

**HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY**

**AND EDUCATION**

**FALCUTY OF MECHANICAL ENGINEERING**

**DEPARTMENT OF MECHATRONICS**

**Optimization Algorithms for Inverse Kinematics of**

**Robot** **Fanuc m2000iA 900L**

|  |  |  |
| --- | --- | --- |
| COURSE CODE: |  |  |
| Teacher: | Do Van Hien |  |
| Member: | Huynh Thanh Phong | 22134009 |
|  | Pham Ngoc Phuc | 22314010 |
|  | Le Thi Hong Phuong | 22134011 |
|  | Vo Hong Quan | 22134012 |

# INTRODUCTION

In a modern life where technological advancements drive progress across industrial, military, and security sectors, robots play a crucial role. Thus, studies on the forward and inverse kinematics of industrial robots (IRs) have been developed in overall the robotics industry. While forward kinematics is relatively straightforward to analyze, in opposite, inverse kinematics poses a significant challenge due to mathematical complexity depending on structure of robots.

Numerous papers have presented kinematic models using the general Denavit-Hartenberg (D-H) conventions (or modified Denavit-Hartenberg) to get homogeneous transformation matrices. But many models fail to incorporate the full set of constructive and functional parameters that are essential for accurately modeling a specific industrial robot, validation methods are often missing, there for making it quite difficult to verify the accuracy of mathematical models.

To address these issues, in this report, we use the Dynamic Differential Annealed Optimization (DDAO) algorithm to minimize the objective function and tackle the complexities associated with inverse kinematics. These results can also support solving dynamics, path planning, and control problems for real-scale industrial robots

# OVERVIEW

## Robot Fanuc m2000iA 900L

### Introduction

The Industrial robots are playing an increasingly important role in modern manufacturing processes, especially in industries that demand high levels of automation and absolute precision. They help improve efficiency, minimize errors, and reduce labor costs in complex processes such as assembly, machining, and material handling. With continuous advancements in technology, industrial robots are becoming more powerful and flexible, meeting the diverse needs of various industries.

The Fanuc M-2000iA/900L is a heavy-payload industrial robot designed for applications requiring substantial lifting capacity and high precision. Utilizing closed-chain kinematics, it operates efficiently in complex tasks within manufacturing environments. With an extended reach of up to 4,638 mm and a broader working envelope compared to other models, this long-arm robot is ideal for heavy-duty operations such as palletizing, assembly, and material handling in large-scale industries. Although it sacrifices some payload capacity to achieve its longer reach, the M-2000iA/900L can still lift up to 900 kg, making it suitable for material handling in the automotive and heavy machinery sectors. Equipped with the R30iA controller, this robot enhances production efficiency and throughput, addressing the evolving demands of modern manufacturing and helping businesses increase their competitiveness globally.

### Technical specifications of Fanuc m2000iA 900L

1. Mechanical structure

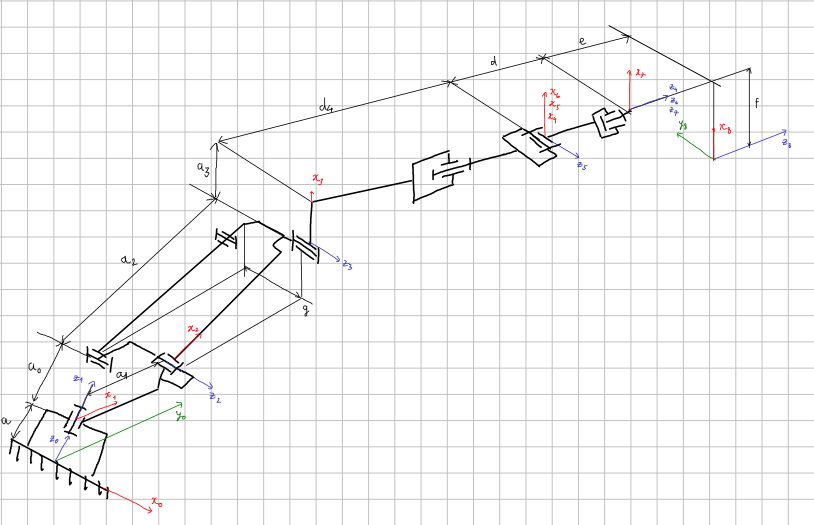
* Manipulation weight (kg): 9600
* Handling capacity (kg): 900
* Reach (m): 4638
* Robot axes: 6

1. Robot motion

|  |  |  |
| --- | --- | --- |
| **Axis** | **Type of motion** | **Range of movement - IRB 6700** |
| Axis1 | Rotation motion | ±165° |
| Axis 2 | Rotation motion | 100°/-60° |
| Axis 3 | Rotation motion | 35°/-130° |
| Axis 4 | Rotation motion | ±360° |
| Axis 5 | Rotation motion | ±120° |
| Axis 6 | Rotation motion | ±360° |

# FORWARD KINEMATIC

## Modified D-H table



(Unit: mm)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Link** | **(mm)** | **()** | **(mm)** | **(rad)** |
| 1 | 0 | 0 | 0 | θ1 |
| 2 | 500 | -90 | 0 | θ2 |
| 3 | 1700 | 0 | 0 | θ3 |
| 4 | 180 | -90 | 2850 | θ4 |
| 5 | 0 | 90 | 0 | θ5 |
| 6 | 0 | -90 | 0 | θ6 |

## Components of rotation matrix

* Transformation matrix between robot links:





Where:

* From modified DH table’s values, we can deduce that:

Link 1: 



Link 2: 



Link 3: 



Link 4: 



Link 5: 



Link 6: 



* From these transformation matrices, we can conclude that the transformation matrix to convert position from the end working point to the global coordinate origin is:



























Note that: , , ,  is denote , , , respectively for minimalist purpose.

# INVERSE KINEMATIC

## Dynamic differential optimization algorithm (DDAO)

The DDAO algorithm is inspired by the dual-phase steel production process, designed for optimization tasks. It generates new solutions iteratively using a cooling schedule and probabilistic acceptance criteria.

### The mathematical framework:

* Generating a new solution:





where:

: the new solution proposed for the iteration number k, k = 1…n

, : randomly chosen solution from the population with random (i) and (j) indices

: a randomly generated solution

*rem:* remonder after division on 2

* Probability of Acceptance:





where:

*P:* Probability of accepting a solution, 

: the difference between the objective value of the proposed solution from equation (1) and the objective value of solution  (L: 1,…population size)

*T*: temperature variable

: cost of new solution

: cost of the current solution

### Structured flow for the Dynamic Differential Annealed Optimization (DDAO) algorithm:

* Initialization of parameters:

Set up the initial population  for i= 1, 2…n

Initialize the parameter temperature T and cooling rate

Calculate the cost of each solution in the population

Identify the best solution  based on the cost: = The best solution

* Main Iteration Loop (until maximum iterations are reached):

While the current iteration t is less than the maximum allowed (While t < Max iteration):

Sub-population Evaluation:

*Initialize and evaluate a sub-population *

*Calculate the cost for each member of the sub-population*

*Sort the sub-population based on the cost.*

*Set the best solution in the sub-population as (= Best solution in sub-population)*

Generate a New Solution:

*Select two random solutions  and  from the original population.*

*Calculate  from equation (1)*

*Sort population *

*If  improves the solution, update to *

*If no improvement is found, replace the worst solution in  using the acceptance criteria based on *

Update the Best Solution:

*Update  if a better solution is found.*

Cooling Schedule:

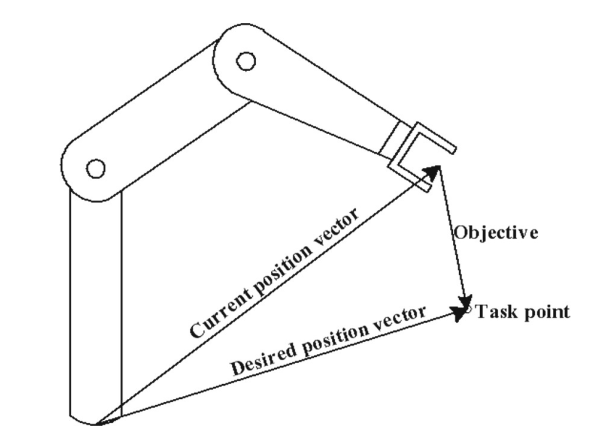
*Update the temperature  by multiplying with the cooling rate*

().

*Increment the iteration count t ()*

* Termination: Return the best solution of 

### Objective function



a) Define of objective function

- The objective function for the inverse kinematics problem aims to determine the joint variables which give the result in a given position and orientation of a robotic manipulator's end-effector. By measuring the Euclidean distance between the current position vector and the desired position vector, algorithm can minimize this function to bring the end-effector closer to the target position.

- Formular of objective function:

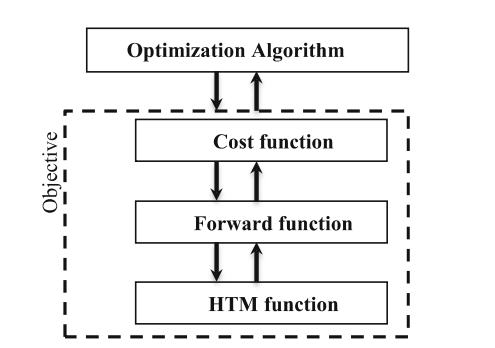
Distance between the current position of the end-effector ( ) and the desired position ():





Where ( ) are the coordinates of the desired task point

b) The procedure of the objective function



- Optimization algorithm sent the candidate solution, which is a set of possible joint variables, to the cost function to evaluate its fitness.

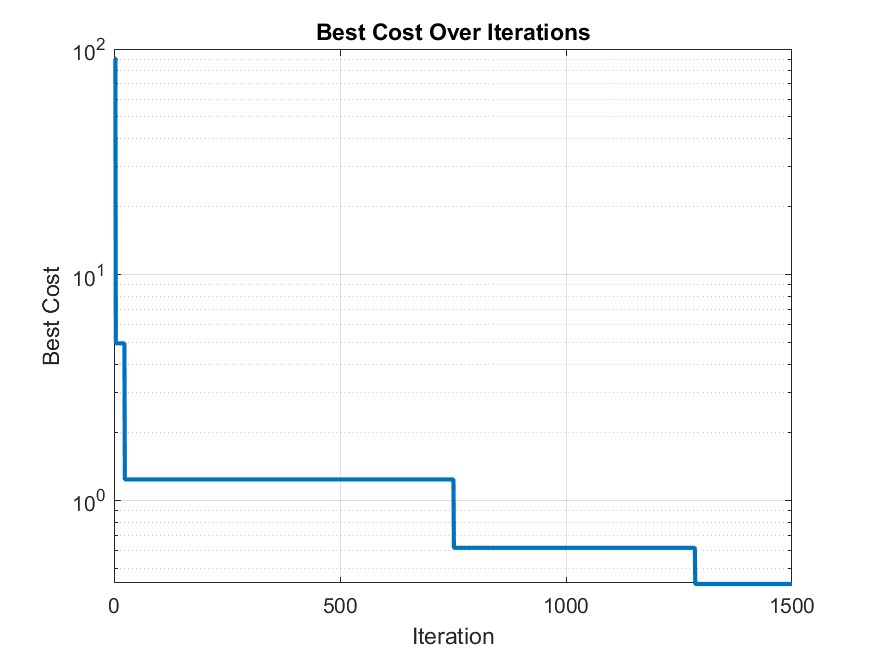
- Cost function contains the desire task point coordinates; it sends the possible solution to Forward function to get x, y, and z coordinates of the tooltip.

- Forward function contains all the forward kinematic equations of the robot arm, by substituting the candidate solution to that equations we can get the overall homogeneous transformation matrix by a repeated call for Hômgenous Transform matrix function. The output of the Forward function is the position vector of the total transformation matrix.

- Cost function will receive the position vector and apply equation (15) to the candidate vector and the desired task position vector. The result is the fitness of the solution that will be back to the main optimization algorithm.

### Implementation In Matlab

1. Training results

****

1. Code to train model

*Homogenous transform matrix*

function [T]= TransformationMatrix(theta,alpha,a,d)

T1 =[[cosd(theta) -sind(theta) 0 a ]

[cosd(alpha)\*sin(theta) cosd(alpha)\*cosd(theta) -sind(alpha) -d\*sind(alpha)]

[sind(alpha)\*sin(theta) sind(alpha)\*cosd(theta) -cosd(alpha) -d\*cosd(alpha)]

[0 0 0 1 ]];

*Forward kinematic*

function [px, py, pz] = ForwardKinematic(theta1, theta2, theta3, theta4, theta5, theta6)

% Define DH parameters and transformations

T01 = TransformationMatrix(0, 0, 680, theta1);

T12 = TransformationMatrix(-90, 320, 0, theta2-90);

T23 = TransformationMatrix(0, 975, 0, theta3);

T34 = TransformationMatrix(-90, 200, 887, theta4);

T45 = TransformationMatrix(90, 0, 0, theta5);

T56 = TransformationMatrix(-90, 0, 200, theta6 + 180);

% Final transformation matrix

T06 = T01 \* T12 \* T23 \* T34 \* T45 \* T56;

px = T06(1, 4);

py = T06(2, 4);

pz = T06(3, 4); % Extract end-effector position

end

*Cost function*

function error = CostFunction(sol, EEP)

[x, y, z] = ForwardKinematic(sol(1), sol(2), sol(3), sol(4), sol(5), sol(6));

error = norm([x, y, z] - EEP); % Euclidean distance as cost

end

*Main optimization loop*

%% Problem Definition

Nvar = 6; % Number of joint angles (decision variables)

VarLength = [1 Nvar]; % Solution vector size

L\_limit = [-180, -70, -28, -300, -120, -300]; % Lower bounds (in degrees)

U\_limit = [180, 85, 110, 300, 120, 300]; % Upper bounds (in degrees)

EEP = [1407, 0, 1855]; % Desired end-effector position

%% Test forward kinematics with initial angles

[x, y, z] = ForwardKinematic(0, 0, 0, 0, 0, 0);

%% Optimization Parameters

MaxIt = 1500; % Max iterations

MaxSubIt = 1500; % Max sub-iterations

T0 = 2000; % Initial temperature

alpha = 0.95; % Cooling rate

Npop = 3; % Population size

%% Initialization

empty\_template.Phase = []; % Store joint angles

empty\_template.Cost = []; % Store cost

% Initialize population and best solution

pop = repmat(empty\_template, Npop, 1);

BestSol.Cost = inf;

% Random initialization of the population

for i = 1:Npop

% Initialize Position

for j=1:length(L\_limit)

pop(i).Phase(j) = unifrnd(L\_limit(j),U\_limit(j),1) ;

end

pop(i).Cost = CostFunction(pop(i).Phase, EEP); % Evaluate cost

if pop(i).Cost < BestSol.Cost

BestSol = pop(i); % Update best solution

end

end

BestCost = zeros(MaxIt, 1); % Store best cost per iteration

T = T0; % Initial temperature

%% Main Optimization Loop

for t = 1:MaxIt

newpop = repmat(empty\_template, MaxSubIt, 1); % Sub-iteration population

% Generate random solutions and ensure bounds

for subit = 1:MaxSubIt

for j=1:length(L\_limit)

newpop(subit).Phase(j) = unifrnd(L\_limit(j),U\_limit(j),1) ; % Random angles

end

newpop(subit).Cost = CostFunction(newpop(subit).Phase, EEP); % Evaluate cost

end

% Sort by cost and select best solution from sub-iterations

[~, SortOrder] = sort([newpop.Cost]);

bnew = newpop(SortOrder(1));

% Differential update

for i = 1:Npop

kk = randi(Npop);

bb = randi(Npop);

if mod(t, 2) == 1

Mnew.Phase = (pop(kk).Phase - pop(bb).Phase) + bnew.Phase;

else

Mnew.Phase = (pop(kk).Phase - pop(bb).Phase) + bnew.Phase .\* rand;

end

% Ensure bounds and evaluate

Mnew.Phase = max(min(Mnew.Phase, U\_limit), L\_limit);

Mnew.Cost = CostFunction(Mnew.Phase, EEP);

% Simulated Annealing Acceptance Criterion

if Mnew.Cost < pop(i).Cost || rand <= exp(-(Mnew.Cost - pop(i).Cost) / T)

pop(i) = Mnew; % Replace with new solution

end

% Update best solution if improved

if pop(i).Cost < BestSol.Cost

BestSol = pop(i);

end

end

BestCost(t) = BestSol.Cost; % Store best cost

disp(['Iteration ' num2str(t) ': Best Cost = ' num2str(BestCost(t))]); % Display progress

T = alpha \* T; % Temperature reduction

end

%% Display Best Solution

disp('Best Solution (Joint Angles):');

disp(BestSol.Phase);

disp('Minimum Cost (Objective Function Value):');

disp(BestSol.Cost);

% Compute end-effector position for best solution

[px\_best, py\_best, pz\_best] = ForwardKinematic(BestSol.Phase(1), BestSol.Phase(2), BestSol.Phase(3), BestSol.Phase(4), BestSol.Phase(5), BestSol.Phase(6));

disp('End-effector position from Best Solution:');

disp(['x = ', num2str(px\_best), ', y = ', num2str(py\_best), ', z = ', num2str(pz\_best)]);

disp('Desired End-effector position:');

disp(['xd = ', num2str(EEP(1)), ', yd = ', num2str(EEP(2)), ', zd = ', num2str(EEP(3))]);

%% Plot Best Cost Over Iterations

figure;

semilogy(BestCost, 'LineWidth', 2);

xlabel('Iteration');

ylabel('Best Cost');

grid on;

title('Best Cost Over Iterations');

% Initialize population and best solution

pop = repmat(empty\_template, Npop, 1);

BestSol.Cost = inf;

% Random initialization of the population

for i = 1:Npop

% Initialize Position

for j=1:length(L\_limit)

pop(i).Phase(j) = unifrnd(L\_limit(j),U\_limit(j),1) ;

end

pop(i).Cost = CostFunction(pop(i).Phase, EEP); % Evaluate cost

if pop(i).Cost < BestSol.Cost

BestSol = pop(i); % Update best solution

end

end

BestCost = zeros(MaxIt, 1); % Store best cost per iteration

T = T0; % Initial temperature

%% Main Optimization Loop

for t = 1:MaxIt

newpop = repmat(empty\_template, MaxSubIt, 1); % Sub-iteration population

% Generate random solutions and ensure bounds

for subit = 1:MaxSubIt

for j=1:length(L\_limit)

newpop(subit).Phase(j) = unifrnd(L\_limit(j),U\_limit(j),1) ; % Random angles

end

newpop(subit).Cost = CostFunction(newpop(subit).Phase, EEP); % Evaluate cost

end

% Sort by cost and select best solution from sub-iterations

[~, SortOrder] = sort([newpop.Cost]);

bnew = newpop(SortOrder(1));

% Differential update

for i = 1:Npop

kk = randi(Npop);

bb = randi(Npop);

if mod(t, 2) == 1

Mnew.Phase = (pop(kk).Phase - pop(bb).Phase) + bnew.Phase;

else

Mnew.Phase = (pop(kk).Phase - pop(bb).Phase) + bnew.Phase .\* rand;

end

% Ensure bounds and evaluate

Mnew.Phase = max(min(Mnew.Phase, U\_limit), L\_limit);

Mnew.Cost = CostFunction(Mnew.Phase, EEP);

% Simulated Annealing Acceptance Criterion

if Mnew.Cost < pop(i).Cost || rand <= exp(-(Mnew.Cost - pop(i).Cost) / T)

pop(i) = Mnew; % Replace with new solution

end

% Update best solution if improved

if pop(i).Cost < BestSol.Cost

BestSol = pop(i);

end

end

BestCost(t) = BestSol.Cost; % Store best cost

disp(['Iteration ' num2str(t) ': Best Cost = ' num2str(BestCost(t))]); % Display progress

T = alpha \* T; % Temperature reduction

end

%% Display Best Solution

disp('Best Solution (Joint Angles):');

disp(BestSol.Phase);

disp('Minimum Cost (Objective Function Value):');

disp(BestSol.Cost);

% Compute end-effector position for best solution

[px\_best, py\_best, pz\_best] = ForwardKinematic(BestSol.Phase(1), BestSol.Phase(2), BestSol.Phase(3), BestSol.Phase(4), BestSol.Phase(5), BestSol.Phase(6));

disp('End-effector position from Best Solution:');

disp(['x = ', num2str(px\_best), ', y = ', num2str(py\_best), ', z = ', num2str(pz\_best)]);

disp('Desired End-effector position:');

disp(['xd = ', num2str(EEP(1)), ', yd = ', num2str(EEP(2)), ', zd = ', num2str(EEP(3))]);

%% Plot Best Cost Over Iterations

figure;

semilogy(BestCost, 'LineWidth', 2);

xlabel('Iteration');

ylabel('Best Cost');

grid on;

title('Best Cost Over Iterations');

BestCost = zeros(MaxIt, 1); % Store best cost per iteration

T = T0; % Initial temperature

%% Main Optimization Loop

for t = 1:MaxIt

newpop = repmat(empty\_template, MaxSubIt, 1); % Sub-iteration population

% Generate random solutions and ensure bounds

for subit = 1:MaxSubIt

for j=1:length(L\_limit)

newpop(subit).Phase(j) = unifrnd(L\_limit(j),U\_limit(j),1) ; % Random angles

end

newpop(subit).Cost = CostFunction(newpop(subit).Phase, EEP); % Evaluate cost

end

% Sort by cost and select best solution from sub-iterations

[~, SortOrder] = sort([newpop.Cost]);

bnew = newpop(SortOrder(1));

% Differential update

for i = 1:Npop

kk = randi(Npop);

bb = randi(Npop);

if mod(t, 2) == 1

Mnew.Phase = (pop(kk).Phase - pop(bb).Phase) + bnew.Phase;

else

Mnew.Phase = (pop(kk).Phase - pop(bb).Phase) + bnew.Phase .\* rand;

end

% Ensure bounds and evaluate

Mnew.Phase = max(min(Mnew.Phase, U\_limit), L\_limit);

Mnew.Cost = CostFunction(Mnew.Phase, EEP);

% Simulated Annealing Acceptance Criterion

if Mnew.Cost < pop(i).Cost || rand <= exp(-(Mnew.Cost - pop(i).Cost) / T)

pop(i) = Mnew; % Replace with new solution

end

% Update best solution if improved

if pop(i).Cost < BestSol.Cost

BestSol = pop(i);

end

end

BestCost(t) = BestSol.Cost; % Store best cost

disp(['Iteration ' num2str(t) ': Best Cost = ' num2str(BestCost(t))]); % Display progress

T = alpha \* T; % Temperature reduction

end

%% Display Best Solution

disp('Best Solution (Joint Angles):');

disp(BestSol.Phase);

disp('Minimum Cost (Objective Function Value):');

disp(BestSol.Cost);

% Compute end-effector position for best solution

[px\_best, py\_best, pz\_best] = ForwardKinematic(BestSol.Phase(1), BestSol.Phase(2), BestSol.Phase(3), BestSol.Phase(4), BestSol.Phase(5), BestSol.Phase(6));

disp('End-effector position from Best Solution:');

disp(['x = ', num2str(px\_best), ', y = ', num2str(py\_best), ', z = ', num2str(pz\_best)]);

disp('Desired End-effector position:');

disp(['xd = ', num2str(EEP(1)), ', yd = ', num2str(EEP(2)), ', zd = ', num2str(EEP(3))]);

%% Plot Best Cost Over Iterations

figure;

semilogy(BestCost, 'LineWidth', 2);

xlabel('Iteration');

ylabel('Best Cost');

grid on;

title('Best Cost Over Iterations');

% Sort by cost and select best solution from sub-iterations

[~, SortOrder] = sort([newpop.Cost]);

bnew = newpop(SortOrder(1));

% Differential update

for i = 1:Npop

kk = randi(Npop);

bb = randi(Npop);

if mod(t, 2) == 1

Mnew.Phase = (pop(kk).Phase - pop(bb).Phase) + bnew.Phase;

else

Mnew.Phase = (pop(kk).Phase - pop(bb).Phase) + bnew.Phase .\* rand;

end

% Ensure bounds and evaluate

Mnew.Phase = max(min(Mnew.Phase, U\_limit), L\_limit);

Mnew.Cost = CostFunction(Mnew.Phase, EEP);

% Simulated Annealing Acceptance Criterion

if Mnew.Cost < pop(i).Cost || rand<= exp(-(Mnew.Cost - pop(i).Cost) /T)

pop(i) = Mnew; % Replace with new solution

end

% Update best solution if improved

if pop(i).Cost < BestSol.Cost

BestSol = pop(i);

end

end

BestCost(t) = BestSol.Cost; % Store best cost

disp(['Iteration ' num2str(t) ': Best Cost = ' num2str(BestCost(t))]); % Display progress

T = alpha \* T; % Temperature reduction

end

%% Display Best Solution

disp('Best Solution (Joint Angles):');

disp(BestSol.Phase);

disp('Minimum Cost (Objective Function Value):');

disp(BestSol.Cost);

% Compute end-effector position for best solution

[px\_best, py\_best, pz\_best] = ForwardKinematic(BestSol.Phase(1), BestSol.Phase(2), BestSol.Phase(3), BestSol.Phase(4), BestSol.Phase(5), BestSol.Phase(6));

disp('End-effector position from Best Solution:');

disp(['x = ', num2str(px\_best), ', y = ', num2str(py\_best), ', z = ', num2str(pz\_best)]);

disp('Desired End-effector position:');

disp(['xd = ', num2str(EEP(1)), ', yd = ', num2str(EEP(2)), ', zd = ', num2str(EEP(3))]);

%% Plot Best Cost Over Iterations

figure;

semilogy(BestCost, 'LineWidth', 2);

xlabel('Iteration');

ylabel('Best Cost');

grid on;

title('Best Cost Over Iterations');

%% Display Best Solution

disp('Best Solution (Joint Angles):');

disp(BestSol.Phase);

disp('Minimum Cost (Objective Function Value):');

disp(BestSol.Cost);

% Compute end-effector position for best solution

[px\_best, py\_best, pz\_best] = ForwardKinematic(BestSol.Phase(1), BestSol.Phase(2), BestSol.Phase(3), BestSol.Phase(4), BestSol.Phase(5), BestSol.Phase(6));

disp('End-effector position from Best Solution:');

disp(['x = ', num2str(px\_best), ', y = ', num2str(py\_best), ', z = ', num2str(pz\_best)]);

disp('Desired End-effector position:');

disp(['xd = ', num2str(EEP(1)), ', yd = ', num2str(EEP(2)), ', zd = ', num2str(EEP(3))]);

%% Plot Best Cost Over Iterations

figure;

semilogy(BestCost, 'LineWidth', 2);

xlabel('Iteration');

ylabel('Best Cost');

grid on;

title('Best Cost Over Iterations');

## Neural Network

### The mathematical framework: