ELECTRICAL ENGINEERING DEPARTMENT

IIT ROORKEE AUTUMN

2021-2022

EEN-351 Artificial Neural Networks

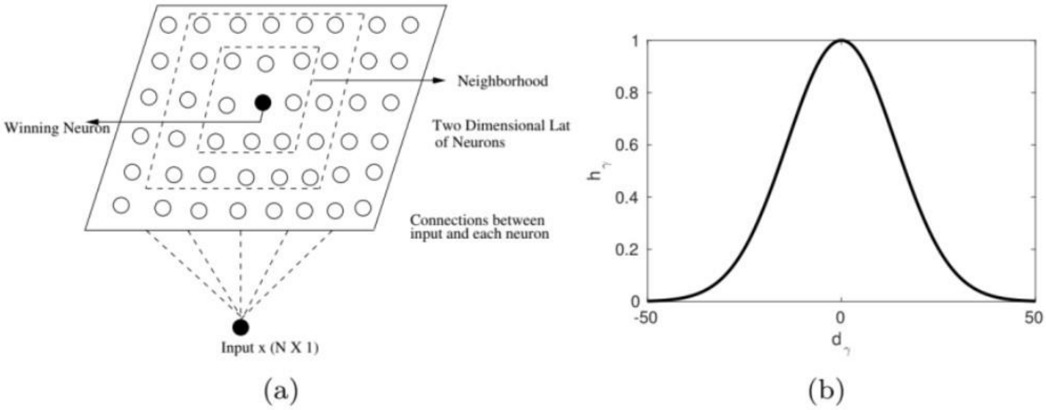
Term Project

(Forward Kinematics of a 2-link Manipulator)

Submitted by: Davinder Singh

Enrolment Number: 19115046

Kohonen proposed an unsupervised learning algorithm that can form clusters for a given data set while preserving topology. A simple configuration of Kohonen self-organizing feature maps is listed below:



The prominent feature of this network is a lattice that can be m dimensional. Although the dimension of lattice is a priori fixed, this dimension usually refers to the topology of the real-world data. Another prominent feature is the concept of excitatory learning with a neighbourhood around the winning neuron. The size of the neighbourhood slowly decrease as learning progresses as shown above. To be precise, in the initial phase, almost all neurons participate in the learning as the network is excited by an input pattern x.

The basic idea is to discover patterns in the input data in a self-organizing way while similar data are represented by a weight vector w_γ associated with the γth neuron. This clustering takes place in following three steps:

Competition: For each input pattern, the neurons in the network compute their respective values of a discriminant function. The neuron with the largest value of the function is declared the winner. This discriminant function is usually a measure of Euclidean distance.

Cooperation: The winning neuron determines the spatial location of a topological neighbourhood of excited neurons, i.e., cooperative neighbouring neurons.

Synaptic Adaptation: The executed neurons which are situated in the neighbourhood of the winning neuron adjust their synaptic weights in the relation to the input pattern.

Algorithm:

Lets we have data {x, y} we can build a neural architecture around KSOM that will learn the unknow map *f().* Let's assume that the following nonlinear map is given:

$$y = f(x), \qquad x = R^n, \qquad y \in R^m$$

We will express this nonlinear function as aggregation of linear functions using first order Taylor series expansion. Given any input vector x_0,

$$y_0 = f(x_0)$$

Using first order Tylor series expansion, the output y can be expressed linearly around $\mathbf{x_0}$ as follows:

$$y_0 = y_0 + (\partial y/\partial x \text{ at } x = x_0)(x-x0)$$

Let's consider the following Kohonen lattice where each neuron is associated with the following linear model:

$$y^\gamma = y_\gamma + A_\gamma(x - w_\gamma)$$

where given x, y^γ is the linear response of the γ^{th} neuron. This is associated with three parametes: $\mathbf{w}_\gamma$, the natural weight vector; $\mathbf{y}_\gamma$ which should converge to f($\mathbf{w}_\gamma$); $\mathbf{A}_\gamma$ which is equivalent of $(\partial y/\partial x \text{ at } x = w_\gamma)$

The linear reponse of each neuron given *x* has a weight of $\mathbf{h}_\gamma$ where $\mathbf{h}_\gamma$ is the neighbourhood function with respect to the winning neuron. Thus the nonlinear map *y = f(x)* can be approximated as:

$$y = \Sigma\, h_\gamma y^\gamma / \Sigma\, h_\gamma$$

where $\mathbf{h}_\gamma$ = **exp(-d γ2/2σ2)** and $\mathbf{d}_\gamma$ is the lattice distance between the winning neuron *I* and the γ^{th} neuron.

The final expression for network response can be given as:

$$y = \Sigma\, h\gamma(y_\gamma + A_\gamma(x - w_\gamma)) / \Sigma\, h_\gamma$$

Weight Update Algorithm:

$$w_\gamma = w_\gamma + \eta\, h_\gamma(x - w_\gamma)$$

Cost Function (Let)

$$E = ½\, \bar{y}^T\bar{y},$$

Where, $\bar{y} = y^d - y$ and y^d is desired response given x while y is network response.

The update law can be derived by using gradient descent:

$$\partial E/\partial y_\gamma = \bar{y}^T \partial y/\partial y_\gamma = -\, \bar{y}^T (h_\gamma / \Sigma\, h_\gamma)$$

Thus, the update law for $\mathbf{y}_\gamma$ becomes:

$$\mathbf{y}_\gamma \leftarrow \mathbf{y}_\gamma + \eta\,(\mathbf{h}_\gamma / \Sigma\,\mathbf{h}_\gamma)\,\bar{\mathbf{y}}$$

For the update law of A_γ the gradient term is derived as:

$$\partial \mathbf{E}/\partial \mathbf{A}_\gamma = \bar{\mathbf{y}}^T \partial \mathbf{y}/\partial \mathbf{A}_\gamma = -\,\bar{\mathbf{y}}^T\,(\mathbf{h}_\gamma / \Sigma\,\mathbf{h}_\gamma)\,(\mathbf{x} - \mathbf{w}_\gamma) = -\,(\mathbf{h}_\gamma / \Sigma\,\mathbf{h}_\gamma)\,(\mathbf{x} - \mathbf{w}_\gamma)\,\bar{\mathbf{y}}^T$$

Thus, the update law becomes:

$$\mathbf{A}_\gamma \leftarrow \mathbf{A}_\gamma + \eta\,\bar{\mathbf{y}}\,((\mathbf{x} - \mathbf{w})^T(\mathbf{h}_\gamma / \Sigma\,\mathbf{h}_\gamma)$$

KSOM based network for inverse kinematics:

The forward kinematics of a 2-link manipulator is given as:

$$x = l_1 \cos\theta_1 + l_2 \cos(\theta_1 + \theta_2)$$
$$y = l_1 \sin\theta_1 + l_2 \sin(\theta_1 + \theta_2)$$

where l_1, l_2 are the respective link lengths, as shown in Fig. below. Given that $l_1 = l_2 = 1m$.

Design a KSOM based network that can model inverse kinematics, i.e. given Cartesian position $(\boldsymbol{x}, \boldsymbol{y})$, make prediction in joint space $(\boldsymbol{\theta_1}, \boldsymbol{\theta_2})$. Take a 2-D lattice of size 10 x 10. Heuristically tune learning rate and variance for distance measure. You are free to increase the lattice size as well.

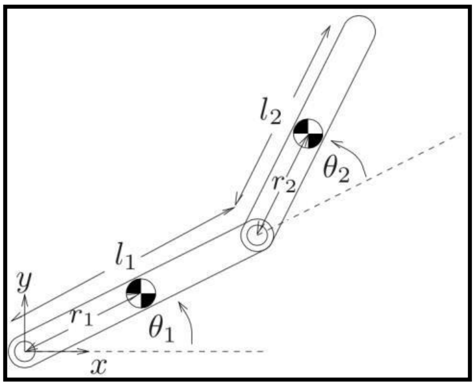


Fig. 1. 2-link Manipulator

- After training is over, the network will predict joint angles for reaching a desired Cartesian position. Given this predicted joint position, compute the actual Cartesian position that the manipulator end-effector has reached using the forward kinematics.

- Fill in the following table to evaluate the performance of your model:

Desired position (x_d, y_d)	Desired position ($\boldsymbol{\theta_1}$, $\boldsymbol{\theta_2}$)	Actual end effector position ($\boldsymbol{x}$, $\boldsymbol{y}$)
(0, 1.414)m		
(1.414, 0)m		
(1, 1)m		
(−1, −1)m		
(0.2, 0.8)m		

- Track a circle of 1.5 m radius around the manipulator base. Plot the result while showing kinematic configuration.

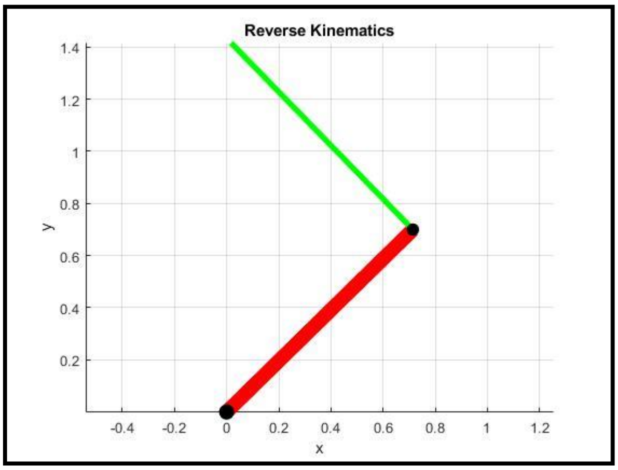
- Repeat the above step to track a straight-line in its workspace.

Matlab Code (Exact Solution of Inverse Kinematic for 2d planer manipulator) :

```
clc; clear; close all;

% initlizations
l1 = 1; l2 = 1;

% parametric equations of a circle
% X = Xcenter + rCos(phi);
% Y = Ycenter + rSin(phi);
r = 1.5;
x_ref = 0.0167; y_ref = 1.4161;
param = [ l1, l2, x_ref, y_ref];
thetas = [0.1, 0.1];

% reverse solutions
thetas = fsolve(@reversesol, thetas,optimoptions('fsolve','Display','iter'),
param);
theta1 = thetas(1); theta2 = thetas(2);
thetas*180/pi
% plots
x_o0 = 0; y_o0 = 0;
x_P = l1*cos(theta1); y_P = l1*sin(theta1);
x_Q = l1*cos(theta1) + l2*cos(theta1 + theta2);
y_Q = l1*sin(theta1) + l2*sin(theta1 + theta2);

line([x_o0, x_P],[y_o0, y_P], 'Linewidth', 10, 'Color', 'red'); hold on;
line([x_P, x_Q],[y_P, y_Q], 'Linewidth', 4, 'Color', 'green');
plot(x_P, y_P,'ko', 'Markersize',8, 'MarkerFaceColor','k');
plot(x_o0, y_o0,'ko', 'Markersize',10, 'MarkerFaceColor','k');
axis('equal')
xlabel('x');
ylabel('y');
title('Reverse Kinematics');
grid();

% function logic
function F = reversesol(x, param)
    l1 = param(1); l2 = param(2);
    x_ref = param(3); y_ref = param(4);
    theta1 = x(1); theta2 = x(2);
    x_Q = l1*cos(theta1) + l2*cos(theta1 + theta2);
    y_Q = l1*sin(theta1) + l2*sin(theta1 + theta2);

    F = [x_Q - x_ref, y_Q - y_ref];
end
```

Example :
(x_ref = 0.0167, y_ref = 1.4161)

Output :
(thetas = [44.4048 89.8391]



Network Code (Inverse Kinematic of system using KSOM based network):

```
clc; clear; close all;

% Inilialization of Model Params
sig_i= 2.5; sig_f = 0.01;
etaw_i = 1; etaw_f = 0.05;
etaA_i = 0.9; etaA_f = 0.9;

% Link lenghts
l1 = 1; l2 = 1;

A_g = 0.1*rand(2, 2, 100);
w_g = 0.1*rand(2, 1, 100); % random small weight
th_g = 0.1*rand(2, 1, 100); % random small thetas

% 2d lattice formation of size 10x10
[lx, ly] = ind2sub([10, 10], 1:100);
lattice = [lx; ly]; iterations = 6000;

% Iterations and update
for i = 1:iterations
    % Let Initial State
    th1 = (rand - 0.5)*2*pi; th2 = (rand - 0.5)*2*pi;
    x = l1*cos(th1) + l2*cos(th2 + th1);
    y = l1*sin(th1) + l2*sin(th2 + th1);

    u = [x; y];
    for j = 1:100
        dist(j) = norm(u-w_g(:,:,j));
    end

    [~, win_val] = min(dist);

    % Winning Neuron
    win = [lx(win_val), ly(win_val)];
    sig(i) = sig_i*((sig_f/sig_i)^(i/iterations));
    eta_wg(i) = etaw_i*((etaw_f/etaw_i)^(i/iterations));
    eta_Ag(i) = etaA_i*((etaA_f/etaA_i)^(i/iterations));
    d = repmat(win', 1, 100)-lattice;
    H_g = exp( - (sum(d.^2))/(2*(sig(i)^2)));

    % Coarse action
    s = sum(H_g); s2 = 0; s3 = 0;
    for k = 1:100
        s1 = H_g(k)*(th_g(:,:,k)+A_g(:,:,k)*(u-w_g(:,:,k)));
        s2 = s2 + s1;
    end

    th_o = s2/s;
    x_o = l1*cos(th_o(1)) + l2*cos(th_o(1) + th_o(2));
    y_o = l1*sin(th_o(1)) + l2*sin(th_o(1) + th_o(2));
    v_o = [x_o; y_o];

    % Fine action
    for k = 1:100
        s4 = H_g(k)*(A_g(:,:,k)*(u-v_o));
```

```
        s3 = s3 + s4;
    end

    th_1 = th_o + s3/s;
    x_1 = l1*cos(th_1(1)) + l2*cos(th_1(1) + th_1(2));
    y_1 = l1*sin(th_1(1)) + l2*sin(th_1(1) + th_1(2));
    v_1 = [x_1; y_1];

    % Update equtions
    del_v = v_1-v_o;
    del_th = th_1-th_o;
    s5 = 0; s7 = 0;

    for k = 1:100
        s6 = H_g(k)*(th_g(:,:,k)+A_g(:,:,k)*(v_o-w_g(:,:,k)));
        s5 = s5 + s6;
    end

    for t = 1:100
        deltheta_g(:,:,t) = (H_g(t)/s)*(th_o-(s5/s));
    end

    for k = 1:100
        s8 = H_g(k)*(A_g(:,:,k)*del_v);
        s7 = s8 + s7;
    end

    for t = 1:100
        deltaA_g(:,:,t) =
(H_g(t)/(s*norm(del_v)^2))*(del_th-s7/s).*(del_v');
        w_g(:,:,t) = w_g(:,:,t) + eta_wg(i)*H_g(t)*(u-w_g(:,:,t));
        th_g(:,:,t) = th_g(:,:,t) + eta_Ag(i)*deltheta_g(:,:,t);
        A_g(:,:,t) = A_g(:,:,t) + eta_Ag(i)*deltaA_g(:,:,t);
    end
end

% Plot final Weights
figure(1); hold on;
for t = 1:100
    plot(w_g(1,1,t), w_g(2,1,t), '*');
end
```

Objective-1:

Fill in the following table to evaluate the performance of your model:

Desired position (x_d, y_d)	Desired position (θ_1, θ_2)	Actual end effector position (x, y)
(0, 1.414)m		
(1.414, 0)m		
(1, 1)m		
(−1, −1)m		
(0.2, 0.8)m		

```
% POINTS TRACKING
u1 = [ 0 1.414; 1.414 0; 1 1; -1 -1; 0.2 0.8]';

v = zeros(5, 4);
for m = 1:size(u1, 2)
    u = u1(:, m);
    for j = 1:100
        dist(j) = norm(u-w_g(:, :, j));
    end

    [~, win_val] = min(dist);
    win = [lx(win_val), ly(win_val)];
    d = repmat(win', 1, 100)-lattice;
    H_g = exp(-(sum(d.^2))/(2*(sig_f^2)));

    % Corse Action
    s = sum(H_g); s2 = 0; s3 = 0;

    for k = 1:100
        s1 = H_g(k)*(th_g(:,:,k)+A_g(:,:,k)*(u-w_g(:,:,k)));
        s2 = s2 + s1;
    end

    theta = s2/s;
    x = l1*cos(theta(1)) + l2*cos(theta(1) + theta(2));
    y = l1*sin(theta(1)) + l2*sin(theta(1) + theta(2));
    % Tracked Point
    v(m, 1) = x; v(m, 2) = y;
    v(m, 3) = theta(1)*180/pi; v(m, 4) = theta(2)*180/pi;
end
array2table(v,...
    'VariableNames', {'X', 'Y', 'theta_1', 'theta_2'},...
    'RowNames', ...
    {'(0.000,1.414)','(1.414,0.000)',...
    '(1.000,1.000)','(-1.00,-1.00)','(0.200,0.800)'})
```

Input:

Desired position (x_d, y_d)	Desired position (θ_1, θ_2)	Actual end effector position (x, y)
(0, 1.414)m		
(1.414, 0)m		
(1, 1)m		
(−1, −1)m		
(0.2, 0.8)m		

Output:

ans =

5×4 table

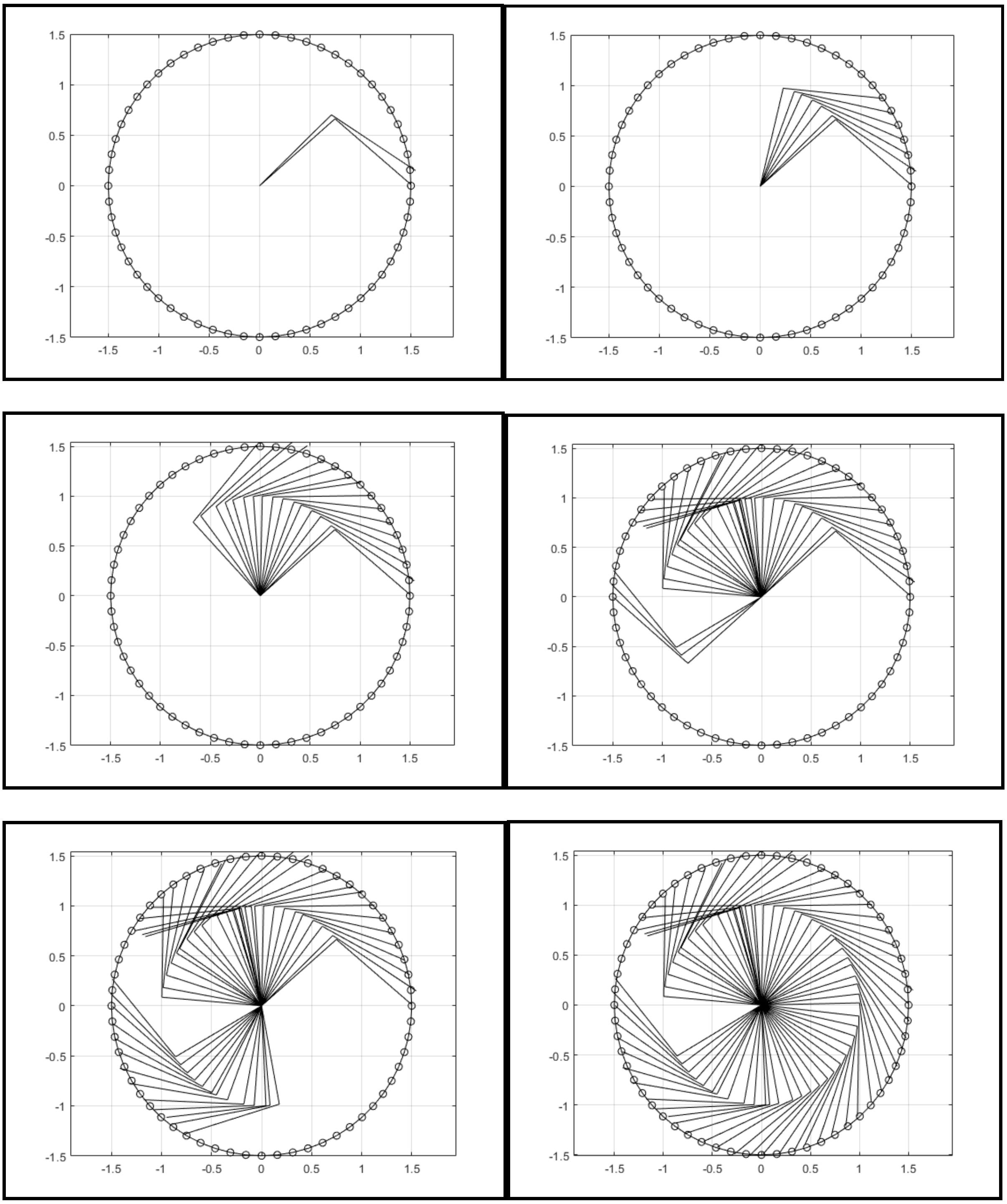
	X	Y	theta_1	theta_2
(0.000,1.414)	-0.012516	1.4166	135.41	-89.803
(1.414,0.000)	1.4044	-0.023321	44.436	-90.775
(1.000,1.000)	0.9991	0.97456	90.033	-91.491
(-1.00,-1.00)	-1.0204	-1.0288	-91.192	-87.146
(0.200,0.800)	0.22644	0.81407	139.46	-130.02

Objective-2:

Track a circle of 1.5 m radius around the manipulator base. Plot the result while showing kinematic configuration.

```
% CIRCLE TRACKING
r = 1.5;
t = 0:pi/30:2*pi; test1 = [r*cos(t); r*sin(t)];

for m = 1:length(t)
    u2 = test1(:, m);
    for j = 1:100
        dist(j) = norm(u2-w_g(:, :, j));
    end

    [~, win_val] = min(dist);
    win = [lx(win_val), ly(win_val)];
    d = repmat(win', 1, 100)-lattice;
    H_g = exp(-(sum(d.^2))/(2*(sig_f^2)));

    % Corse Action
    s = sum(H_g); s2 = 0;

    for k = 1:100
        s1 = H_g(k)*(th_g(:,:,k)+A_g(:,:,k)*(u2-w_g(:,:,k)));
        s2 = s2 + s1;
    end

    theta = s2/s;
    x_o = l1*cos(theta(1)) + l2*cos(theta(1) + theta(2));
    y_o = l1*sin(theta(1)) + l2*sin(theta(1) + theta(2));
    v_o = [x_o; y_o];
    th(:, m) = theta;
end

for i = 1:length(t)
    x_Position(i, :) = [0 l1*cos(th(1, i)) l1*cos(th(1, i)) + l2*cos(th(2, i)
+ th(1, i))];
    y_Position(i, :) = [0 l1*sin(th(1, i)) l1*sin(th(1, i)) + l2*sin(th(2, i)
+ th(1, i))];
end


figure; plot(test1(1, :), test1(2, :), '-ok'); hold on;

for i = 1:length(t)
    h = plot(x_Position(i, :), y_Position(i, :), 'k');
    axis equal
    pause(0.1);
end

axis equal
```

Output: (Snapshot of tracking at different time)



Objective-3:

Track a line around the manipulator base. Plot the result while showing kinematic configuration.

```
% LINE TRACKING

x = linspace(-1, 1, 41); y = 1.2*ones(size(x));
test2 = [x; y]; t = size(x, 2);

for m = 1:t
    u1 = test2(:, m);
    for j = 1:100
        dist(j) = norm(u1-w_g(:, :, j));
    end

    [~, win_val] = min(dist);
    win = [lx(win_val), ly(win_val)];
    d = repmat(win', 1, 100)-lattice;
    H_g = exp(-(sum(d.^2))/(2*(sig_f^2)));

    s = sum(H_g); s2 = 0;

    for k = 1:100
        s1 = H_g(k)*(th_g(:,:,k)+A_g(:,:,k)*(u1-w_g(:,:,k)));
        s2 = s2 + s1;
    end

    theta = s2/s;
    x_o = l1*cos(theta(1)) + l2*cos(theta(1) + theta(2));
    y_o = l1*sin(theta(1)) + l2*sin(theta(1) + theta(2));
    v_o = [x_o; y_o];
    th(:, m) = theta;
end

for i = 1:t
    x_Position(i, :) = [0 l1*cos(th(1, i)) l1*cos(th(1, i)) + l2*cos(th(2, i)
+ th(1, i))];
    y_Position(i, :) = [0 l1*sin(th(1, i)) l1*sin(th(1, i)) + l2*sin(th(2, i)
+ th(1, i))];
end


figure; plot(test2(1, :), test2(2, :), '-ok'); hold on;

for i = 1:t
    h = plot(x_Position(i, :), y_Position(i, :), 'k');
    pause(0.1);
end

axis equal
```

Output: (Snapshot of tracking at different time)