UNIVERSITÀ DEGLI STUDI DI GENOVA

DIBRIS

DEPARTMENT OF COMPUTER SCIENCE AND TECHNOLOGY,
BIOENGINEERING, ROBOTICS AND SYSTEM ENGINEERING

MODELLING AND CONTROL OF MANIPULATORS

Third Assignment

Jacobian Matrices and Inverse Kinematics

Author:

Mamoru Ota

Student ID:

s7784976

Professors:

Enrico Simetti
Giorgio Cannata

Tutors:

Andrea Tiranti
Luca Tarasi
George Kurshakov

December 21, 2024

Contents

Mathematical expression	Definition	MATLAB expression
$< w >$	World Coordinate Frame	w
${}^{a}_{b}R$	Rotation matrix of frame $< b >$ with respect to frame $< a >$	aRb
${}^{a}_{b}T$	Transformation matrix of frame $< b >$ with respect to frame $< a >$	aTb
${}^{a}O_{b}$	Vector defining frame $< b >$ wit respect to frame $< a >$	aOb

Table 1: Nomenclature Table

1 Assignment description

The third assignment of Modelling and Control of Manipulators focuses on Inverse Kinematics (IK) control of a robotic manipulator.

The third assignment consists of three exercises. You are asked to:

- Download the .zip file called MCM_assignment3.zip from the Aulaweb page of this course.
- Implement the code to solve the exercises on MATLAB by filling in the predefined files. In particular, you will find two different main files: *"ex1.m"* for the first exercise and *"ex2.m"* for the second exercise.
- Write a report motivating your answers, following the predefined format on this document.
- **Putting code in the report is not an explanation!**

1.1 Exercise 1

Given the geometric model of an industrial manipulator used in the previous assignment, you have to add a tool frame. The tool frame is rigidly attached to the robot end-effector according to the following specifications:

Use the following specifications ${}^{e}\eta_{t/e} = [0, 0, \pi/10]$, ${}^{e}O_t = [0.2, 0, 0]^\top(m)$ where ${}^{e}\eta_{t/e}$ represents the YPR values from end effector frame to tool frame.

To complete this task you should modify the class *geometricModel* by adding a new method called *getToolTransformWrtBase*

1.2 Exercise 2

Implement an inverse kinematic control loop to control the tool of the manipulator. You should be able to complete this exercise by using the MATLAB classes implemented for the previous assignment (*geometricModel*,*kinematicModel*), and also you need to implement a new class *cartesianControl* (see the template attached). The procedure can be split into the following phases

Q2.1 Compute the cartesian error between the robot end-effector frame ${}^{b}_{t}T$ and the goal frame ${}^{b}_{g}T$.

The goal frame must be defined knowing that:

- The goal position with respect to the base frame is ${}^{b}O_g = [0.15, -0.85, 0.3]^\top(m)$
- The goal frame is rotated of $\theta = \pi/6$ around the y-axis of the base frame (inertial frame).

Q2.2 Compute the desired angular and linear reference velocities of the end-effector with respect to the base: ${}^{b}\nu^{*}_{t/b} = \begin{bmatrix}\kappa_a & 0 \\ 0 & \kappa_l\end{bmatrix} \cdot {}^{b}e$, such that $\kappa_a = 0.8$,$\kappa_l = 0.8$ is the gain.

Q2.3 Compute the desired joint velocities $\dot{\bar{q}}$

Q2.4 Simulate the robot motion by implementing the function: *"KinematicSimulation()"* for integrating the joint velocities in time.

2 Exercise 1

[Comment] For the last exercises include an image of the initial robot image of the final robot configuration
[Comment] For each exercise report the results obtained and provide an explanation of the result obtained (even though it might seem trivial). The matlab code is NOT an explanation of the algorithm.

Abstract

This report about exercise 1 details the integration of a tool frame into the geometric model of an industrial manipulator. The tool frame is rigidly attached to the end-effector of the manipulator and defined using Yaw-Pitch-Roll (YPR) values and a position vector. Transformation matrices were computed to describe the relationship between the manipulator's base and the tool frame.

Introduction

In this exercise, a tool frame is incorporated into the geometric model of a 7-DoF manipulator shown in Figure 1. The specifications for the tool frame are below:

- YPR values: ${}^{e}\eta_{t/e} = [0, 0, \pi/10]$
- Position vector: ${}^{e}O_t = [0.2, 0, 0]^\top (m)$

The transformation matrix ${}^{b}_{t}T$ from the manipulator base to the tool frame is computed. This matrix integrates both the manipulator's joint configuration and the rigid attachment of the tool frame.

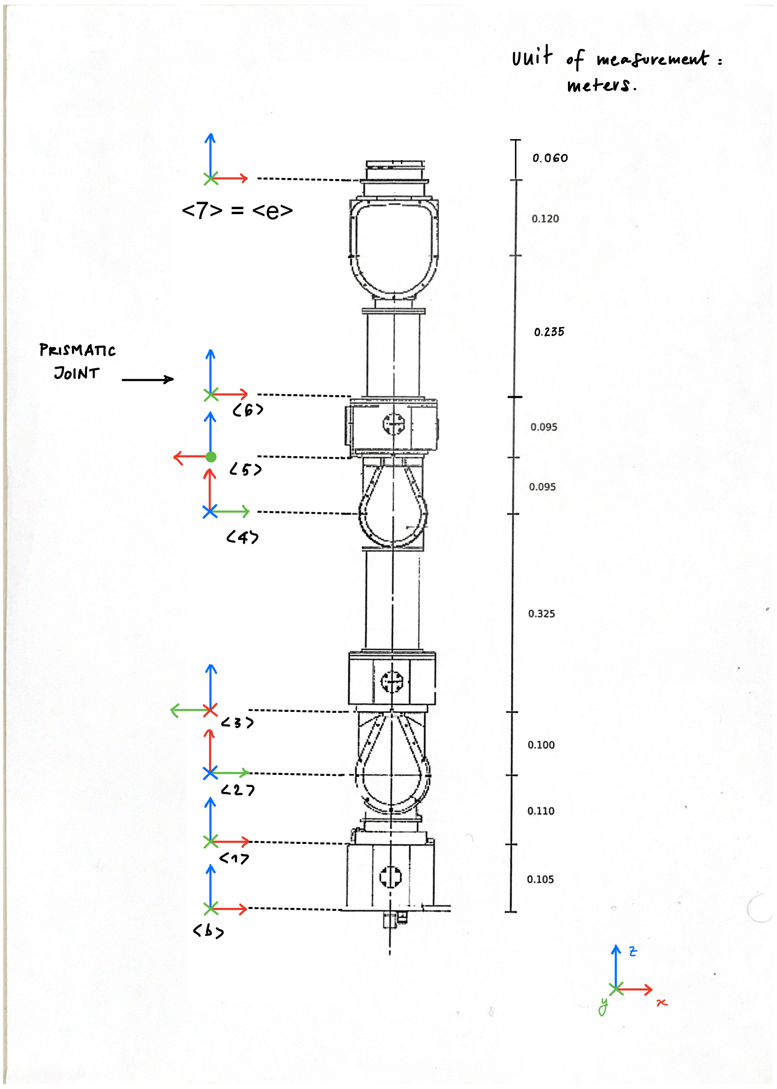

Figure 1: Overview of the manipulator.

Methodology

The tool frame is defined relative to the end-effector using a rotation matrix ${}^{e}_{t}R$ derived from YPR values and a translation ${}^{e}O_t$. The transformation matrix ${}^{e}_{t}T$ is:

$$ {}^{e}_{t}T = \begin{bmatrix} {}^{e}_{t}R & {}^{e}O_t \\ 0^T & 1 \end{bmatrix}. $$

Given YPR angles, the rotation matrix ${}^{e}_{t}R$ can be computed as a product of individual rotations around fixed axes. For example:

$$ {}^{e}_{t}R = R_z(\psi)R_y(\theta)R_x(\phi), $$

The geometric model was extended to include the tool frame transformation. `getToolTransformWrtBase()` is a new method and computes ${}^{b}_{t}T$ using the base-to-end-effector transformation ${}^{b}_{e}T$ and the tool frame definition ${}^{e}_{t}T$:

$$ {}^{b}_{t}T = {}^{b}_{e}T \cdot {}^{e}_{t}T. $$

The following pseudocode outlines the implementation of the geometric model class:

Algorithm 1 Geometric Model Class Implementation

```
1:  Input: ${}^{i}_{j_0}T$ (Initial transformations), $jointType$ (Joint types), ${}^{e}_{t}T$ (Tool frame transformation, optional)
2:  Properties:
3:      ${}^{i}_{j_0}T$: Initial transformations
4:      jointType: Vector of joint types (0: Rotational, 1: Prismatic)
5:      jointNumber: Number of joints
6:      ${}^{i}_{j}T$: Current transformations
7:      q: Joint positions
8:      ${}^{e}_{t}T$: Tool frame transformation (default: identity)
9:  function CONSTRUCTOR(${}^{i}_{j_0}T, jointType, {}^{e}_{t}T$)
10:     if ${}^{e}_{t}T$ not provided then
11:         ${}^{e}_{t}T \leftarrow I_4$                         ▷ Set default to identity matrix
12:     end if
13:     Initialize ${}^{i}_{j_0}T$, jointType, jointNumber, ${}^{i}_{j}T$, q, ${}^{e}_{t}T$
14: end function
15: function UPDATEDIRECTGEOMETRY($q$)
16:     $self.q \leftarrow q$
17:     $self.{}^{i}_{j}T \leftarrow self.{}^{i}_{j_0}T$
18:     for $i \leftarrow 1$ to $jointNumber$ do
19:         if $jointType[i] = 0$ then                          ▷ Rotational joint
20:             $T \leftarrow$ RotationMatrix($q[i]$)
21:         else if $jointType[i] = 1$ then                     ▷ Prismatic joint
22:             $T \leftarrow$ TranslationMatrix($q[i]$)
23:         else
24:             Error: Invalid joint type
25:         end if
26:         Update ${}^{i}_{j}T \leftarrow {}^{i}_{j}T \cdot T$
27:     end for
28: end function
29: function GETTRANSFORMWRTBASE($k$)
30:     ${}^{b}_{k}T \leftarrow I_4$
31:     for $i \leftarrow 1$ to $k$ do
32:         ${}^{b}_{k}T \leftarrow {}^{b}_{k}T \cdot {}^{i}_{j}T$
33:     end for
34:     return ${}^{b}_{k}T$
35: end function
36: function GETTOOLTRANSFORMWRTBASE
37:     ${}^{b}_{e}T \leftarrow$ getTransformWrtBase($jointNumber$)
38:     ${}^{b}_{t}T \leftarrow {}^{b}_{e}T \cdot {}^{e}_{t}T$
39:     return ${}^{b}_{t}T$
40: end function
```

Results

Transformation Matrices

The manipulator's initial configuration ($q_0 = [0, 0, 0, 0, 0, 0, 0]$) yields the following results:

Transformation Matrices ${}^{i}_{j}T$ (Initial Configuration):

$$
{}^{b}_{1}T = \begin{bmatrix} 1.0000 & 0 & 0 & 0 \\ 0 & 1.0000 & 0 & 0 \\ 0 & 0 & 1.0000 & 0.1050 \\ 0 & 0 & 0 & 1.0000 \end{bmatrix}
$$

$$
{}^{1}_{2}T = \begin{bmatrix} 0 & 1.0000 & 0 & 0 \\ 0 & 0 & 1.0000 & 0 \\ 1.0000 & 0 & 0 & 0.1100 \\ 0 & 0 & 0 & 1.0000 \end{bmatrix}
$$

$$
{}^{2}_{3}T = \begin{bmatrix} 0 & 0 & 1.0000 & 0.1000 \\ 0 & -1.0000 & 0 & 0 \\ 1.0000 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1.0000 \end{bmatrix}
$$

$$
{}^{3}_{4}T = \begin{bmatrix} 0 & 0 & 1.0000 & 0 \\ 0 & -1.0000 & 0 & 0 \\ 1.0000 & 0 & 0 & 0.3250 \\ 0 & 0 & 0 & 1.0000 \end{bmatrix}
$$

$$
{}^{4}_{5}T = \begin{bmatrix} 0 & 0 & 1.0000 & 0.0950 \\ -1.0000 & 0 & 0 & 0 \\ 0 & -1.0000 & 0 & 0 \\ 0 & 0 & 0 & 1.0000 \end{bmatrix}
$$

$$
{}^{5}_{6}T = \begin{bmatrix} -1.0000 & 0 & 0 & 0 \\ 0 & -1.0000 & 0 & 0 \\ 0 & 0 & 1.0000 & 0.0950 \\ 0 & 0 & 0 & 1.0000 \end{bmatrix}
$$

$$
{}^{6}_{7}T = \begin{bmatrix} 1.0000 & 0 & 0 & 0 \\ 0 & 1.0000 & 0 & 0 \\ 0 & 0 & 1.0000 & 0.3550 \\ 0 & 0 & 0 & 1.0000 \end{bmatrix}
$$

Tool Frame Transformation ${}^{e}_{t}T$:

$$
{}^{e}_{t}T = \begin{bmatrix} 1.0000 & 0 & 0 & 0.2000 \\ 0 & 0.9511 & -0.3090 & 0 \\ 0 & 0.3090 & 0.9511 & 0 \\ 0 & 0 & 0 & 1.0000 \end{bmatrix}.
$$

Base-to-Tool Transformation ${}^{b}_{t}T$:

$$
{}^{b}_{t}T = \begin{bmatrix} 1.0000 & 0 & 0 & 0.2000 \\ 0 & 0.9511 & -0.3090 & 0 \\ 0 & 0.3090 & 0.9511 & 1.1850 \\ 0 & 0 & 0 & 1.0000 \end{bmatrix}.
$$

Conclusion

The task successfully demonstrated the extension of a manipulator's geometric model to include a rigidly attached tool frame.

3 Exercise 2

Introduction

This report presents the implementation of an inverse kinematic control loop for a manipulator tool. The exercise focuses on controlling the end-effector position and orientation through the following steps:

1. Compute the Cartesian error between the robot end-effector and the goal frame.
2. Calculate the desired angular and linear reference velocities of the end-effector.
3. Compute the desired joint velocities using inverse kinematics.
4. Simulate the motion of the robot using the computed joint velocities.

Problem Setup

The target frame is defined by:

- Position: $bO_g = [-0.14, -0.85, 0.6]^T$ (m)
- Orientation (YPR angles): $b\eta_g = [-3.02, -0.40, -1.33]^T$ (rad)

The goal frame transformation matrix is:

$$
{}^b_gT = \begin{bmatrix} -0.9143 & -0.3465 & 0.2100 & -0.1400 \\ -0.1117 & -0.2826 & -0.9527 & -0.8500 \\ 0.3894 & -0.8945 & 0.2197 & 0.6000 \\ 0 & 0 & 0 & 1.0000 \end{bmatrix}
$$

Methodology

Q2.1 Cartesian Error & Q2.2 Desired Velocity

The Cartesian error is divided into two components:

- **Position error:**
$$e_p = bO_g - bO_t$$
where bO_g is the goal position and bO_t is the current position of the end-effector.
- **Orientation error:**
$$e_o = \text{RotToAngleAxis}({}^b_gR \cdot ({}^b_tR)^T)$$
where ${}^b_gR$ and ${}^b_tR$ are the rotation matrices of the goal and current end-effector frames, respectively.

The orientation error, e_o, is calculated using the function ‘RotToAngleAxis‘, which converts a rotation matrix into an equivalent angle-axis representation.

- **Cartesian Error Calculation** The following algorithm computes the Cartesian errors and desired velocities for inverse kinematic control:

Algorithm 2 Cartesian Control for Inverse Kinematics

```
procedure GETCARTESIANREFERENCE($^b_gT$)
    $^b_tT \leftarrow$ Get tool transform w.r.t base
    Compute Orientation Error:
    $^bR_{error} \leftarrow ^b_g R \cdot (^b_tR)^T$
    $[h, \theta] \leftarrow$ RotToAngleAxis($^bR_{error}$)
    $e_o \leftarrow h \cdot \theta$
    Compute Position Error:
    $e_p \leftarrow bO_g - bO_t$
    Scale Errors:
    $angular\_velocity \leftarrow \kappa_a \cdot e_o$
    $linear\_velocity \leftarrow \kappa_l \cdot e_p$
    Output: $x_{dot} \leftarrow [angular\_velocity; linear\_velocity]$
end procedure
```

The function takes the goal transformation matrix, ^b_gT, as input and outputs the Cartesian reference velocities, x_{dot}, which include both angular and linear velocities. These velocities are scaled by proportional gains κ_a and κ_l.

- **Rotation to Angle-Axis Conversion**

  The 'RotToAngleAxis' function computes the orientation error by converting a rotation matrix into an angle-axis representation. The algorithm is given below:

Algorithm 3 Rotation to Angle-Axis Conversion

```
procedure ROTTOANGLEAXIS($R$)
    Input: $R$ (3x3 rotation matrix)
    Output: Axis $h$, Angle $\theta$
    Check Inputs:
    if size($R$) $\neq (3,3)$ or det($R$) $\neq 1$ or $R^TR \neq I$ then
        Error: Invalid rotation matrix.
    end if
    Compute Angle:
    $\theta \leftarrow \arccos\left(\frac{\text{trace}(R)-1}{2}\right)$
    Compute Axis:
    if $\theta \approx 0$ then
        $h \leftarrow [0; 0; 0]$
    else if $\theta \approx \pi$ then
        Compute axis using eigenvector method.
    else
        $h \leftarrow \frac{1}{2\sin(\theta)}[R_{32} - R_{23}; R_{13} - R_{31}; R_{21} - R_{12}]$
        Normalize $h$
    end if
    Output: $h$, $\theta$
end procedure
```

This algorithm first validates the input rotation matrix to ensure it is orthogonal and has a determinant of 1. It then calculates the rotation angle θ and axis h based on the matrix properties.

- **Outputs**

  The Cartesian errors for position and orientation are displayed as follows:

  - Position Error (e_p) shows the difference between the current and target positions.
  - Orientation Error (e_o) captures the angular difference using the angle-axis representation.

Q2.3 Desired Joint Velocities

The joint velocities were computed using the Jacobian J:

$$\dot{q} = J^{\dagger} \cdot x_{dot}$$

where $J^{\dagger}$ is the pseudo-inverse of J.

- **Jacobian Computation**

  The Jacobian matrix is calculated using the following algorithm using a geometric model:

Algorithm 4 Jacobian Computation

```
procedure UPDATEJACOBIAN
    Input: Geometric model (gm)
    Output: Jacobian matrix $J$
    Initialize:
    $J \leftarrow$ zeros$(6, numJoints)$
    ${}^b_eT \leftarrow$ GetTransformWrtBase$(numJoints)$
    for $i = 1 \rightarrow numJoints$ do
        ${}^b_jT \leftarrow$ GetTransformWrtBase$(i)$
        ${}^b_jR \leftarrow {}^b_j T(1:3, 1:3)$
        ${}^b_jP \leftarrow {}^b_j T(1:3, 4)$
        ${}^b_eP \leftarrow {}^b_e T(1:3, 4)$
        $zAxis \leftarrow {}^b_j R(:, 3)$
        if Joint is rotational (0) then
            $J(1:3, i) \leftarrow zAxis$
            $J(4:6, i) \leftarrow zAxis \times ({}^bP_e - {}^bP_j)$
        else if Joint is prismatic (1) then
            $J(1:3, i) \leftarrow [0; 0; 0]$
            $J(4:6, i) \leftarrow zAxis$
        else
            Error: Invalid joint type
        end if
    end for
    Return: $J$
end procedure
```

In this algorithm:

- For each joint, get the translation and rotation matrices.
- Calculate the linear and angular velocity contributions relative to each joint's position and the end effector position.
- Check if the joint type is revolute or prismatic and perform calculations accordingly.

Q2.4 Kinematic Simulation

The following algorithm updates the joint positions based on velocities, sample time, and joint limits:

Algorithm 5 Kinematic Simulation

```
procedure KINEMATICSIMULATION$(q, q\_dot, ts, q\_min, q\_max)$
    $q\_new \leftarrow q + q\_dot \cdot ts$                          ▷ Update joint configuration
    $q\_new \leftarrow \max(q\_min, \min(q\_new, q\_max))$           ▷ Apply joint limits
    Output: Updated joint configuration $q\_new$
end procedure
```

Results

Manipulator Motion

Figure 2 shows the motion of the manipulator during the simulation. The manipulator starts from the initial configuration and moves toward the target goal configuration based on the inverse kinematic control algorithm implemented. The trajectory is visualized by plotting intermediate positions of the joints and end-effector at different time steps.

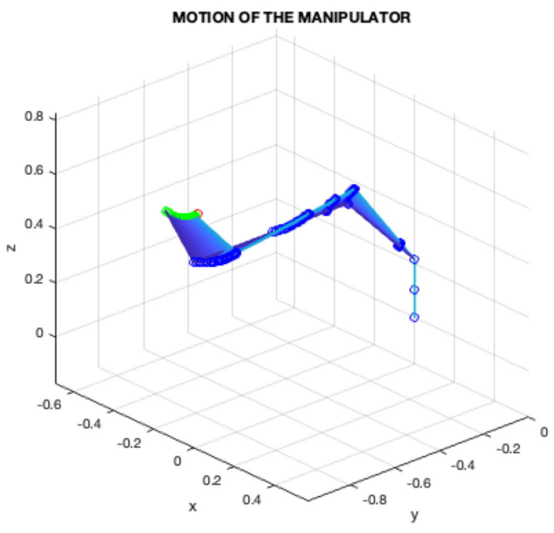

Figure 2: Manipulator motion during simulation.

Configurations

Figure 3 illustrates the initial and final configurations of the manipulator. The initial configuration, represented by red lines, shows the starting pose of the robot, while the final configuration, shown in blue lines, represents the achieved pose after the control loop converges to the goal position and orientation. The comparison highlights the manipulator's ability to follow the desired trajectory and reach the specified goal.

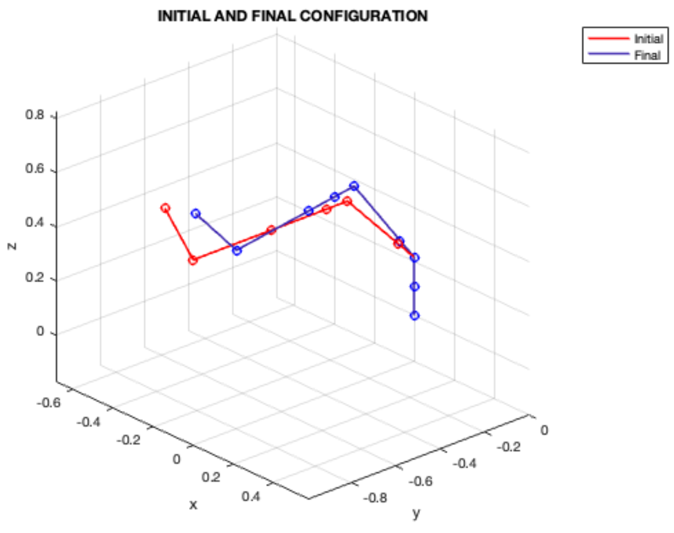

Figure 3: Initial and final configurations.

Velocities

Figure 4 shows the velocity profiles of the end-effector in both angular and linear directions during the simulation. The six components of velocity ($\omega_x, \omega_y, \omega_z$ for angular velocities and v_x, v_y, v_z for linear velocities) are plotted. These profiles demonstrate the dynamic behavior of the manipulator as it adjusts its velocities to reduce the Cartesian error over time.

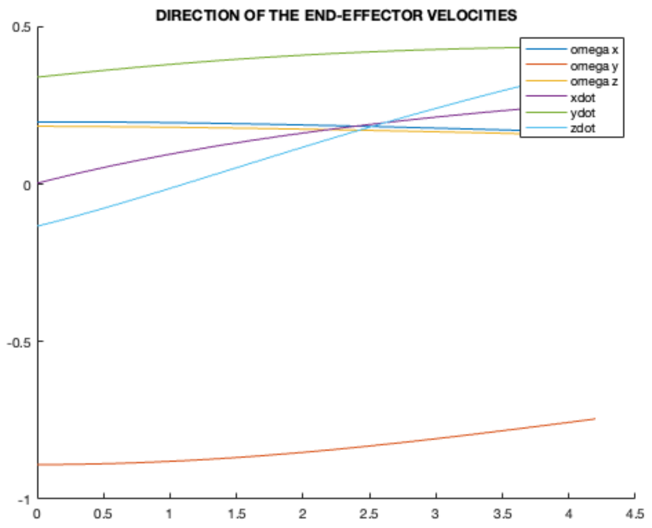

Figure 4: End-effector velocities.

Discussion

The results indicate that the implemented inverse kinematic control successfully guided the manipulator to the desired goal configuration. The following points summarize key observations:

- **Trajectory Execution:** The manipulator smoothly transitions from the initial configuration to the goal configuration. The visualization in Figure 2 confirms that the control algorithm effectively computes the required joint velocities and updates joint positions iteratively.
- **Convergence:** The final configuration in Figure 3 closely matches the target pose, verifying the accuracy of the inverse kinematics computation. Minimal deviation highlights the stability of the algorithm.
- **Velocity Profiles:** The velocity profiles shown in Figure 4 exhibit smooth variations in angular and linear velocities. This demonstrates that the control gains ($\kappa_a = 0.8$, $\kappa_l = 0.8$) effectively regulate the end-effector's motion, ensuring convergence without oscillations or instability.
- **Error Handling:** The Cartesian errors in position and orientation are progressively reduced to near-zero values as the simulation proceeds. The algorithm's robustness in handling small numerical errors and avoiding overshooting further validates its reliability.
- **Limitations:** While the implemented algorithm performs well in the given example, its performance under varying constraints (e.g., joint limits, singularities) should be analyzed in future studies. Additionally, the effect of different control gains on the trajectory and stability could be investigated to optimize performance further.

Overall, the simulation results demonstrate the effectiveness of the inverse kinematic control algorithm in guiding the manipulator to the desired pose. Further improvements and testing may focus on optimizing gains and enhancing computational efficiency.

Conclusion

This report demonstrated the implementation of inverse kinematics for manipulator control. The manipulator successfully reached the goal configuration while maintaining smooth motion.

4 Appendix

[Comment] Add here additional material (if needed)

4.1 Appendix A

4.2 Appendix B