5.3: Joint Status vs. Time

Fig.5.3 shows that the velocity and acceleration at all six joints of the robotic arm during the initial phase follow smooth curves with no abrupt changes. This indicates that the 5th polynomial interpolation achieves a smooth motion of the arm, reducing mechanical stress on the joints and improving accuracy during operations. As a result, the robot is more reliable and efficient in precision tasks.

Fig. 5.4 displays the position and velocity change curves at the end of the robotic arm during the second phase. It is evident that only the y-axis of the end of the robot arm

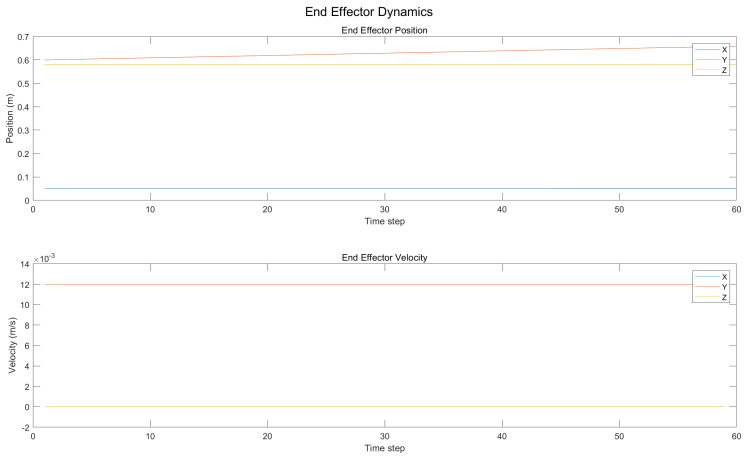


Figure 5.4: End Effector Dynamics in Second Stage

increases linearly with time in the second phase, while the other axes remain constant. Additionally, the end velocity is 0.012m/s only in the y-axis, while the other axes remain at 0. This indicates that the end of the robot arm has successfully achieved uniform linear motion.

This comprehensive trajectory planning for the UR5e robotic arm ensures precise positioning and controlled movements, critical for applications like acupuncture where accuracy and consistency are paramount. The use of polynomial interpolation and the Inverse Jacobian Method effectively balances smooth transitions and strict linear motion, optimizing the robot's performance for specialized tasks.

Chapter 6

Conclusion

In this project, a thorough study has been done for Universal Robot UR5e. A classic D-H representation is modeled based on the parameters of the robot based on the technical specification. Next, the forward kinetics of the manipulator is derived by calculating the transformation matrix based on the previous D-H parameters. The inverse kinetics problem of UR5e is also solved to convert Cartesian vectors back to joint angles. Subsequently, a 3D plane that resembles human back surface is defined to help determine the 8 target points of piercing. A program is then written to move the end-effector of the robot to each location.

This projects also have some limitations. For example, the static forces and joint torques are not calculated based on the end point force of the manipulator. This result can determine the effectiveness of piercing. In addition, the trajectories of the end-effector need to be adjusted for every operation since each human back and its corresponding target points are different. Moreover, all the tasks are performed based on simulations in MATLAB. Physical experiments that involve a real UR5e model are required to verify whether the task of body piercing can be achieved by this manipulator.

The UR5e is a relatively small-scale robot that can perform various tasks with high accuracy. Compared to previous generations of robotic arms, UR5e has a larger payload and working range. Looking ahead, the technological advancements of manipulators allow them to be involved in more industries in the future.

Appendix

Algorithm 1: Robot Model Setup Using D-H Parameters

[1] **Define DH parameters for each link:** $L1 \leftarrow Link([0, 0.1625, 0, \pi/2, 0])$ DH parameters for Link 1 $L2 \leftarrow Link([\pi/2, 0, -0.425, 0, 0])$ DH parameters for Link 2 $L3 \leftarrow Link([0, 0, -0.3922, 0, 0])$ DH parameters for Link 3 $L4 \leftarrow Link([0, 0.1333, 0, \pi/2, 0])$ DH parameters for Link 4 $L5 \leftarrow Link([\pi, 0.0997, 0, -\pi/2, 0])$ DH parameters for Link 5 $L6 \leftarrow Link([0, 0.0996, 0, 0, 0])$ DH parameters for Link 6 **Create a serial link robot model:** $robot \leftarrow SerialLink([L1, L2, L3, L4, L5, L6], 'name', 'UR5e')$ Assemble the robot with a specified name

Algorithm 2: Forward Kinematics Solver

Input: Joint angles vector $\theta = [\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6]$

Output: Transformation matrix T

% Initialize transformation matrix;

$T = I_4$;

for $i = 1$ **to** 6 **do**

  $T = T \cdot A\_mtx(DH[i,1], DH[i,2], DH[i,3], DH[i,4])$;

end

% Auxiliary function to compute individual transformation matrices;

Function $A\_mtx(\alpha, a, d, \theta)$**:**

  $$A_{\text{int\_i0}} = \begin{bmatrix} 1 & 0 & 0 & a \\ 0 & \cos(\alpha) & -\sin(\alpha) & 0 \\ 0 & \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix};$$

  $$A_{\text{i1\_int}} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 & 0 \\ \sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & d \\ 0 & 0 & 0 & 1 \end{bmatrix};$$

  $A_{\text{i0\_i1}} = A_{\text{i1\_int}} \cdot A_{\text{int\_i0}}$;

  return A_{i0_i1};

Algorithm 3: Inverse Kinematics Solver

Input: Position vector of the end effector $Position$

Output: Solution of joint angles θ_{sol}

Define symbolic variables $\theta = [\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6]$;

Obtain the symbolic transformation matrix T_{symb} from the forward kinematics function $FK\_func(\theta)$;

Convert T_{symb} into a numerical function T_func with respect to θ;

Define the error function $error\_function$ as the norm of the difference between $T_{current}$ computed by $T\_func(\theta)$ and the target position $Position$;

Initialize the guess for θ as $\theta_0 = [0, 0, 0, 0, 0, 0]$;

Set the lower and upper bounds for θ as $lb = [-\pi, -\pi/2, -\pi, -\pi, -\pi/2, -\pi]$ and $ub = [\pi, \pi/2, \pi, \pi, \pi/2, \pi]$;

Use the optimizer $fmincon$ with options set for 'sqp' algorithm and iteration display, to minimize $error\_function$ within bounds;

if *optimization is successful* **then**

    Output the solution θ_{sol};

end

else

    Display "Solution failed";

end

Algorithm 4: Find Jacobian Matrix

Input: None directly, uses symbolic joint angles $\theta = [\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6]$

Output: Symbolic Jacobian matrix jL and numeric Jacobian matrix J_val

% Define symbolic variables for joint angles;

Define symbolic joint angles: $\theta_{\text{symb}} = [\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6]$;

% Compute transformation matrix using forward kinematics;

$T = FK\_func(\theta_{\text{symb}})$;

% Extract the position vector from the transformation matrix;

Position vector $p = T[1:3, 4]$;

% Determine the size of position vector and number of joints;

Number of elements in position vector: $numVec = \text{length}(p)$;

Number of joints: $numJoint = \text{length}(\theta_{\text{symb}})$;

% Initialize the Jacobian matrix as a zero matrix;

Initialize Jacobian matrix $jL = \text{sym(zeros)}(numVec, numJoint)$;

for $ind1 = 1$ **to** $numVec$ **do**

  for $ind2 = 1$ **to** $numJoint$ **do**

    Compute partial derivatives:

    $jL[ind1, ind2] = \text{diff}(p[ind1], \theta_{\text{symb}}[ind2])$;

  end

end

% Function to compute numerical Jacobian matrix;

Function $find\_jacobian\_num(jL,\ joint\_ang)$**:**

  Define joint angle values from input $joint\_ang$;

  $J\_val = \text{subs}(jL, [\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6], joint\_ang)$;

  Convert to numeric values: $J\_val = \text{double}(J\_val)$;

  return J_val;

Algorithm 5: Polynomial Trajectory Planning

Input: List of positions pos_list, index j

Output: Matrix of joint angles Θ over time steps

% Time parameters;

Assume total time for motion: $T = 1$ second;

Generate time vector: $t = linspace(0, T, step)$;

% Assume initial and final velocities and accelerations are zero;

Initialize velocities and accelerations:

$\theta\_dot\_0 = \theta\_ddot\_0 = \theta\_dot\_f = \theta\_ddot\_f = [0, 0, 0, 0, 0, 0]$;

% Calculate coefficients for a quintic polynomial;

$a0 = homePosition$;

$a1 = \theta\_dot\_0$;

$a2 = \theta\_ddot\_0/2$;

Calculate $a3, a4, a5$ based on boundary conditions;

% Compute joint angles at each time step using the quintic polynomial;

Initialize matrix for joint angles: $\Theta = zeros(step, 6)$;

for $i = 1$ **to** $step$ **do**

    Compute time instance $t_i = t[i]$;

    Compute joint angles at t_i:

    $\Theta[i, :] = a0 + a1 \cdot t_i + a2 \cdot t_i^2 + a3 \cdot t_i^3 + a4 \cdot t_i^4 + a5 \cdot t_i^5$;

end

Algorithm 6: Inverse Jacobian Trajectory Planning

Input: Final position $Final\_pos$, joint angles matrix Θ from the last time step, total insertion time $Total\_inser\_time$, total insertion length $Total\_insert\_len$, Jacobian matrix function Jac_mtx

Output: Updated matrix of joint angles $Inser\_Theta$

Set total steps for insertion: $Total\_insert\_step = 60$;

Initialize joint angles matrix for insertion: $Inser\_Theta = zeros(Total\_insert\_step, 6)$;

Set initial joint angles for insertion: $Inser\_Theta[1, :] = \Theta[step, :]$;

Compute time step for insertion: $dt = Total\_inser\_time / Total\_insert\_step$;

Set starting point for insertion: $Insert\_Start\_point = Final\_pos$;

for $i = 1$ **to** $Total\_insert\_step - 1$ **do**

  Compute next endpoint for insertion: $Insert\_End\_point = Insert\_Start\_point + [0, Total\_insert\_len/Total\_insert\_step, 0, 0, 0, 0]$;

  Compute next set of joint angles: $Inser\_Theta[i + 1, :] = Insert(Insert\_Start\_point, Insert\_End\_point, Inser\_Theta[i, :], dt, Jac\_mtx)$;

  Update starting point for next step: $Insert\_Start\_point = Insert\_End\_point$;

end

Function $Insert(Start\_point, End\_point, init\_q, dt, jac\_mtx)$;

  Compute displacement vector: $displacement = (End\_point[1:3] - Start\_point[1:3])/dt$;

  Calculate the numerical value of the Jacobian matrix: $Jac\_val = find\_jacobian\_num(jac\_mtx, init\_q)$;

  Compute end velocity: $end\_vel = displacement'$;

  Calculate differential joint angles using pseudo-inverse: $dq = pinv(Jac\_val) \cdot end\_vel$;

  Update joint angles: $q = init\_q + dq' \cdot dt$;

  return q;