BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI
HYDERABAD CAMPUS

SECOND SEMESTER 2022-2023
ME F366 LABORATORY PROJECT

FINAL PROJECT REPORT

Title: Humanoid Robot Control
Date: 05/05/2023

Project Overview

Objectives:
In this project I'm required to:

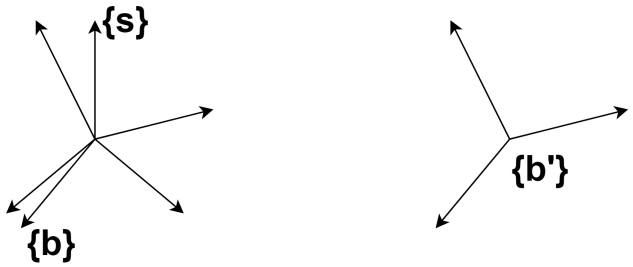
1) Model the tree-structure robot in MATLAB code and find out the jacobian of the manipulator using screw-theory approach or using the geometric approach.

2) Use the evaluated jacobian in the chosen numerical inverse kinematic solver to obtain the desired joint displacements that would achieve the desired manipulator configuration.

3) Using the developed IK, trace out the trajectory of one or more end-effectors of the robot.

Theoretical Background

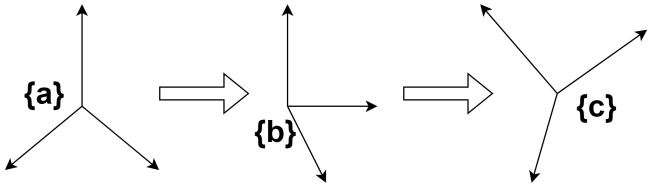
Review of robotic manipulator basics:
Refer to the following figure which is used to define the basic definitions below:



Rotation matrix:
It is the relationship between two coordinate frames where one is purely rotated about an axis to get the other. It is a 3x3 matrix as follows:

$$R_{sb} = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix}$$

Subscript-cancellation rule:-
Consider the below figure with the rotation matrices describing one transformation from one frame to another as R_{ab} and R_{bc}:



We can then express the rotation matrix describing frame {c} relative to {a} (R_{ac}) as follows:

$$R_{ac} = R_{ab}R_{bc}$$

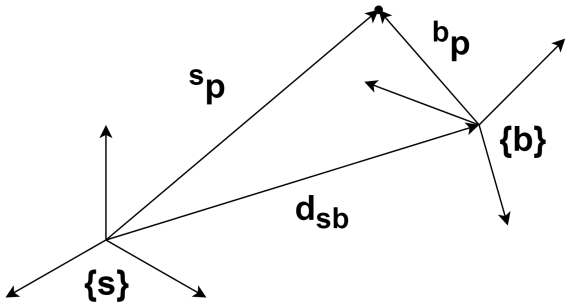
We can remember this expression as *canceling* the adjacent subscripts of the two rotation matrices.

Translation matrix:
It is the relationship between two coordinate frames in which one frame is both translated as well as rotated w.r.t another frame. It is a 4x4 matrix as follows:

$$T_{sb'} = \begin{bmatrix} R_{sb'} & p_{sb'} \\ 0_{1\times 3} & 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & p_1 \\ r_{21} & r_{22} & r_{23} & p_2 \\ r_{31} & r_{32} & r_{33} & p_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

We can use the same *subscript-cancellation rule* as above to convert a transformation matrix from one frame to another.

Deriving expression for homogeneous transformation matrix:-



The transformation from space frame to body frame can be described as the relationship between space and body coordinates in representing a point:

$$^s\vec{p} = {}^b\vec{p} + \vec{d_{sb}}$$

Assuming pure translation to the origin of body frame (to intermediate frame, {i}), followed pure rotation we have:

$$^s p = {}^i\ p + d_{si}$$
$$^i p = R_{ib}\,{}^b p$$

Since $R_{ib} = R_{sb}$ and $d_{si} = d_{sb}$, we have:

$$\Rightarrow {}^s\ p = R_{sb}\,{}^b p + d_{sb}$$

Hence Proved.

Jacobian matrix:

It is the relationship between the joint velocities of the manipulator and the end-effectors' linear and angular velocities. It is a matrix represented by the symbol *J* as follows:

$$\nu(t) = J(\theta)\dot{\theta(t)}$$

$$Where\ \nu(t)\ is\ the\ end-effector\ twist\ of\ the\ manipulator$$

The component of Jacobian for:

1) Revolute joint:

$$J_i(\theta) = \begin{bmatrix} J_{vi} \\ J_{\omega i} \end{bmatrix} = \begin{bmatrix} P_{i-1} \times^{i-1} P_n \\ P_{i-1} \end{bmatrix}$$

2) Prismatic joint:

$$J_i(\theta) = \begin{bmatrix} J_{vi} \\ J_{\omega i} \end{bmatrix} = \begin{bmatrix} P_{i-1} \\ 0 \end{bmatrix}$$

where P_{i-1} is the vector pointing along the axis of joint J_{i-1} w.r.t space(base) frame {s}.

Screw Theory Formulation for Rigid Bodies:

Angular Velocity:

The angular velocity of any rigid body is given by the combination of *direction of axis of rotation + speed of rotation*. Thus we can simply multiply the unit axis vector with the angular speed to get a mathematical representation of the angular velocity for a rigid body.

$$\Rightarrow \omega_s = \hat{\omega_s}\dot{\theta}$$

Consider that the body frame {b} is rotating about an axis ω_s w.r.t the space frame {s} as shown in the below figure:

Now due to this rotation, the x-axis of the body frame {b} traces a circle and its linear velocity is given as:

$$\dot{\hat{x}_b} = \omega_s \times \hat{x}_b$$

Similarly,

$$\dot{\hat{y}}_b = \omega_s \times \hat{y}_b$$

$$\dot{\hat{z}}_b = \omega_s \times \hat{z}_b$$

$$\dot{R}_{sb} = \begin{bmatrix} \dot{\hat{x}}_b & \dot{\hat{y}}_b & \dot{\hat{z}}_b \end{bmatrix} = \begin{bmatrix} \omega_s \end{bmatrix} R_{sb}$$

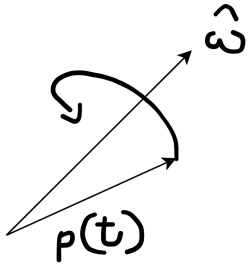
To express the angular velocity of the body frame in body frame coordinates instead of the space frame, we have:

$$\omega_b = R_{bs}\omega_s = R_{sb}^{-1}\omega_s = R_{sb}^{T}\omega_s$$

Important relations summarized:-

$$\omega_s = R\omega_b \qquad \begin{bmatrix} \omega_s \end{bmatrix} = \dot{R}R^{-1} \qquad \begin{bmatrix} \omega_b \end{bmatrix} = R^{-1}\dot{R}$$

Exponential Coordinates of Rotation:
An alternative representation for the rotation of a frame other than the rotation matrix is the exponential coordinate representation. We simply multiply the unit axis vector with the magnitude of rotation in this method.
To get the relationship between this representation and the homogeneous transformation matrix, we have:



The linear velocity is related to the angular velocity of the body at any time instant as follows:

$$\dot{p}(t) = \hat{\omega} \times p(t)$$
$$\Rightarrow \dot{p}(t) = \begin{bmatrix} \hat{\omega} \end{bmatrix} p(t)$$

The solution to the above equation simplifies to a matrix exponential as follows:

where *exp([ω]θ)* is the matrix exponential function and *p(t)* is the direction of a vector at a certain instant.

This matrix exponential can be evaluated by using the Cayley Hamilton theorem [3], and the final expression is as follows:

$$Rot(\hat{\omega}, \theta) = I + \sin\theta \left[\hat{\omega}\right] + (1 - \cos\theta) \left[\hat{\omega}\right]^2$$

The inverse of this operation involves converting the 3-parameter representation of rotation into the redundant 9-parameter rotation matrix [2] as follows:

$$\left[\hat{\omega}\right] = \frac{1}{2\sin\theta}(R - R^T)$$

Twists:

The combined rotational and translational motion of any rigid body motion can be interpreted as the instantaneous motion about some screw axis. This screw axis is defined by three parameters: the direction of the axis of screw, the amount of rotation, and the pitch of the screw.

$$Pitch(h) = \frac{linear\ speed}{angular\ speed}$$

We define the screw axis as follows:

$$\mathcal{S} = \begin{bmatrix} \mathcal{S}_\omega \\ \mathcal{S}_v \end{bmatrix} = \begin{bmatrix} angular\ velocity\ when\ \dot{\theta} = 1 \\ linear\ velocity\ of\ the\ origin\ when\ \dot{\theta} = 1 \end{bmatrix}$$

Multiplying this screw axis with the instantaneous rotational velocity of the screw, we have:

$$Twist(\nu) = \mathcal{S}\dot{\theta}$$

The *Twist* refers to the combined rotational and translational velocity of a body relative to a frame. We have two cases for the twist of a body, when the pitch is finite and when it is infinite:

$$h = \inf \Rightarrow \mathcal{S}_\omega = 0, \|\mathcal{S}_v\|, \dot{\theta} = linear\ speed$$

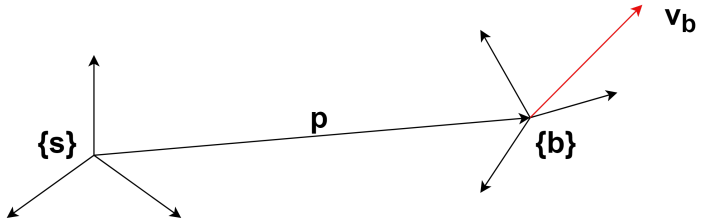
$$\|\mathcal{S}_\omega\| = 1, \dot{\theta} = rotational\ speed$$

To represent a twist represented in one frame relative to another, we need to use a different matrix for transformation as follows:

$$Given,\ T = \begin{bmatrix} R & p \\ 0_{1\times 3} & 1 \end{bmatrix}\ the\ matrix\ adjoint\ is\ [Ad_T] = \begin{bmatrix} R & 0_{3\times 3} \\ [p]\,R & R \end{bmatrix}$$

Proof:-

Consider the following figure where the body frame is both translating as well as rotating, and the body has a velocity vb w.r.t body frame {b}:



We know that -

$$\omega_s = R_{sb}\omega_b\ (1)$$

The total velocity of the body w.r.t the space frame {s} can be given as the sum of linear velocity component of body {b} + the linear velocity component due to the rotation of {b}-frame w.r.t {s}-frame:

$$\vec{v_s} = \vec{v_b} + \vec{\omega} \times \vec{p}$$

Representing the above equation in the space frame coordinates, we have:

$$v_s = R_{sb}v_b + [p_{sb}]\,\omega_s$$
$$\Rightarrow v_s = R_{sb}v_b + [p_{sb}]\,R_{sb}\omega_b\ (2)$$

From (1) & (2), we have -

$$\nu_s = \begin{bmatrix} \omega_s \\ v_s \end{bmatrix} = \begin{bmatrix} R & 0 \\ [p]\,R & R \end{bmatrix} \begin{bmatrix} \omega_b \\ v_b \end{bmatrix} = [Ad_{T_{sb}}]\,\nu_b$$

Hence Proved.

The relationship between the twists and transformation matrices is given by the following expressions:

$$[\nu_b] = T^{-1}\dot{T} = \begin{bmatrix} [\omega_b] & v_b \\ 0 & 0 \end{bmatrix}$$
$$[\nu_s] = \dot{T}T^{-1} = \begin{bmatrix} [\omega_s] & v_s \\ 0 & 0 \end{bmatrix}$$

Proof:-
Consider the multiplication of the following two matrices -

$$T^{-1}\dot{T} = \begin{bmatrix} R^T & -R^T p \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \dot{R} & \dot{p} \\ 0 & 0 \end{bmatrix}$$
$$\Rightarrow \begin{bmatrix} R^T\dot{R} & R^T\dot{p} \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} [\omega_b] & v_b \\ 0 & 0 \end{bmatrix}$$

Hence Proved.

Exponential coordinates of Rigid Body motion:
Just as in the case of rotation where we obtained the exponential coordinates by multiplying the unit rotational axis vector with the speed of rotation, a similar expression gives the exponential coordinates for rigid body motion:

$$Exponential\ Coordinates: \ \mathcal{S}\theta$$

The closed form solution for the rigid body motion has two cases for the matrix exponential function [2]:
Case (i): Motion is purely translational (*pitch* = ∞)

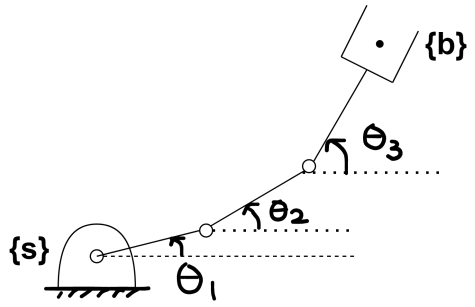
$$\exp([\mathcal{S}]\,\theta) = \begin{bmatrix} I & \mathcal{S}_v\theta \\ 0 & 1 \end{bmatrix}$$

Case (ii): Motion has a rotational component (*pitch* = finite)

$$\exp([\mathcal{S}]\,\theta) = \begin{bmatrix} [\mathcal{S}_\omega] & (I\theta + (1-\cos\theta)\,[\mathcal{S}_\omega] + (\theta - \sin\theta)\,[\mathcal{S}_\omega]^2)\mathcal{S}_v \\ 0 & 1 \end{bmatrix}$$

Background for the Project

Some important equations used:
Refer to the below schematic to be able to relate to all the formulae discussed henceforth:



Forward Kinematics POE (Product of exponentials) formula:
In the space frame {s} [2]:-

$$T(\theta) = T_{sb} = (\prod_{i=1}^{n} \exp{[S_i]\theta_i})M$$

where S_i refers to the screw axis w.r.t the space frame {s}
M is the transformation matrix of the end-effector when the joint displacements are all zeros

In the end-effector frame {b} [2]:-

$$T(\theta) = T_{sb} = M(\prod_{i=1}^{n} \exp{[B_i]\theta_i})$$

where B_i refers to the screw axis w.r.t the body frame {b}
M is the transformation matrix of the end-effector when the joint displacements are all zeros

Jacobian:
We know from velocity kinematics that:

$$\nu = J(\theta)\dot{\theta}$$

Inorder to evaluate the jacobian, we can follow two methods [2]:

Space Jacobian (J_s):-

The space Jacobian $J_s(\theta)$ is defined by

$$\mathcal{V}_s = J_s(\theta)\dot{\theta}$$

$$\text{where} \quad J_s(\theta) = [J_{s1}\ J_{s2}(\theta)\ \cdots\ J_{sn}(\theta)] \in \mathbb{R}^{6\times n}$$

$$\text{with} \quad J_{s1} = \mathcal{S}_1$$

$$\text{and} \quad J_{si}(\theta) = [\mathrm{Ad}_{e^{[\mathcal{S}_1]\theta_1}\cdots e^{[\mathcal{S}_{i-1}]\theta_{i-1}}}]\mathcal{S}_i \text{ for } i = 2, \ldots, n.$$

Body Jacobian (J_b):-

The body Jacobian $J_b(\theta)$ is defined by

$$\mathcal{V}_b = J_b(\theta)\dot{\theta}$$

$$\text{where} \quad J_b(\theta) = [J_{b1}(\theta)\ \cdots\ J_{bn-1}(\theta)\ J_{bn}] \in \mathbb{R}^{6\times n}$$

$$\text{with} \quad J_{bn} = \mathcal{B}_n$$

$$\text{and} \quad J_{bi}(\theta) = [\mathrm{Ad}_{e^{-[\mathcal{B}_n]\theta_n}\cdots e^{-[\mathcal{B}_{i+1}]\theta_{i+1}}}]\mathcal{B}_i \text{ for } i = 1, \ldots, n-1.$$

Different Inverse Kinematic Solvers:
In order to solve inverse kinematics problems for which the closed-loop forms are either not possible to derive or are very difficult to do so, various numerical methods have been developed that iteratively solve such equations. Most of the methods that can be used are based on root-finding algorithms.

Three different root-finding algorithms that can be used for this purpose:

1) Gauss-Seidel Method
2) Newton-Raphson Method
3) Broyden's Method

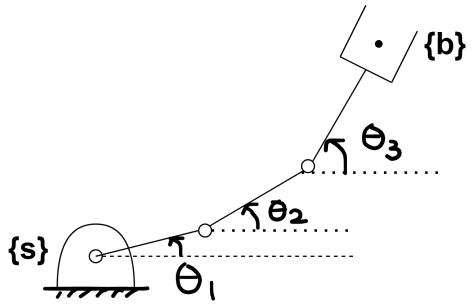
Gauss-Seidel method is an iterative method that is used to solve systems of linear equations. Although it is applicable to any matrix with non-zero elements on the diagonals, convergence to the solution is guaranteed only when the matrix is either strictly diagonally dominant, symmetric, or positive definite in nature.

Although the Gauss-Seidel method is better than *Newton-Raphson method* in terms of memory requirement and computation time per iteration, it overall takes more time to converge as it takes more no of iterations to do so. Also it is less reliable and is suitable only in smaller systems, while most of the larger systems use Newton-Raphson method for computation.

Newton-Raphson starts with an initial guess at the solution and uses the tangent of the current iteration to improve this estimate step-by-step. One disadvantage of this method is the requirement of the initial guess being close to the solution for convergence. However in robotic applications where we need to generate a trajectory for the end-effector as the inputs to be provided are quite close, this method is highly suitable for the job.

Broyden's method is a quasi-Newton method for finding the roots in k-variables. Its idea is to compute the whole Jacobian only at the first iteration and then perform rank-one updates in subsequent iterations. Broyden's method is typically faster than Newton-Raphson method and is also less sensitive to perturbations.

Newton Raphson Inverse Kinematic Solver:



The Forward Kinematics Problem can be stated as:-

$$\theta \rightarrow f(\theta)$$

On the other hand, the inverse kinematics problem deals with:-

$$Given\ x_d\ find\ \theta_d\ such\ that,$$
$$f(\theta_d) = x_d$$
$$\Rightarrow x_d - f(\theta_d) = 0$$

where x_d represents the desired body frame {b} position while θ_d represents the required joint displacements to achieve this configuration

The above problem is equivalent to finding the root of the function $g(\theta) = x_d - f(\theta)$. We can solve such a problem using a root-finding algorithm like *Newton-Raphson*.

The Newton-Raphson method can be summarized in the following steps:-

1) Plot the graph $x_d - f(\theta)$.

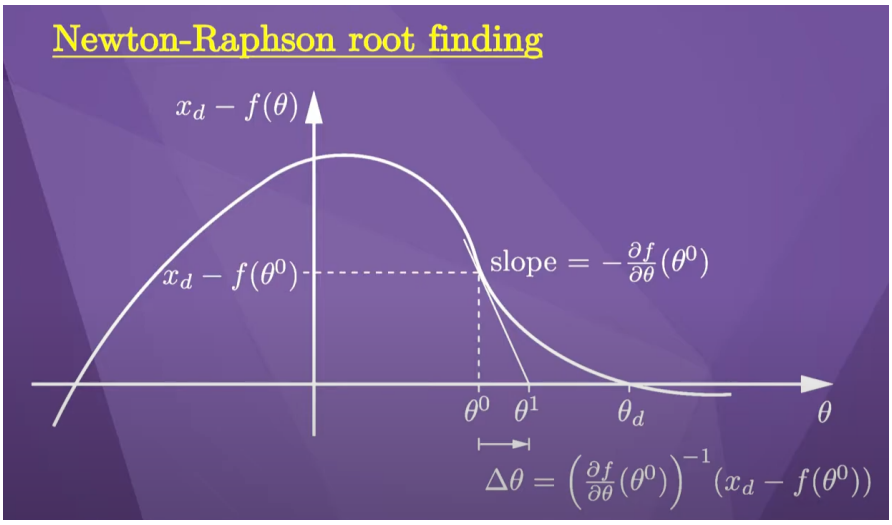
2) Choose an initial guess at the solution and call it θ_0. Mark this point on the graph.

3) Construct the tangent at θ_0 and make it intersect on the horizontal axis. This is the value of our estimate.

4) At this intersection point, construct a vertical line to intersect the plot at a point.

5) Repeat steps 1) to 4) till convergence is reached.

The below schematic depicts the steps followed in Newton-Raphson method clearly:-



Mathematically, we can represent this algorithm as:

$$\theta^1 = \theta^0 + (-1/m) * (x_d - f(\theta^0))$$

$$where\ m = \frac{\partial(x_d - f(\theta))}{\partial\theta}\ at\ \theta^0$$

As one might guess, this algorithm may fail to converge or even diverge when the initial estimate chosen is not close to the root desired.

We can extend this algorithm in the case of a robotic manipulator by writing the Taylor Series expansion as follows:

$$x_d = f(\theta_d) = f(\theta^i) + \frac{\partial f}{\partial \theta}|_{\theta^i}(\theta_d - \theta^i) + higher\ order\ terms$$

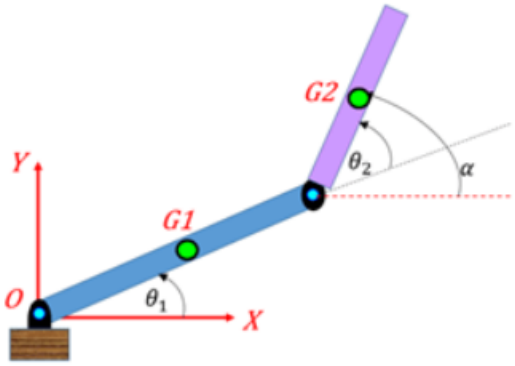
$$where\ \frac{\partial f}{\partial \theta}|_{\theta^i} = J(\theta^i)\ and\ \theta_d - \theta^i = \Delta\theta$$

By ignoring the higher order terms (*h.o.t*) we get the following equation:

$$\Delta\theta = J^{-1}(\theta^i)(x_d - f(\theta^i))$$

Implementing Numerical IK on a 2R robot

Refer to the following figure for relating with the two approaches to IK that follow:



Geometrical Approach:
In this approach, given the final end-effector coordinates (X_E,Y_E) we need to find the joint space angles required to achieve the desired configuration.
Fixing a frame at joint-1 we have,

$$X_E = l_1 cos(\theta 1) + l_2 cos(\theta 1 + \theta 2)$$
$$Y_E = l_1 sin(\theta 1) + l_2 sin(\theta 1 + \theta 2)$$

Now, we can find the Jacobian used for error calculation as follows:

$$J(\theta) = J(\theta 1, \theta 2) = \frac{\partial f}{\partial \theta} = \frac{\partial f}{\partial \theta 1}\dot{\theta 1} + \frac{\partial f}{\partial \theta 2}\dot{\theta 2}$$

where $f(\theta)$ is the function mapping (X_E, Y_E) to (θ_1, θ_2)

Screw Theory Approach:
In this approach given the desired transformation matrix (T_E) from base to end-effector we find the required joint-space angles as follows:

$$\mathcal{B}_1 = \begin{bmatrix} 0 & 0 & 1 & 0 & I1 + I2 & 0 \end{bmatrix}^T$$

$$\mathcal{B}_2 = \begin{bmatrix} 0 & 0 & 1 & 0 & I2 & 0 \end{bmatrix}^T$$

where $\mathcal{B}_1$,$\mathcal{B}_2$ are the screw-axes of joints-1 & 2 w.r.t the body frame {b} respectively

Now finding the components of the manipulator jacobian of joints J_1 & J_2,

$$J_{b1} = \left[Adj(\exp(-[\mathcal{B}_2]\theta_2)) \right] \mathcal{B}_1$$

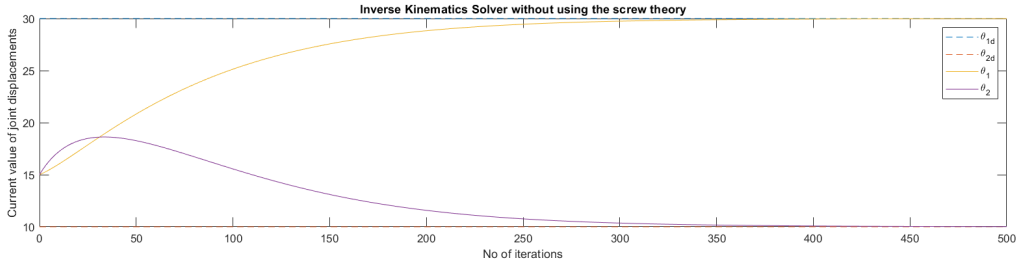
$$J_{b2} = \mathcal{B}_2$$

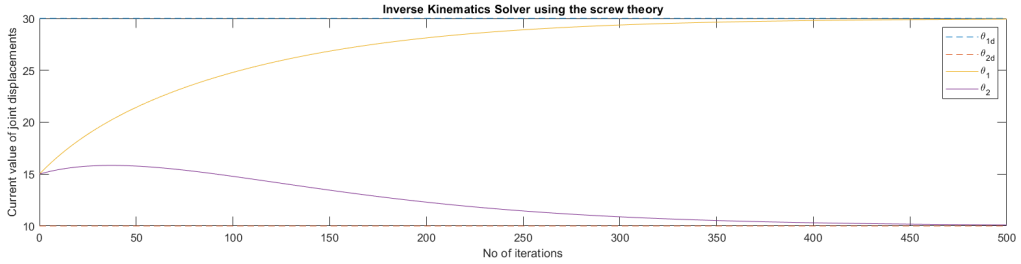
The overall jacobian is as follows:

$$\Rightarrow J = \begin{bmatrix} J_{b1} & J_{b2} \end{bmatrix}$$

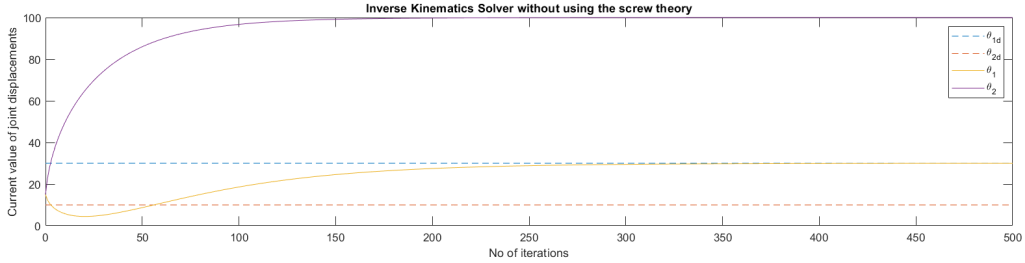
Results:

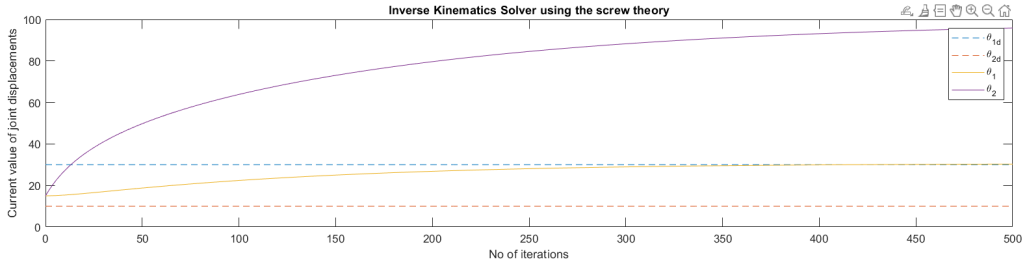
When desired joint angles are: θ_{1d} = 30°, θ_{2d} = 10°





When desired joint angles are: θ_{1d} = 30°, θ_{2d} = 100°


Inverse Kinematics Solver without using the screw theory
Current value of joint displacements
No of iterations
θ_{1d}
θ_{2d}
θ_1
θ_2


Inverse Kinematics Solver using the screw theory
Current value of joint displacements
No of iterations
θ_{1d}
θ_{2d}
θ_1
θ_2

Implementing trajectory generation on a 2R robot

Implementation of some basic trajectories:
Once a specific IK solver is completed, the remaining job is to provide a time series of end-effector positions as inputs to obtain the time history joint displacements required to generate a particular trajectory.

In this project, I have given the following input trajectories:

1) Circular Trajectory: To check whether the algorithm works well in generating smooth, continuous end-effector paths.

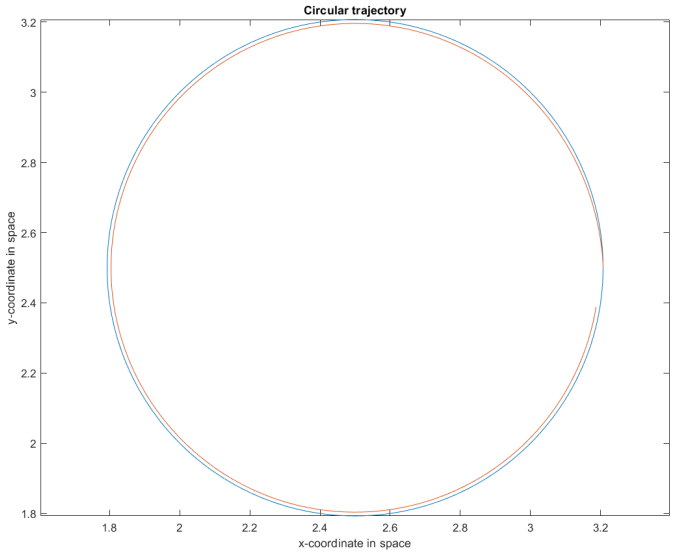
2) Straight-line trajectory: To check whether it performs well for a linear variation of inputs.

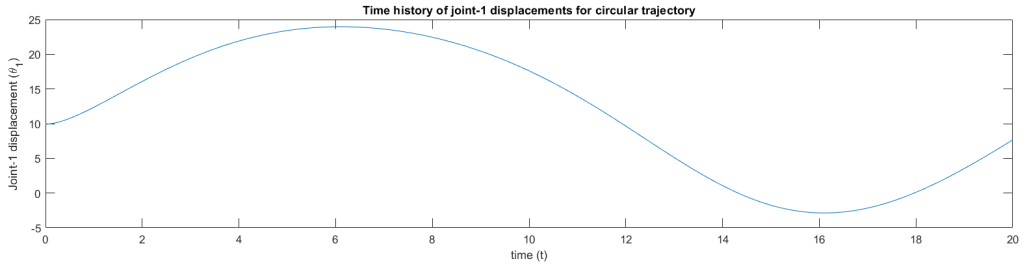
3) Square Trajectory: To check how the algorithm performs at the sharp edges where the components of end-effector velocity change drastically.

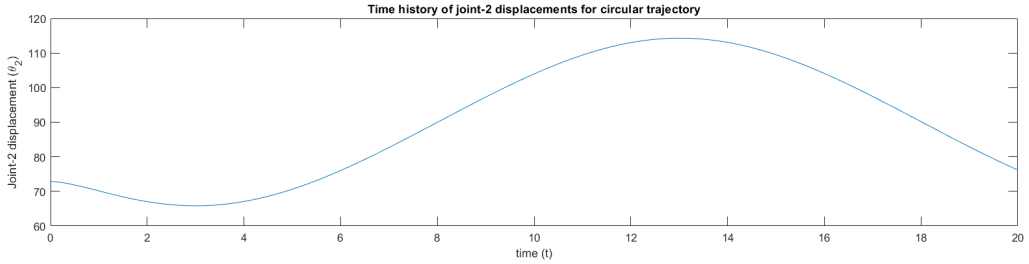
Yet to be implemented:
Given a certain set of via points, generate the trajectory to move through these via points.

Note: In the below results, a very low sampling rate of 10 samples/sec is chosen for testing purposes.

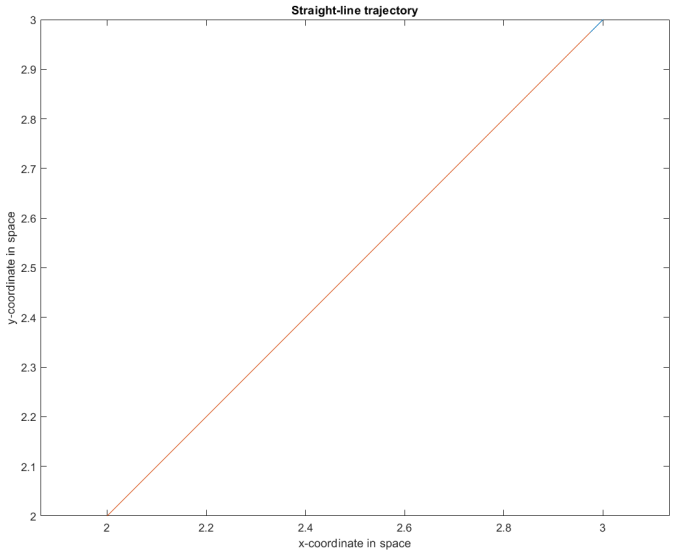
Results:
Circular Trajectory:

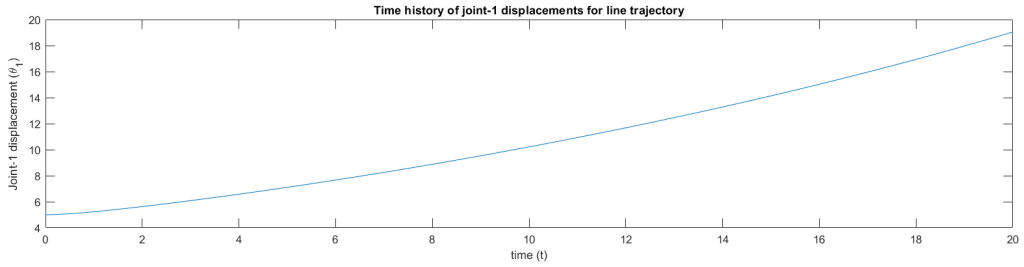


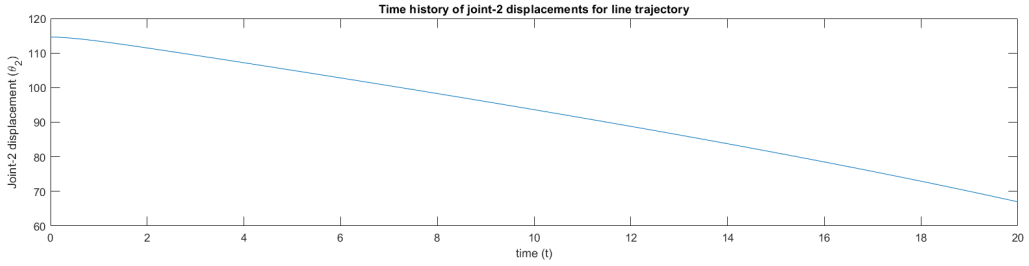




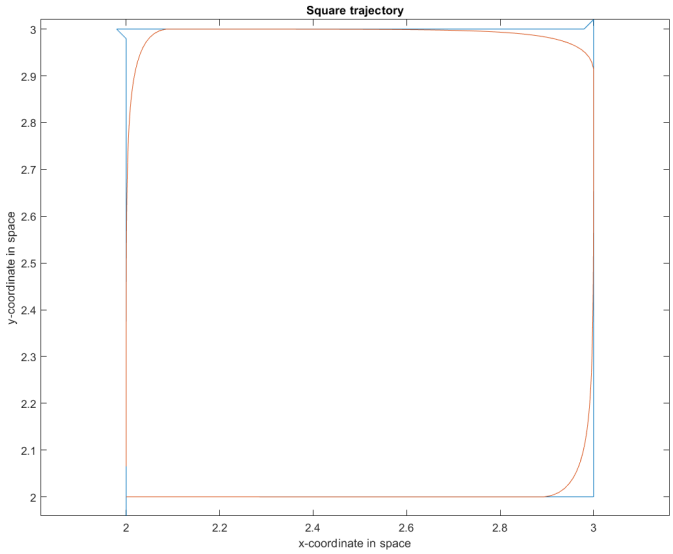
Straight-line Trajectory:

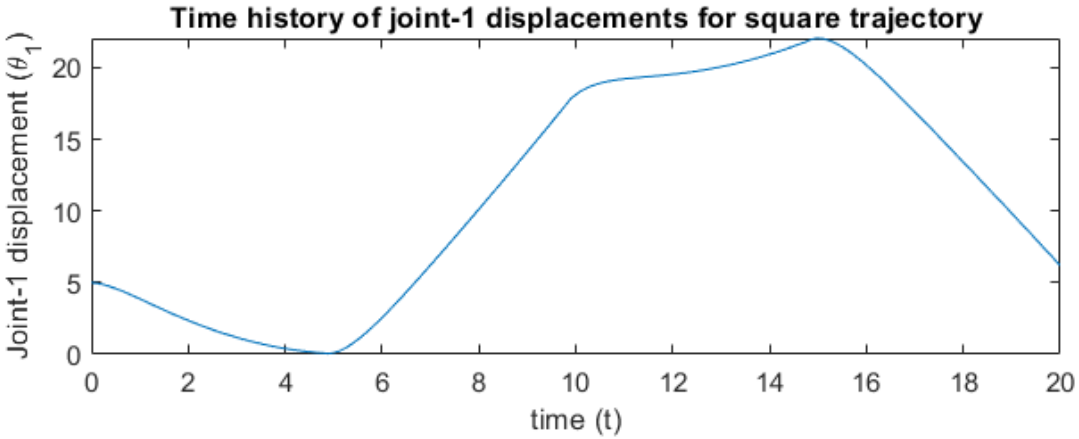


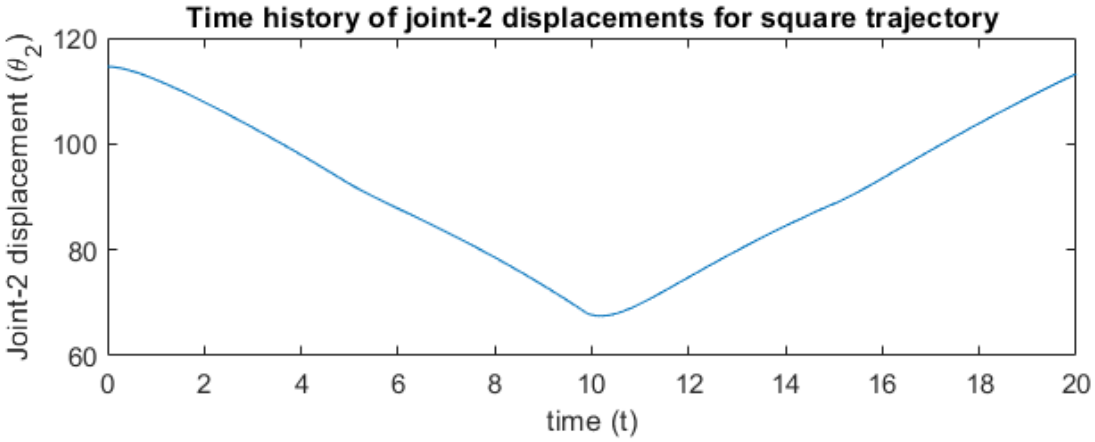




Square Trajectory:

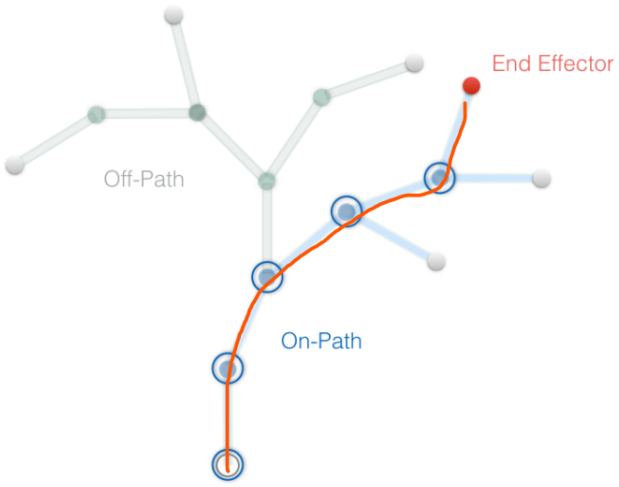

Square trajectory
y-coordinate in space
x-coordinate in space
3
2.9
2.8
2.7
2.6
2.5
2.4
2.3
2.2
2.1
2
2
2.2
2.4
2.6
2.8
3


Time history of joint-1 displacements for square trajectory
Joint-1 displacement (θ_1)
time (t)
0
5
10
15
20
0
2
4
6
8
10
12
14
16
18
20


Time history of joint-2 displacements for square trajectory
Joint-2 displacement (θ_2)
time (t)
120
100
80
60
0
2
4
6
8
10
12
14
16
18
20

Spatial tree robot

The jacobian computation in the case of a tree-type robot can be simplified to tracing out the *effective manipulator* in play for moving a specific end-effector which is equivalent to computing the dfs (depth-first-search) nodes of the manipulator. See the figure below for better understanding:



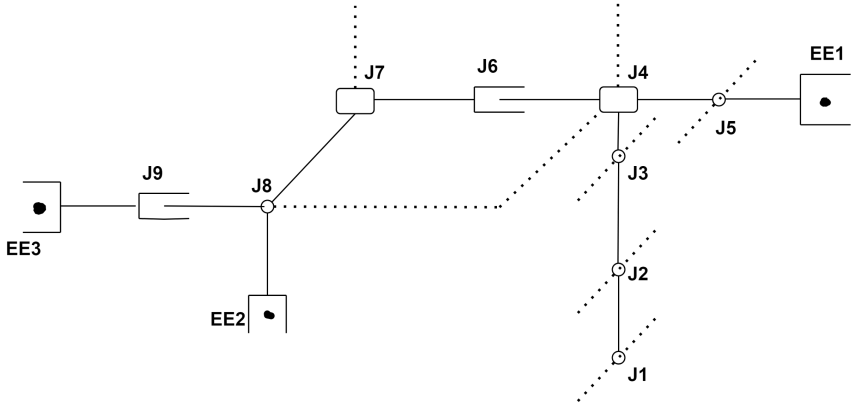
Here the *effective manipulator* is highlighted in red color

After finding out the *effective single-chain manipulator* to be considered for analysis, the problem reduces to simply solving the inverse kinematics for the *chain of links*. This can be done by using the equations for spatial transformation matrices, jacobians, etc. that were developed in the beginning of this report.

Implementation of IK on a tree-type robot

Model of robot considered:

For testing out the control of a tree-type manipulator, I have taken the following branched-robot model.



The different axis orientations are as implied by the diagram above, while the dimensions are as listed below:

J_1J_2 = 2 m J_2J_3 = 3 m J_3J_4 = 1 m
J_4J_5 = 2 m J_4J_6 = 1 m J_6J_7 = 2 m
J_7J_8 = 3 m J_8J_9 = 6 m J_5E_1 = 2 m
J_8E_2 = 1 m J_9E_3 = 1 m

Approach used for implementation:

In this work, the tree-robot was modeled as a *graph* in MATLAB. This is done so as to make use of the *inbuilt dfs* functionality that comes with this data type. For our example, the *graph edges* and the *screw-axes* for each joint should be defined as follows:

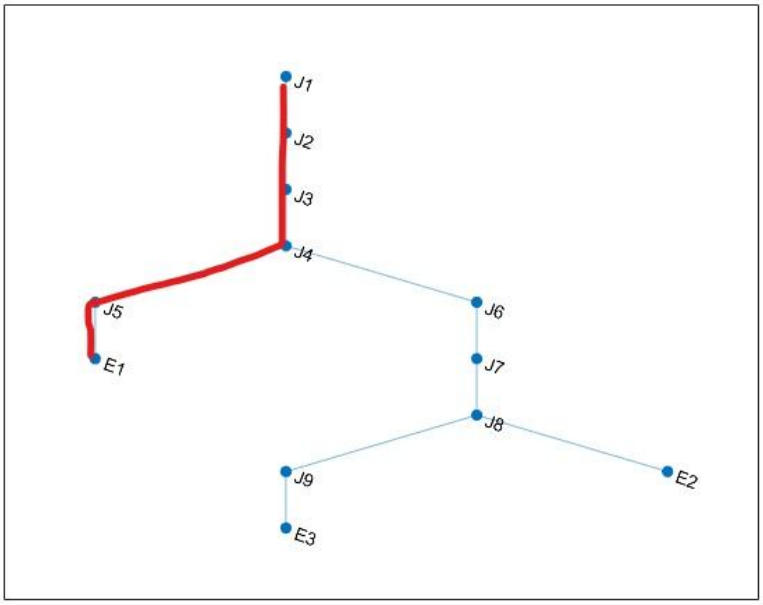
<u>Edges</u>:

J_1 -> J_2 J_2 -> J_3
J_3 -> J_4 J_4 -> J_5
J_4 -> J_6 J_6 -> J_7
J_7 -> J_8 J_8 -> J_9
J_5 -> E_1 J_8 -> E_2
J_9 -> E_3

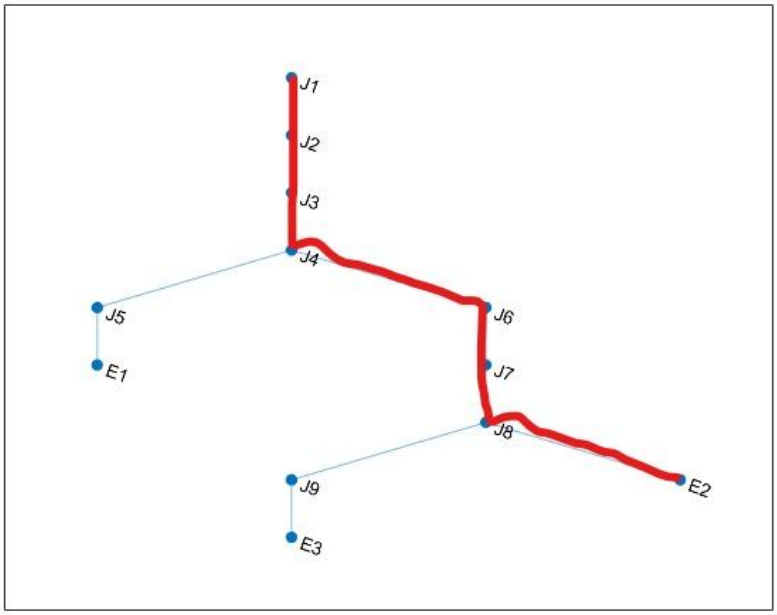
Screw-axes for joints:

J_1: $[1\ 0\ 0\ 0\ 0\ 0]^T$ J_2: $[1\ 0\ 0\ 0\ 2\ 0]^T$
J_3: $[1\ 0\ 0\ 0\ 5\ 0]^T$ J_4: $[0\ 0\ 1\ 0\ 0\ 0]^T$
J_5: $[1\ 0\ 0\ 0\ 6\ -2]^T$ J_6: $[0\ 0\ 0\ 0\ 1\ 0]^T$
J_7: $[0\ 0\ 1\ -3\ 0\ 0]^T$ J_8: $[0\ 0\ 1\ 3\ 6\ 0]^T$
J_9: $[0\ 0\ 0\ 0\ 6\ 0]^T$

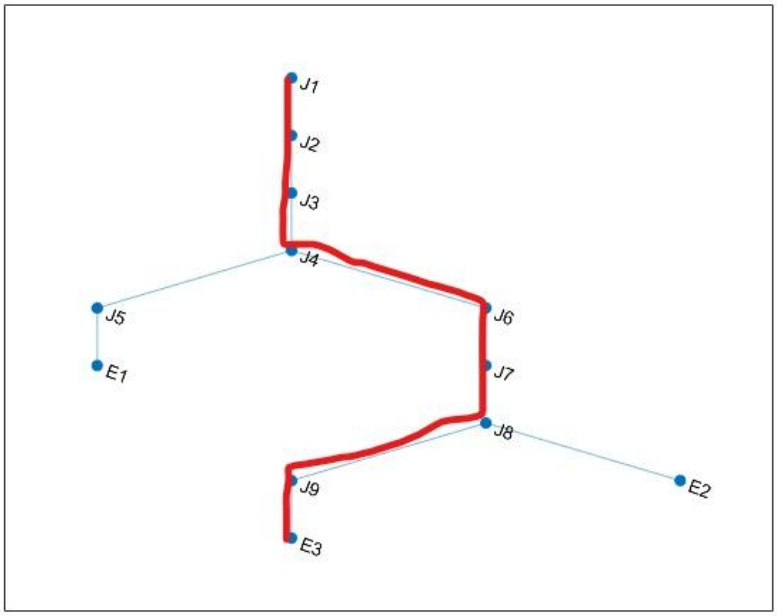
The *effective manipulators* to be considered for each end-effector are as follows:
End effector1:-



End effector2:-



End effector3:-



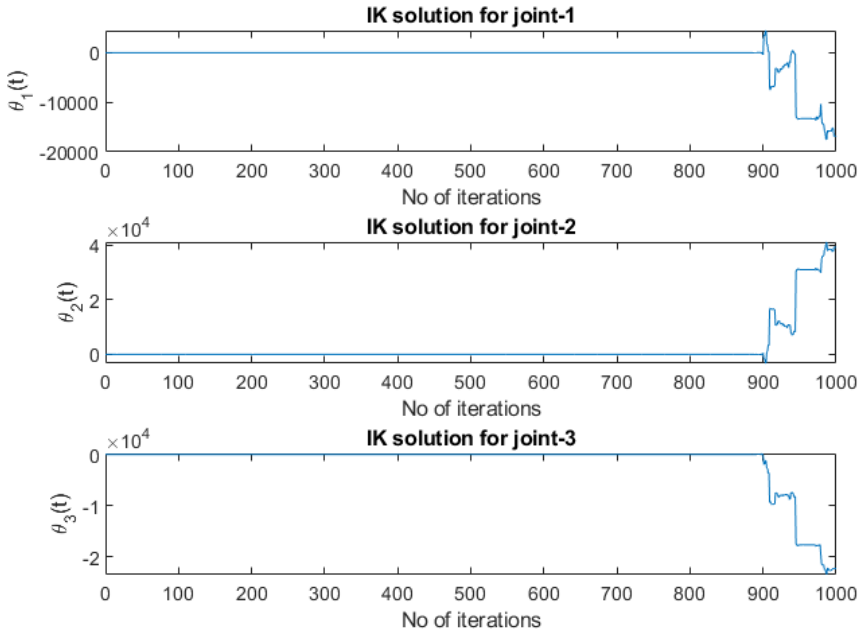
In this work, the *transformation* and *jacobian matrices* are defined in the space frame {s} and not in the body frame {b} because the robot

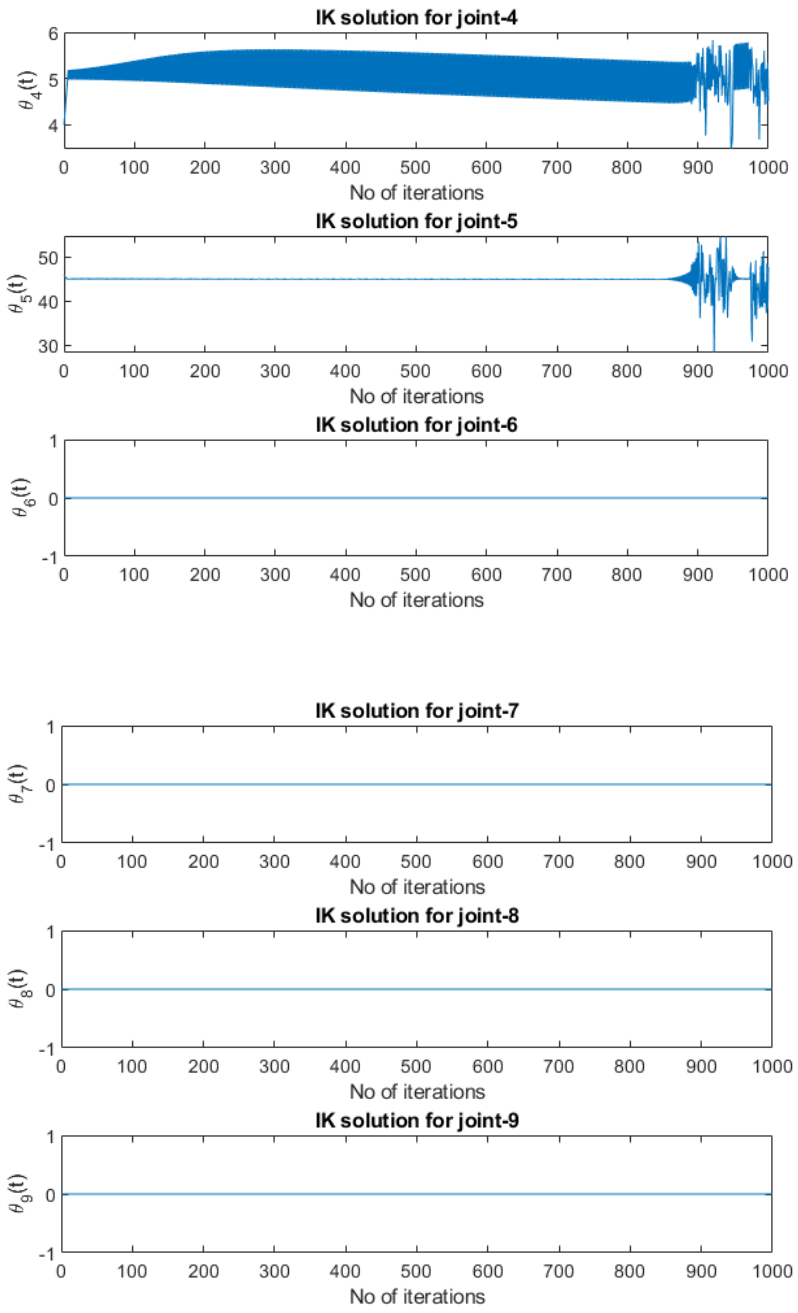
has more than one end-effector. A *damping coefficient* was used in the solver to *minimize the oscillatory behavior* of the solution.

Results:
The performance of the IK solver has been tested out for each end-effector with and without the use of damping. The observations are as follows:

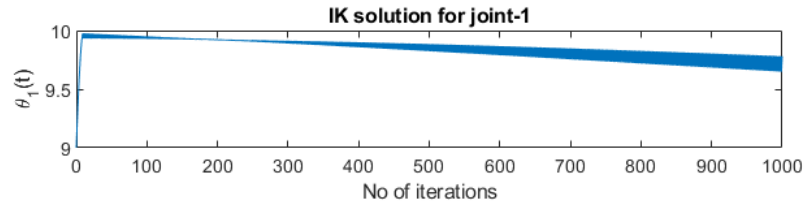
Without damping (Damping = 1):
In this case, the IK solver doesn't use a damping constant to *reduce* the *effect of the error terms* at every update.

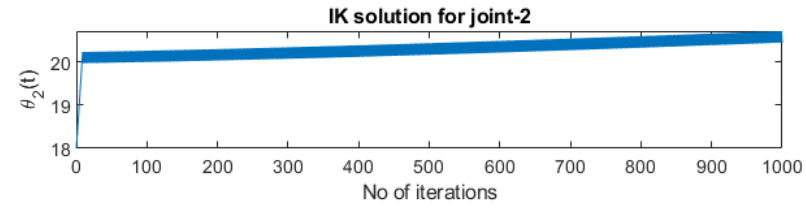
End-effector1:-

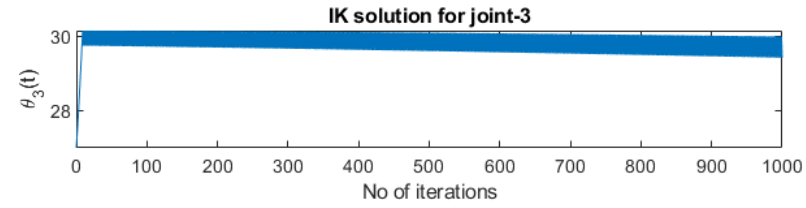


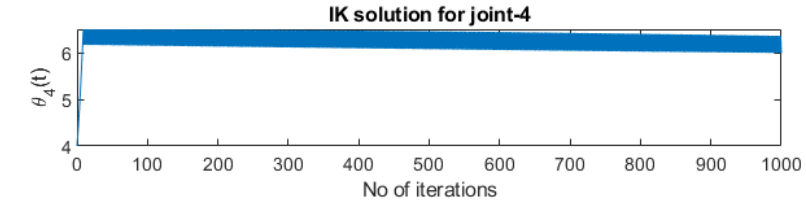


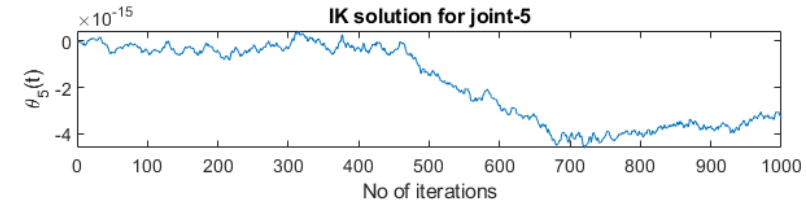
End-effector2:-

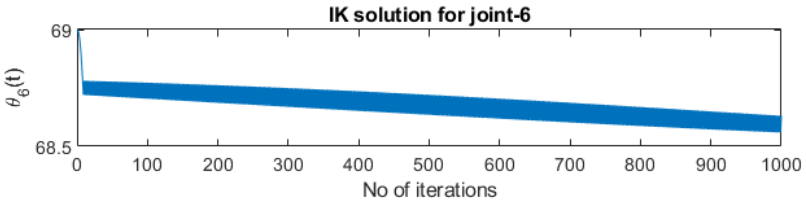

IK solution for joint-1
10
9.5
9
θ_1(t)
0
100
200
300
400
500
600
700
800
900
1000
No of iterations

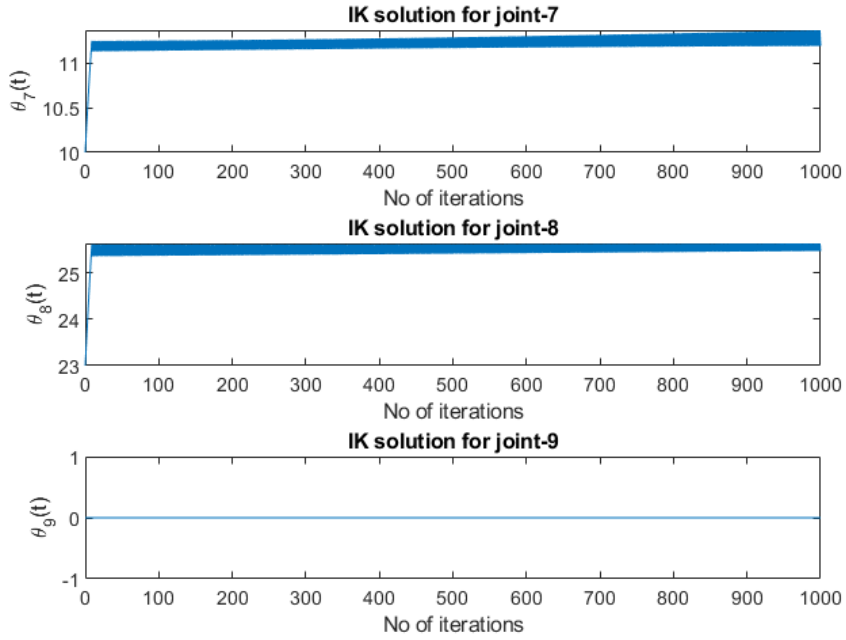

IK solution for joint-2
20
19
18
θ_2(t)
0
100
200
300
400
500
600
700
800
900
1000
No of iterations


IK solution for joint-3
30
28
θ_3(t)
0
100
200
300
400
500
600
700
800
900
1000
No of iterations

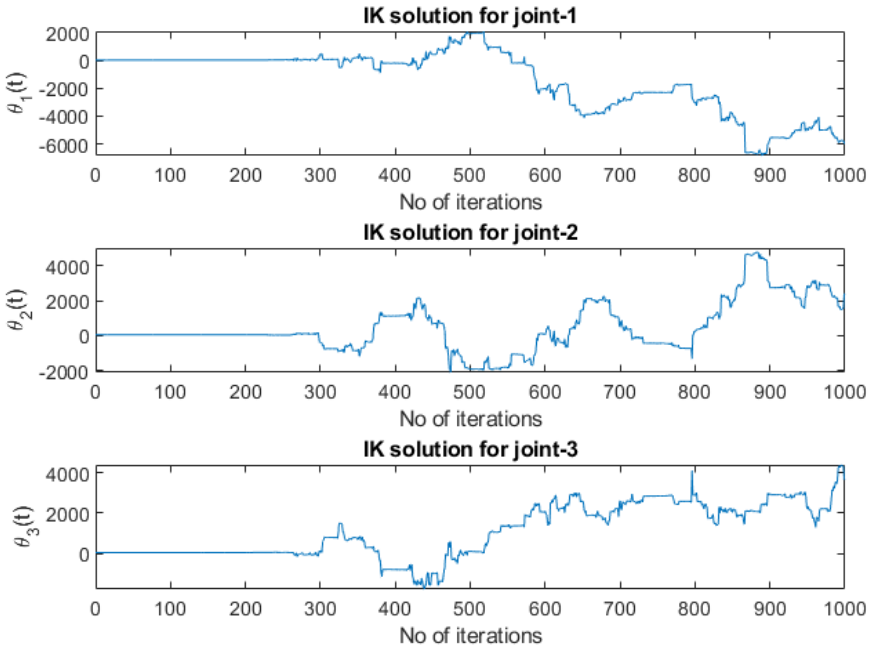

IK solution for joint-4
6
5
4
θ_4(t)
0
100
200
300
400
500
600
700
800
900
1000
No of iterations

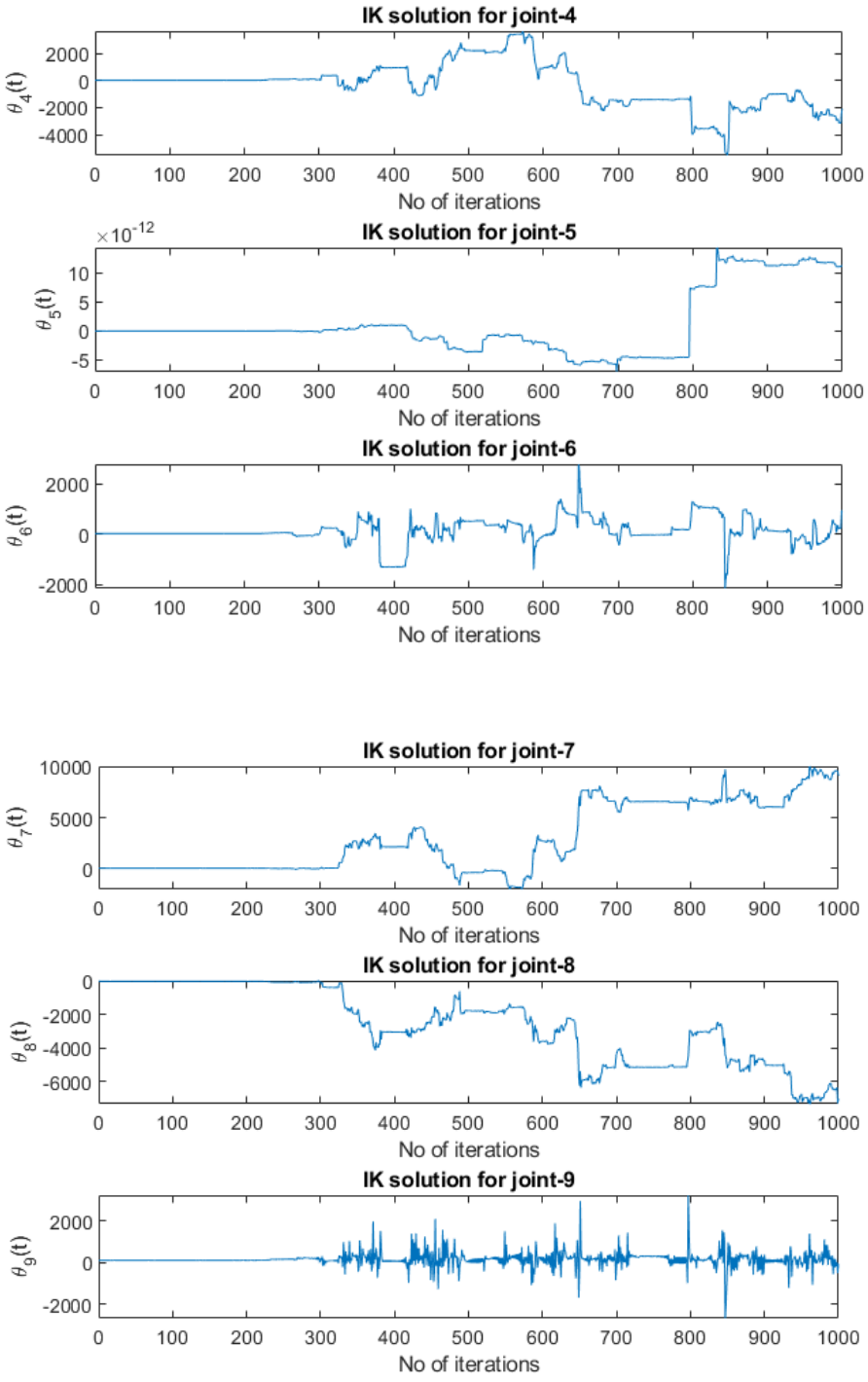

IK solution for joint-5
×10-15
0
-2
-4
θ_5(t)
0
100
200
300
400
500
600
700
800
900
1000
No of iterations


IK solution for joint-6
69
68.5
θ_6(t)
0
100
200
300
400
500
600
700
800
900
1000
No of iterations



End-effector3:-



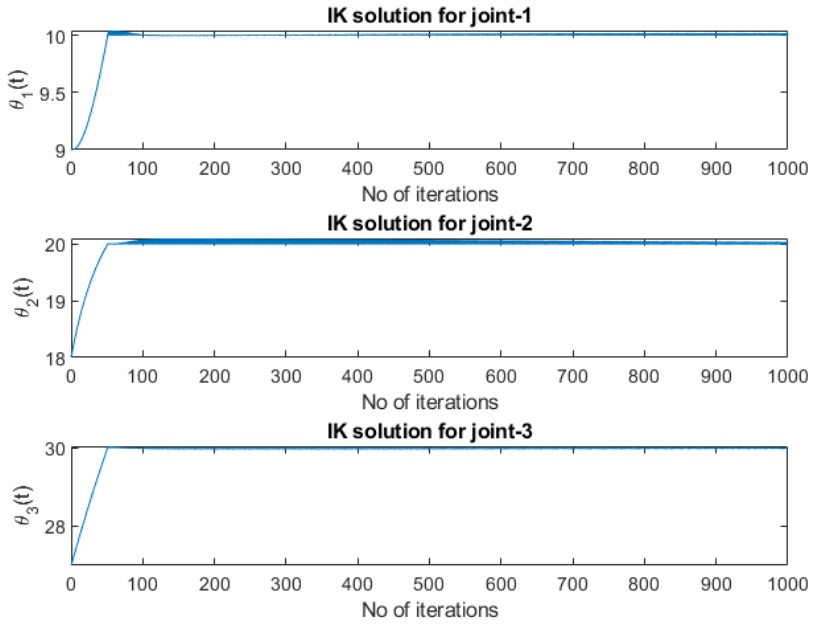


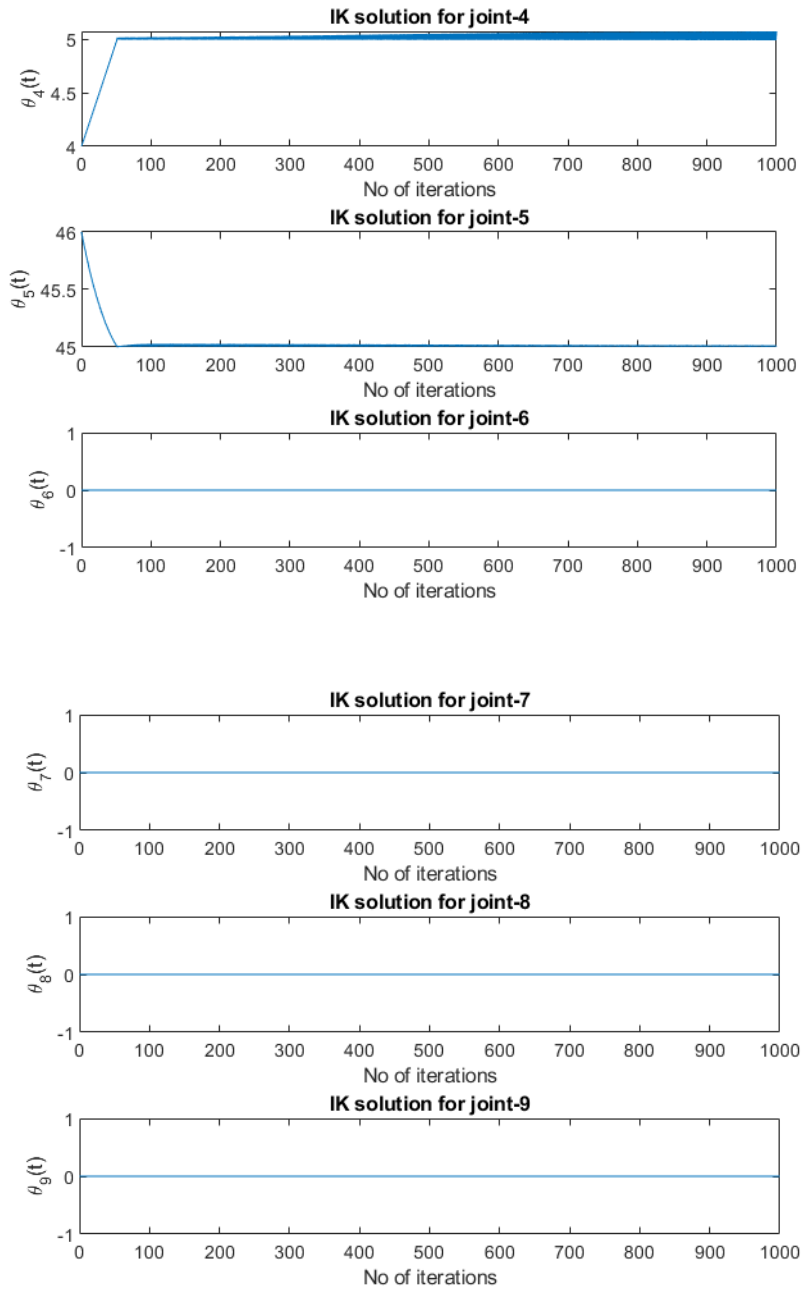
The solution is *very oscillatory* and *diverging* in nature. This could be due to the fact that the actual nature of the end-effector position vs joint displacement curve has a *very gradual slope* causing the error term in the Newton-Raphson Algorithm to be extremely high.

To fix this issue, I have introduced *damping* to *limit the error terms* getting added at each update of theta.

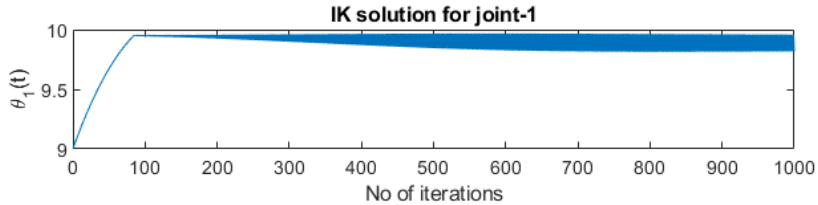
This improved the performance of the solver a bit with as low as “0.1 damping” and with “0.01 damping”, the performance was very good as shown:

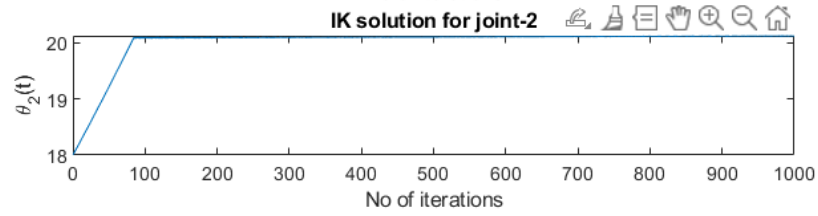
<u>Damping = 0.1</u>:
End-effector1:-

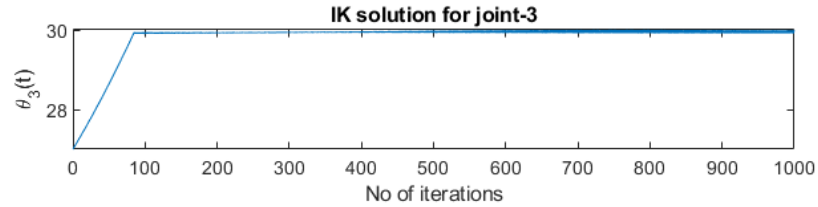


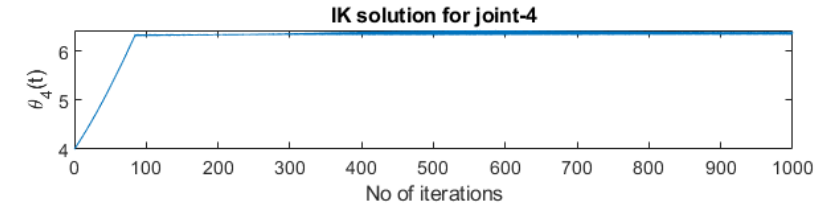


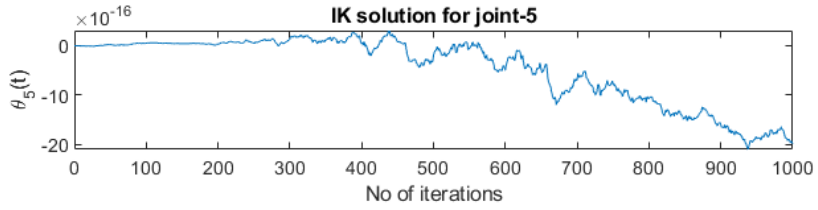
End-effector2:-

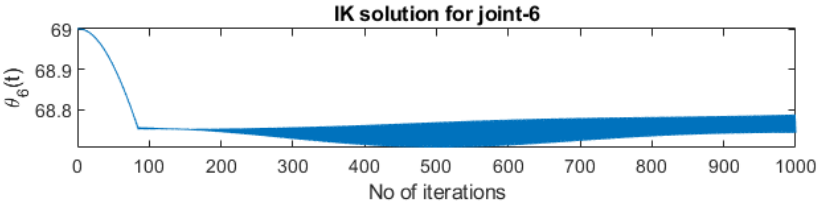

IK solution for joint-1
θ1(t)
10
9.5
9
0
100
200
300
400
500
600
700
800
900
1000
No of iterations

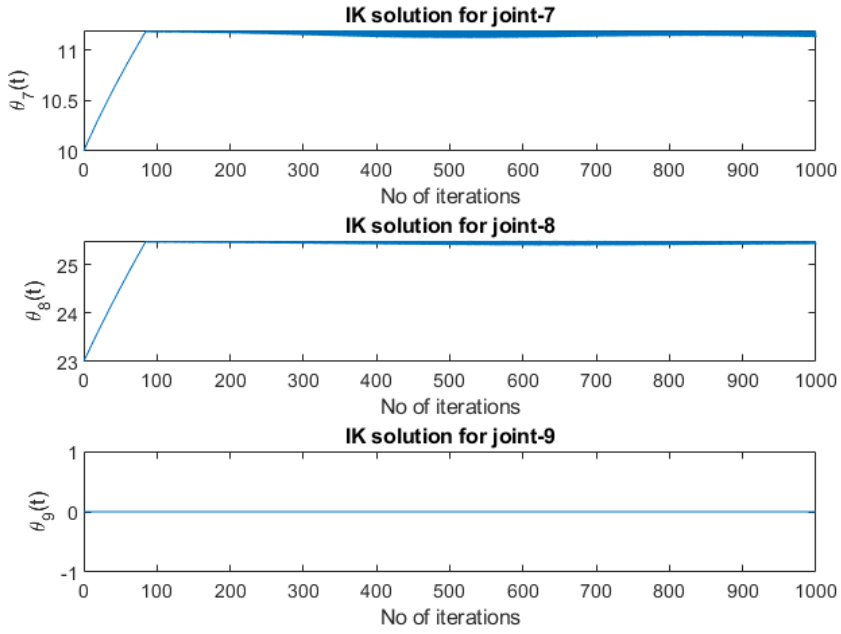

IK solution for joint-2
θ2(t)
20
19
18
0
100
200
300
400
500
600
700
800
900
1000
No of iterations


IK solution for joint-3
θ3(t)
30
28
0
100
200
300
400
500
600
700
800
900
1000
No of iterations

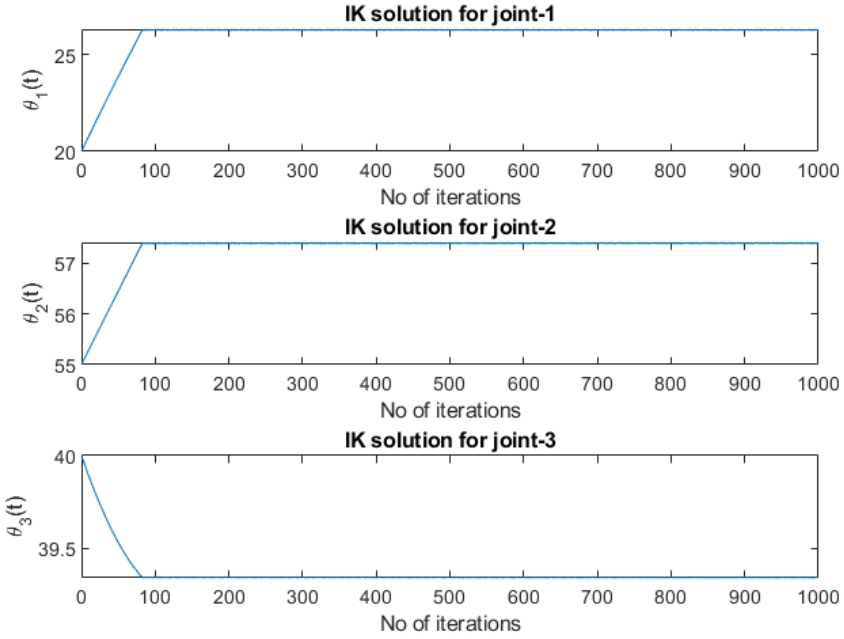

IK solution for joint-4
θ4(t)
6
5
4
0
100
200
300
400
500
600
700
800
900
1000
No of iterations

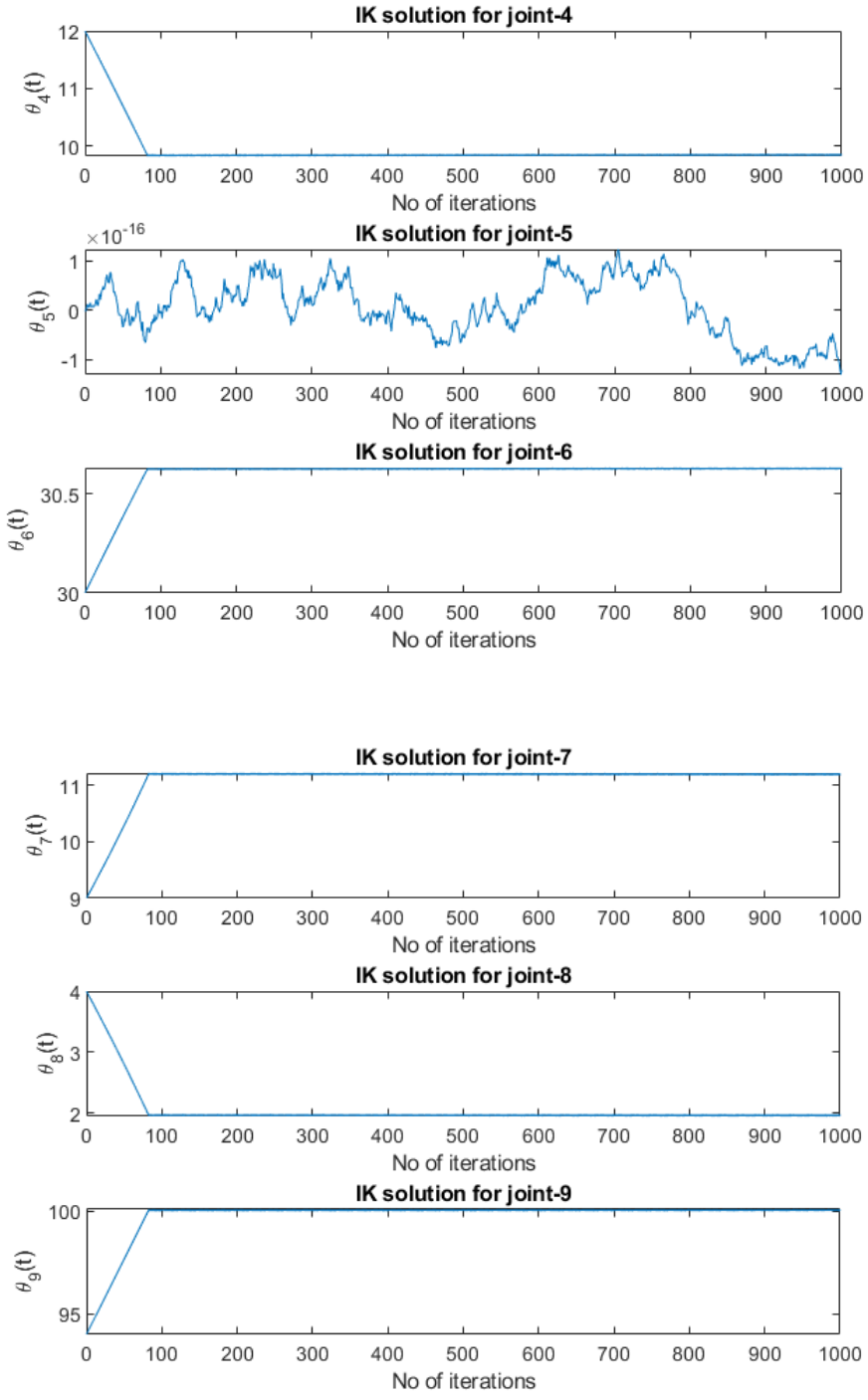

IK solution for joint-5
×10-16
θ5(t)
0
-10
-20
0
100
200
300
400
500
600
700
800
900
1000
No of iterations


IK solution for joint-6
θ6(t)
69
68.9
68.8
0
100
200
300
400
500
600
700
800
900
1000
No of iterations

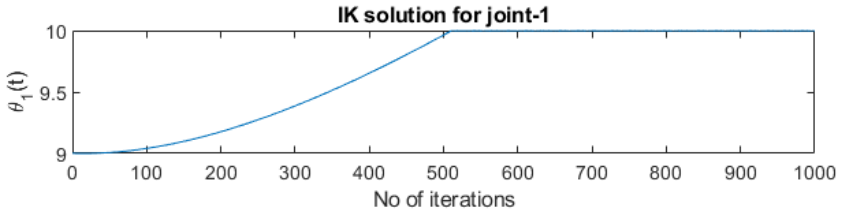


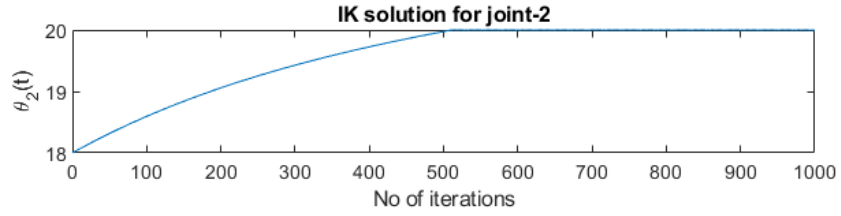
End-effector3:-

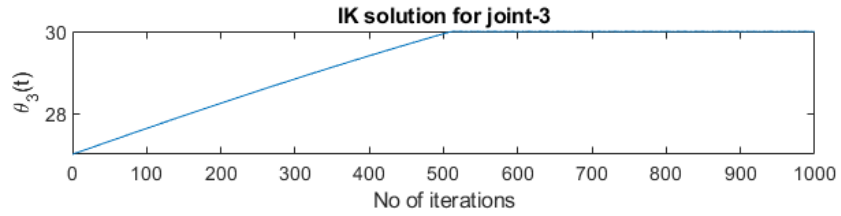


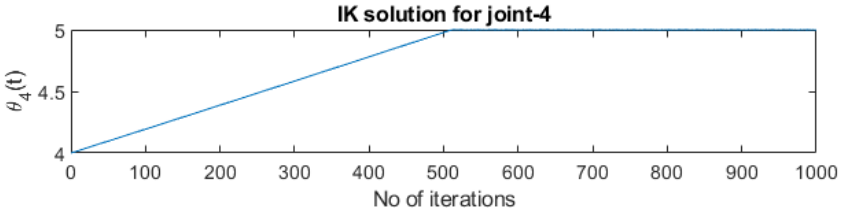


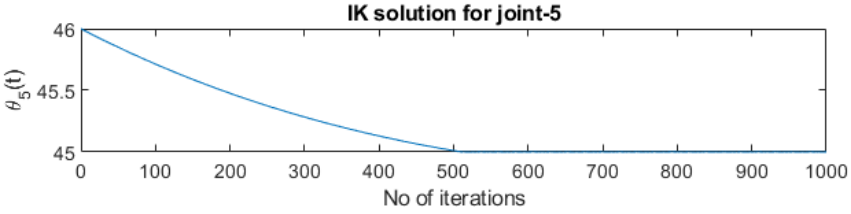
Damping = 0.01:
End-effector1:-

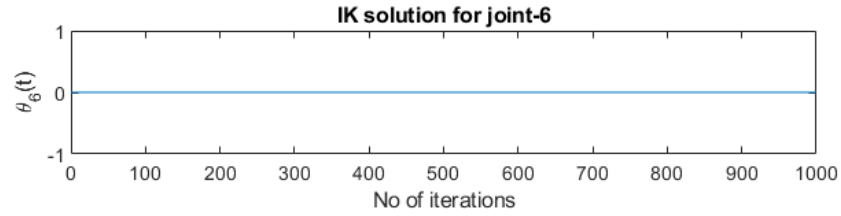

IK solution for joint-1
10
9.5
9
θ1(t)
0
100
200
300
400
500
600
700
800
900
1000
No of iterations

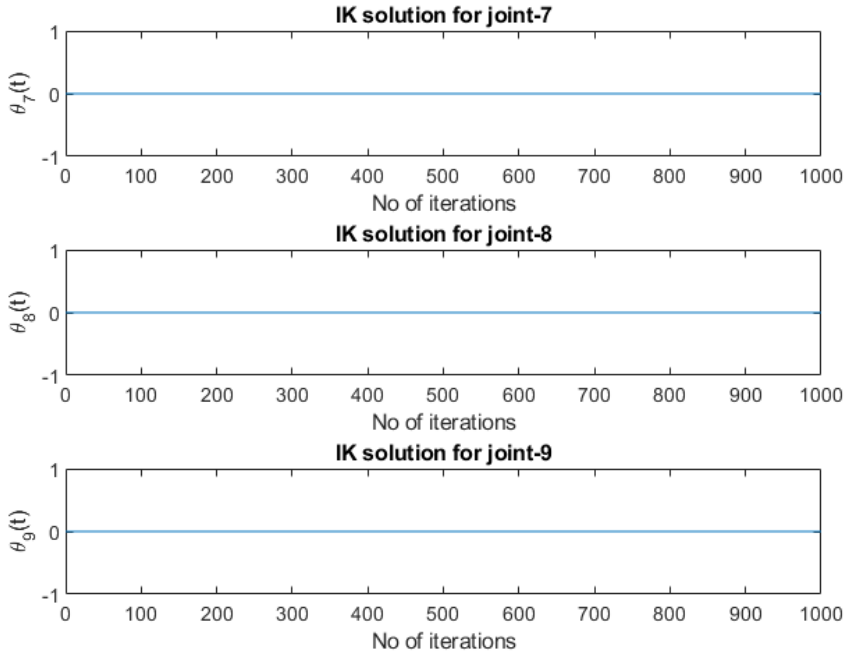

IK solution for joint-2
20
19
18
θ2(t)
0
100
200
300
400
500
600
700
800
900
1000
No of iterations


IK solution for joint-3
30
28
θ3(t)
0
100
200
300
400
500
600
700
800
900
1000
No of iterations

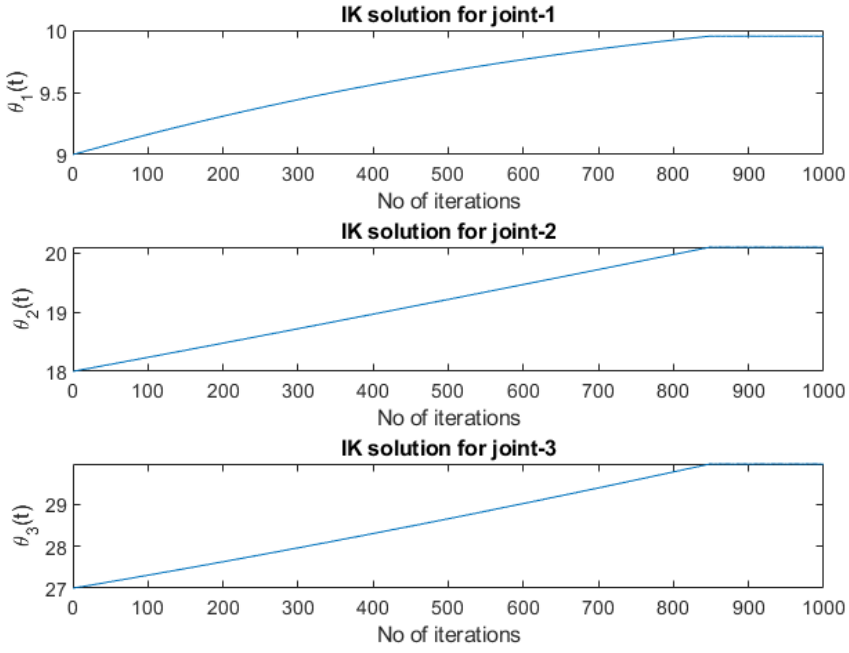

IK solution for joint-4
5
4.5
4
θ4(t)
0
100
200
300
400
500
600
700
800
900
1000
No of iterations

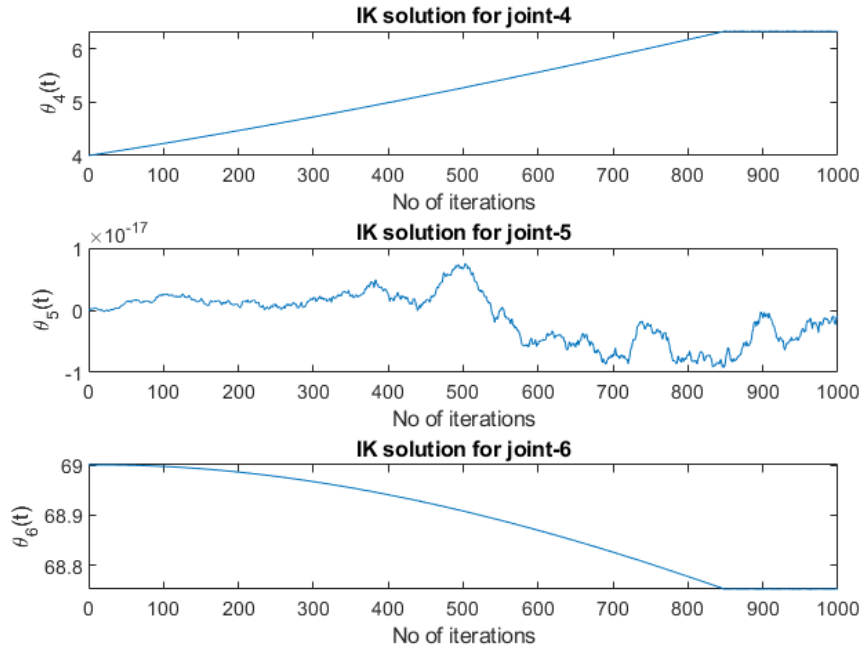

IK solution for joint-5
46
45.5
45
θ5(t)
0
100
200
300
400
500
600
700
800
900
1000
No of iterations

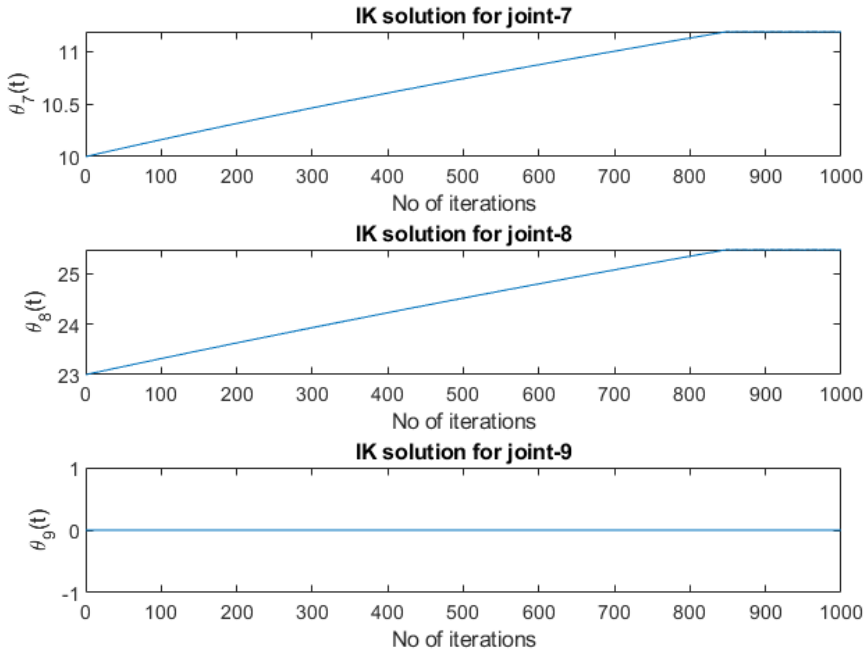

IK solution for joint-6
1
0
-1
θ6(t)
0
100
200
300
400
500
600
700
800
900
1000
No of iterations



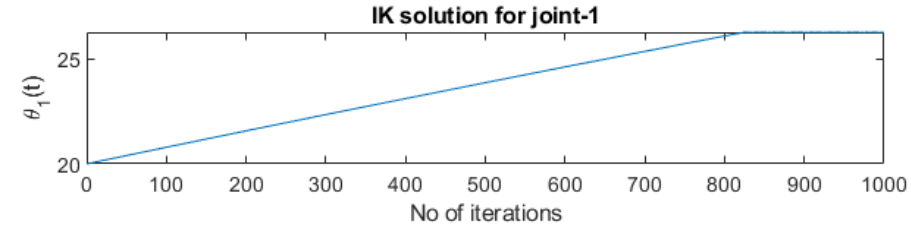
End-effector2:-

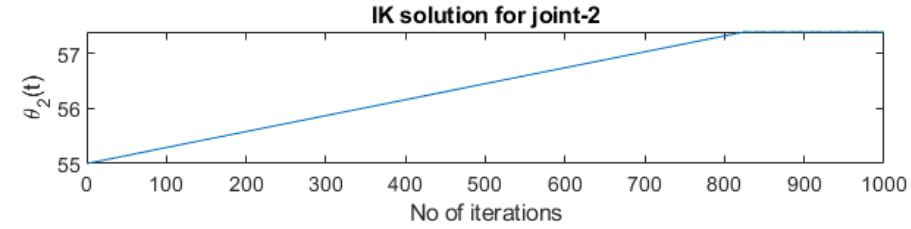


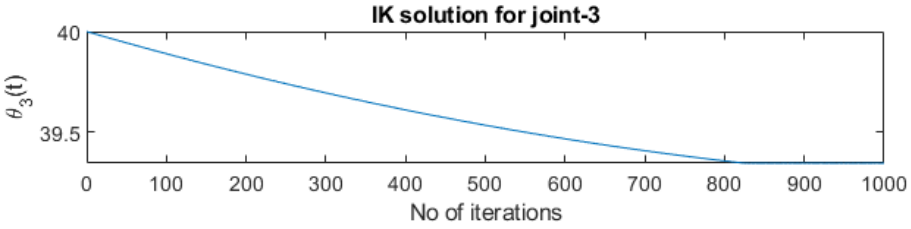


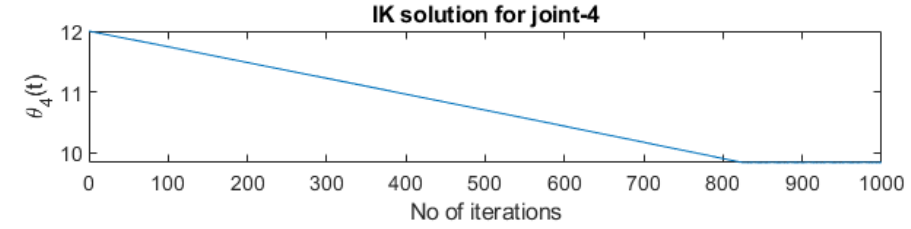


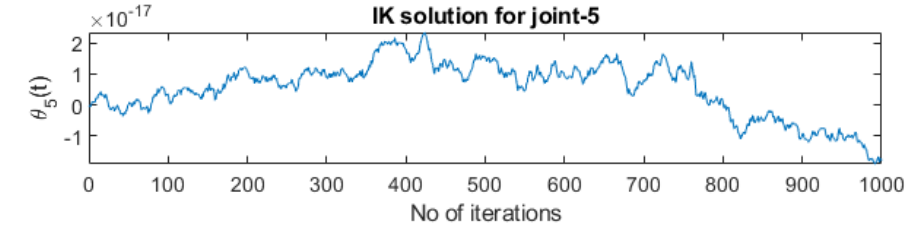
End-effector3:-

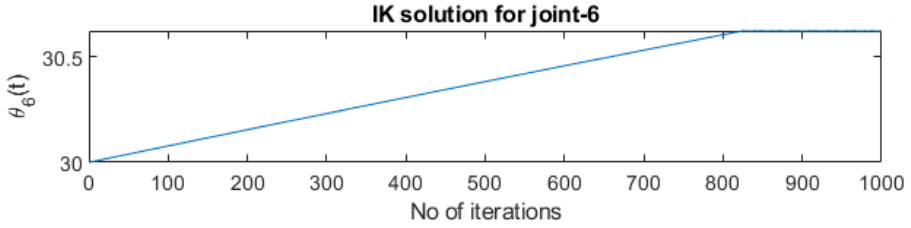

IK solution for joint-1
θ1(t)
25
20
0 100 200 300 400 500 600 700 800 900 1000
No of iterations

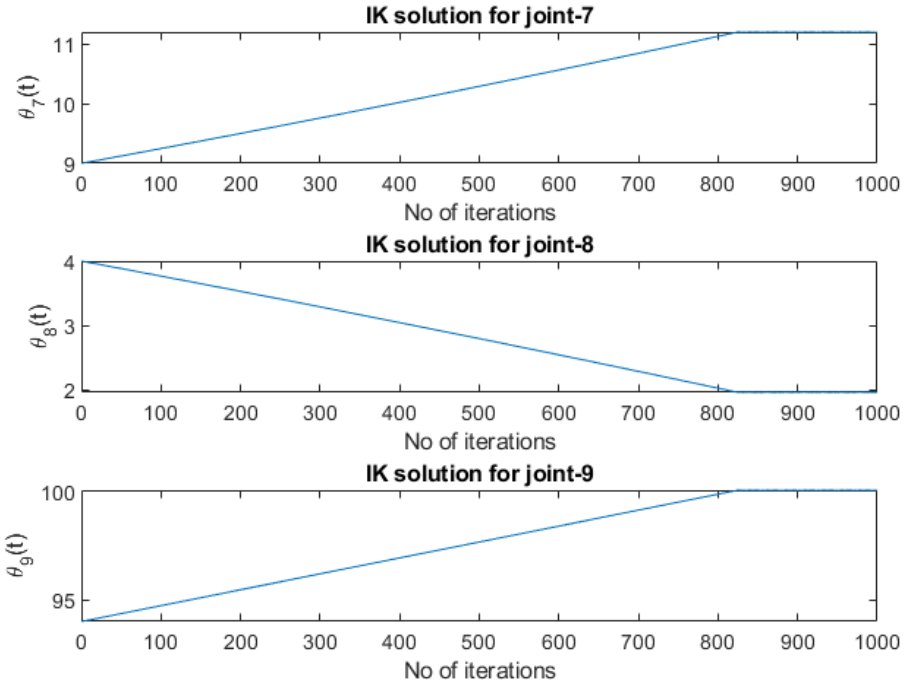

IK solution for joint-2
θ2(t)
57
56
55
0 100 200 300 400 500 600 700 800 900 1000
No of iterations


IK solution for joint-3
θ3(t)
40
39.5
0 100 200 300 400 500 600 700 800 900 1000
No of iterations


IK solution for joint-4
θ4(t)
12
11
10
0 100 200 300 400 500 600 700 800 900 1000
No of iterations


IK solution for joint-5
×10-17
θ5(t)
2
1
0
-1
0 100 200 300 400 500 600 700 800 900 1000
No of iterations


IK solution for joint-6
θ6(t)
30.5
30
0 100 200 300 400 500 600 700 800 900 1000
No of iterations


IK solution for joint-7
θ_7(t)
11
10
9
0
100
200
300
400
500
600
700
800
900
1000
No of iterations
IK solution for joint-8
θ_8(t)
4
3
2
0
100
200
300
400
500
600
700
800
900
1000
No of iterations
IK solution for joint-9
θ_9(t)
100
95
0
100
200
300
400
500
600
700
800
900
1000
No of iterations

Conclusion

In this project, I have learnt the math necessary to deal with complex robotic manipulators for which finding out the inverse kinematics analytically from the transformation matrix might not be possible or is very much tedious. In particular, the screw-theory was formulated and studied in the case of robotics. Later on, for solving the numerical inverse kinematics problem, a *root finding algorithm* called *Newton-Raphson* was employed.

Then, I have implemented the numerical inverse kinematics for a 2 DOF robot using Newton-Raphson approach by two methods: *Geometrical approach* and the *Screw-theory approach*. The experimental results of both these techniques were recorded. Using the developed *IK solver*, I have implemented the *trajectory generation* for the 2 DOF robot and calculated the *time-series of joint displacements* necessary to achieve such a trajectory.

Finally, I have coded the tree-type manipulator using *graph data structure* in MATLAB and solved the inverse kinematics problem again by using the Newton Raphson root finder. The results were recorded for the control of *different end-effectors* in play and also for *different levels of damping*.

References

1) Lynch, K. M., & Park, F. C. (2017). *Modern Robotics*. Retrieved from Northwestern: Center for Robotics and Biosystems: http://hades.mech.northwestern.edu/index.php/Modern_Robotics

2) Lynch, K. M., & Park, F. C. (2017). Modern Robotics: Mechanics, Planning, and Control. Cambridge University Press.

3) Nagrath, I. J., & Gopal, M. (2006). *Control Systems Engineering.* Delhi: New Age International (P) Ltd., Publishers.

Appendix (MATLAB Codes)

Numerical IK for 2R robot

Geometrical approach:

```
clear all;
close all;
clc;

%% Defining constant parameters and initial values

% Defining link lengths
l1 = 3;
l2 = 2;

% Desired joint angle displacements
theta_d = [30;10];

% Desired position vector
pos =
[l1*cosd(theta_d(1))+l2*cosd(theta_d(1)+theta_d(2));
l1*sind(theta_d(1))+l2*sind(theta_d(1)+theta_d(2))];

% Initial guess at solution
theta0 = [15; 15];

% % Iterations
% No of iterations
N = 500;
```

```
% Current value of theta
theta_curr = theta0;

% Array of thetas
theta = zeros(2,N+1);
theta(:,1) = theta0;

% Constant to damp the error in each iteration
damper = 0.9;

%% Algorithm to iteratively update the solution

for i = 1:N
    J_curr =
[-l1*sind(theta_curr(1))-l2*sind(theta_curr(1)+theta_cu
rr(2)), -l2*sind(theta_curr(1)+theta_curr(2));

l1*cosd(theta_curr(1))+l2*cosd(theta_curr(1)+theta_curr
(2)), l2*cosd(theta_curr(1)+theta_curr(2))];
    pos_curr =
[l1*cosd(theta_curr(1))+l2*cosd(theta_curr(1)+theta_cur
r(2));
l1*sind(theta_curr(1))+l2*sind(theta_curr(1)+theta_curr
(2))];

    theta_curr = theta_curr -
damper*pinv(J_curr)*(pos_curr - pos);
    theta(:,i+1) = theta_curr;
end

%% Results:
```

```
subplot(2,1,1);
plot(0:N,30*ones(N+1,1),'--','DisplayName','Expected
joint1 angle');
legend
hold on;
plot(0:N,10*ones(N+1,1),'--','DisplayName','Expected
joint2 angle');
plot(0:N,theta(1,:),'DisplayName','theta1');
plot(0:N,theta(2,:),'DisplayName','theta2');
title("Inverse Kinematics Solver without using the
screw theory");
xlabel("No of iterations");
ylabel("Current value of joint displacements");
legend("\theta_{1d}", "\theta_{2d}", "\theta_1",
"\theta_2");
```

Screw theory approach:

```
%% Defining constant parameters and initial values

% Defining link lengths
l1 = 3;
l2 = 2;

% Initial guess at the solution
theta0 = [15, 15];

% Desired joint displacements
theta_d = [30;10];
```

```
% Desired position & orientation
pos =
[l1*cosd(theta_d(1))+l2*cosd(theta_d(1)+theta_d(2));
l1*sind(theta_d(1))+l2*sind(theta_d(1)+theta_d(2))];
orien = theta_d(1)+theta_d(2);

% Desired transformation of end effector w.r.t space
frame
Tsd = [cosd(orien), -sind(orien), 0, pos(1);
       sind(orien), cosd(orien), 0, pos(2);
       0, 0, 1, 0;
       0, 0, 0, 1];

% Initial value of theta
theta_curr = theta0;

% Constant to damp the error in each iteration
damper = 0.5;

%% Algorithm to compute the jacobian and finally
iteratively converge to the solution

N = 500;
theta = zeros(N+1,2);
theta(1,:) = theta0;
for i = 1:N
    % Current orientation
    orien_curr = theta_curr(1)+theta_curr(2);

    % Current position
```

```
    pos_curr =
[l1*cosd(theta_curr(1))+l2*cosd(theta_curr(1)+theta_cur
r(2)),
l1*sind(theta_curr(1))+l2*sind(theta_curr(1)+theta_curr
(2))];

    % Current transformation of {b} w.r.t {s}
    curr_Tsb = [cosd(orien_curr), -sind(orien_curr), 0,
pos_curr(1);
                sind(orien_curr), cosd(orien_curr), 0,
pos_curr(2);
                0, 0, 1, 0;
                0, 0, 0, 1];

    % Overall transformation matrix
    trans_mat = pinv(curr_Tsb)*Tsd;

    % Angular velocity in the body frame {b}
    curr_angvel_box =
(1/(2*sind(orien_curr)))*(trans_mat(1:3, 1:3) -
transpose(trans_mat(1:3, 1:3)));

    % Linear velocity in the body frame {b}
    curr_vel = (eye(3)/deg2rad(orien_curr) -
curr_angvel_box/2 +(1/deg2rad(orien_curr) -
0.5*cotd(orien_curr/2))*(curr_angvel_box^2))*trans_mat(
1:3,4);

    % Twist in the body frame {b}
    curr_twist = [curr_angvel_box(3,2);
curr_angvel_box(1,3); curr_angvel_box(2,1); curr_vel];
```

```
    % Screw axes of joints w.r.t body frame {b}
    screw1_curr = [0;0;1;0;l1+l2;0];
    screw2_curr = [0;0;1;0;l2;0];

    % Jacobian computation
    J_b = zeros(6,2);

    % Jacobian component for joint-1
    % Skew-symmetric representation of S_omega
    omega_skew_curr =
[0,-screw2_curr(3),screw2_curr(2);

screw2_curr(3),0,-screw2_curr(1);

-screw2_curr(2),screw2_curr(1),0];

    % Body transformation matrix due to rotation of
joint-2
    trans_mat_curr =
inv([eye(3)+sind(theta_curr(2))*omega_skew_curr+(1-cosd
(theta_curr(2)))*omega_skew_curr^2,
(eye(3)*deg2rad(theta_curr(2))+(1-cosd(theta_curr(2)))*
omega_skew_curr+(deg2rad(theta_curr(2)-sind(theta_curr(
2))))*omega_skew_curr^2)*screw2_curr(4:6);
        zeros(1,3), 1]);

    % Skew-matrix of position vector
    pos_skew_curr =
[0,-trans_mat_curr(3,4),trans_mat_curr(2,4);
```

```
trans_mat_curr(3,4),0,-trans_mat_curr(1,4);

-trans_mat_curr(2,4),trans_mat_curr(1,4),0];

    % Adjoint Matrix of joint-2 w.r.t body {b}
    adjoint_mat_curr = [trans_mat_curr(1:3,1:3),
zeros(3,3);

pos_skew_curr*trans_mat_curr(1:3,1:3),
trans_mat_curr(1:3,1:3)];

    % Jacobian component for joint-1
    J_b(:,1) = adjoint_mat_curr*screw1_curr;

    % Jacobian component for joint-2
    J_b(:,2) = screw2_curr;

    theta_curr = theta_curr +
damper*transpose(pinv(J_b)*curr_twist);
    theta(i+1,:) = theta_curr;
end


%% Results:

subplot(2,1,2);
plot(0:N,30*ones(N+1,1),'--');
hold on;
plot(0:N,10*ones(N+1,1),'--');
plot(0:N,theta(:,1));
plot(0:N,theta(:,2));
```

```
title("Inverse Kinematics Solver using the screw
theory");
xlabel("No of iterations");
ylabel("Current value of joint displacements");
legend("\theta_{1d}", "\theta_{2d}", "\theta_1",
"\theta_2");
```

Trajectory planning for 2R robot

IK solver code:

```
function [theta1, theta2] = twoR_IK_fun(l1, l2,
theta_i, pos_d, N)

    theta_curr = theta_i;

    for i = 1:N
        J_curr =
[-l1*sind(theta_curr(1))-l2*sind(theta_curr(1)+theta_cu
rr(2)), -l2*sind(theta_curr(1)+theta_curr(2));

l1*cosd(theta_curr(1))+l2*cosd(theta_curr(1)+theta_curr
(2)), l2*cosd(theta_curr(1)+theta_curr(2))];
        pos_curr =
[l1*cosd(theta_curr(1))+l2*cosd(theta_curr(1)+theta_cur
r(2));
l1*sind(theta_curr(1))+l2*sind(theta_curr(1)+theta_curr
(2))];

        theta_curr = theta_curr -
```

```
pinv(J_curr)*(pos_curr - pos_d);
    end

    theta1 = theta_curr(1);
    theta2 = theta_curr(2);
end
```

Other important functions' codes:

```
% position function
function [x,y] = position(l1, l2, t1, t2)
    x = l1*cosd(t1) + l2*cosd(t1 + t2);
    y = l1*sind(t1) + l2*sind(t1 + t2);
end
```

```
% function to find initial values of theta given the
position
function [theta1, theta2] = solution_t1t2(l1, l2, x, y,
tp1, tp2)
    syms t1 t2;

    eqns = [l1*cos(t1) + l2*cos(t1 + t2) == x,
l1*sin(t1) + l2*sin(t1 + t2) == y];

    [ts1, ts2] = solve(eqns, [t1, t2]);

    ts1 = rad2deg(double(ts1));
    ts2 = rad2deg(double(ts2));

    v = sum(abs([ts1 - tp1, ts2 - tp2]), 2);
```

```
    [val, ind] = min(v);

    theta1 = ts1(ind);
    theta2 = ts2(ind);
end
```

Trajectory planning code:

```
clear all;
close all;
clc;

%% Defining some constant parameters and initial values

% Defining link lengths
l1 = 3;
l2 = 2;

% Time vector
space = 0.1;
t = 0:space:20;


%% Desired position and velocity vectors

% Straight-line trajectory
% Straight line from (2, 2) to (3, 3)
pos_d_line = [2 + (1/t(length(t)))*t; 2 +
(1/t(length(t)))*t];
```

```
% Square trajectory
% (2, 2) -> (3, 2) -> (3, 3) -> (2, 3) -> (2, 2)
step = 1/t(floor(length(t)/4));
pos_d_square = zeros(2,length(t));
for i = 1:length(t)
    if(i <= floor(length(t)/4))
        pos_d_square(:,i) = [2 + step*t(i); 2];
    elseif(i <= floor(length(t)/2))
        pos_d_square(:,i) = [3; 2 + step*(t(i) -
t(floor(length(t)/4)))];
    elseif(i <= floor(3*length(t)/4))
        pos_d_square(:,i) = [3 - step*(t(i) -
t(floor(length(t)/2))); 3];
    else
        pos_d_square(:,i) = [2; 3 - step*(t(i) -
t(floor(3*length(t)/4)))];
    end
end

% Circle trajectory
% Traces a circle centred at (2, 2) of radius = 1
pos_d_circle = [2.5 + (1/sqrt(2))*cos(2*pi*(1/20)*t);
2.5 + (1/sqrt(2))*sin(2*pi*(1/20)*t)];


%% Calculating the joint velocities required to move
the end-effector in specific trajectory

% Computation time (no of iterations)
ct = 10;
```

```
% Initial theta estimates
tg1 = 15;
tg2 = 15;

% Circular trajectory
[t1_c, t2_c] = solution_t1t2(l1, l2, pos_d_circle(1,1),
pos_d_circle(2,1), tg1, tg2);
theta_i_c = [t1_c; t2_c];

theta_a_circle = [theta_i_c, zeros(2, length(t)-1)];
pos_a_circle = [pos_d_circle(:,1),
zeros(2,length(t)-1)];
for i = 2:length(t)
    [t1_c, t2_c] = twoR_IK_fun(l1, l2,
theta_a_circle(:,i-1), pos_d_circle(:,i), ct);
    theta_a_circle(:,i) = [t1_c; t2_c];

    [p1_c,p2_c] = position(l1, l2, theta_a_circle(1,i),
theta_a_circle(2,i));
    pos_a_circle(:,i) = [p1_c; p2_c];
end

% Square trajectory
[t1_sq, t2_sq] = solution_t1t2(l1, l2, pos_d_square(1),
pos_d_square(2), tg1, tg2);
theta_i_sq = [t1_sq; t2_sq];

theta_a_square = [theta_i_sq, zeros(2, length(t)-1)];
pos_a_square = [pos_d_square(:,1),
zeros(2,length(t)-1)];
```

```
for i = 2:length(t)
    [t1_sq, t2_sq] = twoR_IK_fun(l1, l2,
theta_a_square(:,i-1), pos_d_square(:,i), ct);
    theta_a_square(:,i) = [t1_sq; t2_sq];

    [p1_sq,p2_sq] = position(l1, l2,
theta_a_square(1,i), theta_a_square(2,i));
    pos_a_square(:,i) = [p1_sq; p2_sq];
end

% Straight-line trajectory
[t1_l, t2_l] = solution_t1t2(l1, l2, pos_d_line(1),
pos_d_line(2), tg1, tg2);
theta_i_l = [t1_l; t2_l];

theta_a_line = [theta_i_l, zeros(2, length(t)-1)];
pos_a_line = [pos_d_line(:,1), zeros(2,length(t)-1)];
for i = 2:length(t)
    [t1_l, t2_l] = twoR_IK_fun(l1, l2,
theta_a_line(:,i-1), pos_d_line(:,i), ct);
    theta_a_line(:,i) = [t1_l; t2_l];

    [p1_l,p2_l] = position(l1, l2, theta_a_line(1,i),
theta_a_line(2,i));
    pos_a_line(:,i) = [p1_l; p2_l];
end

%% Plotting the results

% Circular Trajectory
figure;
```

```
subplot(2,1,1);
plot(t, theta_a_circle(1,:));
title('Time history of joint-1 displacements for
circular trajectory');
xlabel("time (t)");
ylabel("Joint-1 displacement (\theta_1)");

subplot(2,1,2);
plot(t, theta_a_circle(2,:));
title('Time history of joint-2 displacements for
circular trajectory');
xlabel("time (t)");
ylabel("Joint-2 displacement (\theta_2)");

figure;
plot(pos_d_circle(1,:), pos_d_circle(2,:));
title('Circular trajectory');
hold on
plot(pos_a_circle(1,:), pos_a_circle(2,:));
xlabel("x-coordinate in space");
ylabel("y-coordinate in space");
axis equal;

% Straight-line trajectory
figure;
subplot(2,1,1);
plot(t, theta_a_line(1,:));
title('Time history of joint-1 displacements for line
trajectory');
xlabel("time (t)");
ylabel("Joint-1 displacement (\theta_1)");
```

```
subplot(2,1,2);
plot(t, theta_a_line(2,:));
title('Time history of joint-2 displacements for line
trajectory');
xlabel("time (t)");
ylabel("Joint-2 displacement (\theta_2)");

figure;
plot(pos_d_line(1,:), pos_d_line(2,:));
title('Straight-line trajectory');
hold on
plot(pos_a_line(1,:), pos_a_line(2,:));
xlabel("x-coordinate in space");
ylabel("y-coordinate in space");
axis equal;

% Square trajectory
figure;
subplot(2,1,1);
plot(t, theta_a_square(1,:));
title('Time history of joint-1 displacements for square
trajectory');
xlabel("time (t)");
ylabel("Joint-1 displacement (\theta_1)");

subplot(2,1,2);
plot(t, theta_a_square(2,:));
title('Time history of joint-2 displacements for square
trajectory');
xlabel("time (t)");
```

```
ylabel("Joint-2 displacement (\theta_2)");

figure;
plot(pos_d_square(1,:), pos_d_square(2,:));
hold on
plot(pos_a_square(1,:), pos_a_square(2,:));
title('Square trajectory');
xlabel("x-coordinate in space");
ylabel("y-coordinate in space");
axis equal;
```

Numerical IK for tree-type robot

Codes for some important functions used:

```
function out_mat = adj_mat(trans_matrix)
    out_mat = [trans_matrix(1:3,1:3), zeros(3,3);

skew_mat(trans_matrix(1:3,4))*trans_matrix(1:3,1:3),
trans_matrix(1:3,1:3)];
end
```

```
% Function to evaluate the tree robot's jacobian given
the manipulator
% object, joint displacements, and the end-effector to
be considered.
function [out_mat] = jacobian(tree_robot, theta,
end_effector)

    % Initializing the tree-robot's node variables
    joint_name = tree_robot.Nodes.Name;
```

```
    screw_axes = tree_robot.Nodes.screwAxes;
    joint_type = tree_robot.Nodes.jointType;

    % To convert the literal end-effector to its index
in the graph nodes
    % so that we can use the inbuilt 'dfs' function
    pos = 1;
    for i = 1:length(joint_name)
        if(joint_name{i} == end_effector)
            pos = i;
            break;
        end
    end

    % The effective manipulator that needs to be
considered while
    % calculating the jacobian is given by the dfs
result from the
    % end-effector to the root of the tree
    dfsResult = dfsearch(tree_robot, pos);
    path = [pos];
    for i  = 2:length(dfsResult)
        path = [path, dfsResult(i)];

        if(dfsResult(i) == 1)
            break;
        end
    end

    % Initializing the manipulator jacobian
    out_mat = zeros(6, 9);
```

```
    % Calculating the manipulator jacobian
    out_mat(:,1) = screw_axes{path(length(path))};
    for i = flip(2:(length(path)-1))
        curr_trans_mat = eye(4);
        for j = flip((i+1):(length(path)))
            ind = path(j);
            curr_trans_mat =
curr_trans_mat*trans_mat(screw_axes{ind}, theta(ind));
        end

        out_mat(:,path(i)) =
adj_mat(curr_trans_mat)*screw_axes{path(i)};
    end
end
```

```
function [out_mat] = matLog(trans_matrix)
    theta = acosd((trace(trans_matrix) - 2)/2);

    angvel_skew =
(1/(2*sind(theta)))*(trans_matrix(1:3, 1:3) -
transpose(trans_matrix(1:3, 1:3)));

    vel = (eye(3)/deg2rad(theta) - angvel_skew/2
+(1/deg2rad(theta) -
0.5*cotd(theta/2))*(angvel_skew^2))*trans_matrix(1:3,4)
;

    out_mat = [angvel_skew(3,2); angvel_skew(1,3);
angvel_skew(2,1); vel];
end
```

```
function out_mat = skew_mat(vec)
    out_mat = [0, -vec(3), vec(2);
               vec(3), 0, -vec(1);
               -vec(2), vec(1), 0];
end
```

```
% Function to calculate the transformation matrix given
the screw-axis and
% amount of rotation
function out_mat = trans_mat(screw_axis, theta)
    omega_skew = skew_mat(screw_axis(1:3));

    out_mat = [eye(3) + sind(theta)*omega_skew + (1 -
cosd(theta))*(omega_skew^2), (eye(3)*deg2rad(theta) +
(1 - cosd(theta))*omega_skew + (deg2rad(theta) -
sind(theta))*(omega_skew^2))*screw_axis(4:6);
               zeros(1, 3), 1];
end
```

```
% Function to evaluate the configuration matrix of the
robot given the
% robot object, joint displacements, and the
end-effector to be considered.
function [out_mat] = trans_mat_manipulator(tree_robot,
theta, end_effector)

    % Initializing the tree-robot's node variables
    joint_name = tree_robot.Nodes.Name;
    screw_axes = tree_robot.Nodes.screwAxes;
```

```
    joint_type = tree_robot.Nodes.jointType;

    % To convert the literal end-effector to its index
in the graph nodes
    % so that we can use the inbuilt 'dfs' function
    pos = 1;
    for i = 1:length(joint_name)
        if(joint_name{i} == end_effector)
            pos = i;
            break;
        end
    end

    % The effective manipulator that needs to be
considered while
    % calculating the jacobian is given by the dfs
result from the
    % end-effector to the root of the tree
    dfsResult = dfsearch(tree_robot, pos);
    path = [pos];
    for i  = 2:length(dfsResult)
        path = [path, dfsResult(i)];

        if(dfsResult(i) == 1)
            break;
        end
    end

    % Iteratively evaluating the spatial transformation
matrix for the
    % chosen end-effector
```

```
    out_mat = eye(4);
    for i = flip(2:length(path))
        ind = path(i);
        out_mat = out_mat*trans_mat(screw_axes{ind},
theta(ind));
    end
end
```

IK solver code:

```
clearvars;
close all;
clc;


%% Defining the tree_robot in code

% Adjacency matrix for the tree
parent = {'J1', 'J2', 'J3', 'J4', 'J4', 'J6', 'J7',
'J8', 'J5', 'J8', 'J9'};
child =  {'J2', 'J3', 'J4', 'J5', 'J6', 'J7', 'J8',
'J9', 'E1', 'E2', 'E3'};

% Creating the tree-object
tree = graph(parent, child);

% Naming different links of the robot
tree.Edges.Names = {'L1'; 'L2'; 'L3'; 'L4'; 'L6'; 'L7';
'L8'; 'L10'; 'L5'; 'L9'; 'L11'};
```

```
% Indexing different nodes of the tree robot
tree.Nodes.index = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10; 11;
12];

% Defining the joints
tree.Nodes.jointType = ['R'; 'R'; 'R'; 'R'; 'R'; 'P';
'R'; 'R'; 'P'; 'N'; 'N'; 'N'];

% Defining the joint screw-axes
screw_axes = cell(12,1);
screw_axes(1) = {[1; 0; 0; 0; 0; 0]};
screw_axes(2) = {[1; 0; 0; 0; 2; 0]};
screw_axes(3) = {[1; 0; 0; 0; 5; 0]};
screw_axes(4) = {[0; 0; 1; 0; 0; 0]};
screw_axes(5) = {[1; 0; 0; 0; 6; -2]};
screw_axes(6) = {[0; 0; 0; 0; 1; 0]};
screw_axes(7) = {[0; 0; 1; -3; 0; 0]};
screw_axes(8) = {[0; 0; 1; 3; 6; 0]};
screw_axes(9) = {[0; 0; 0; 0; 6; 0]};
tree.Nodes.screwAxes = screw_axes;

% Initial Configurations
M1 = [[1,0,0;0,0,-1;0,1,0], [0;4;6]; zeros(1, 3), 1];
M2 = [[1,0,0;0,-1,0;0,0,-1], [3;-3;5]; zeros(1, 3), 1];
M3 = [[1,0,0;0,0,1;0,-1,0], [3;-10;6]; zeros(1, 3), 1];

% Tree visualization for clarity
plot(tree);

%% Initial values and parameter definitions
```

```
% The end effector that is considered
end_effector = 'E3';

% Target joint displacements
theta_goal1 = [10, 20, 30, 5, 45, 0, 0, 0, 0];
theta_goal2 = [10, 20, 30, 5, 0, 65, 12, 26, 0];
theta_goal3 = [25, 60, 38, 10, 0, 34, 11, 2, 100];

% Initial guess at joint displacements
theta_init1 = [9, 18, 27, 4, 46, 0, 0, 0, 0];
theta_init2 = [9, 18, 27, 4, 0, 69, 10, 23, 0];
theta_init3 = [20, 55, 40, 12, 0, 30, 9, 4, 94];

% Goal, initial configuration, and initial guess to be
chosen
if string(end_effector) == "E1"
    M = M1;
    theta_goal = theta_goal1;
    theta_init = theta_init1;
elseif string(end_effector) == "E2"
    M = M2;
    theta_goal = theta_goal2;
    theta_init = theta_init2;
else
    M = M3;
    theta_goal = theta_goal3;
    theta_init = theta_init3;
end

% Desired transformation matrix
Tsd = trans_mat_manipulator(tree, theta_goal,
```

```
end_effector)*M;

%% Algorithm to iteratively find the solution of the
required IK problem

N = 1000;
damper = 0.01;
theta_curr = theta_init;
theta = [transpose(theta_curr),
zeros(length(theta_curr), N)];
for i = 1:N
    % To get the current configuration w.r.t space
frame {s}
    curr_trans_mat = trans_mat_manipulator(tree,
theta_curr, end_effector)*M;

    % To define the current jacobian matrix that comes
into picture during
    % motion
    curr_J = jacobian(tree, theta_curr, end_effector);

    % The twist required to translate from the current
configuration to the
    % desired configuration
    curr_twist =
adj_mat(curr_trans_mat)*matLog(pinv(curr_trans_mat)*Tsd
);

    % Updating the value of the solution theta
    theta_curr = theta_curr +
damper*transpose(pinv(curr_J)*curr_twist);
```

```
    % Storing the history of the plausible solutions of
theta
    theta(:,i+1) = transpose(theta_curr);
end

%% Plotting the results

% Iterator
itr =0:N;

figure;
subplot(3, 1, 1);
plot(itr, theta(1,:));
title('IK solution for joint-1');
xlabel('No of iterations');
ylabel('\theta_1(t)');
subplot(3, 1, 2);
plot(itr, theta(2,:));
title('IK solution for joint-2');
xlabel('No of iterations');
ylabel('\theta_2(t)');
subplot(3, 1, 3);
plot(itr, theta(3,:));
title('IK solution for joint-3');
xlabel('No of iterations');
ylabel('\theta_3(t)');

figure;
subplot(3, 1, 1);
plot(itr, theta(4,:));
```

```
title('IK solution for joint-4');
xlabel('No of iterations');
ylabel('\theta_4(t)');
subplot(3, 1, 2);
plot(itr, theta(5,:));
title('IK solution for joint-5');
xlabel('No of iterations');
ylabel('\theta_5(t)');
subplot(3, 1, 3);
plot(itr, theta(6,:));
title('IK solution for joint-6');
xlabel('No of iterations');
ylabel('\theta_6(t)');

figure;
subplot(3, 1, 1);
plot(itr, theta(7,:));
title('IK solution for joint-7');
xlabel('No of iterations');
ylabel('\theta_7(t)');
subplot(3, 1, 2);
plot(itr, theta(8,:));
title('IK solution for joint-8');
xlabel('No of iterations');
ylabel('\theta_8(t)');
subplot(3, 1, 3);
plot(itr, theta(9,:));
title('IK solution for joint-9');
xlabel('No of iterations');
ylabel('\theta_9(t)');
```