به نام خدا

Fundamentals of intelligent systems

Dr. aliyari

Mini project 2

Amin Elmi Ghiasi

https://github.com/AminElmiGhiasi/ML_course

Alireza Moradmand

1403

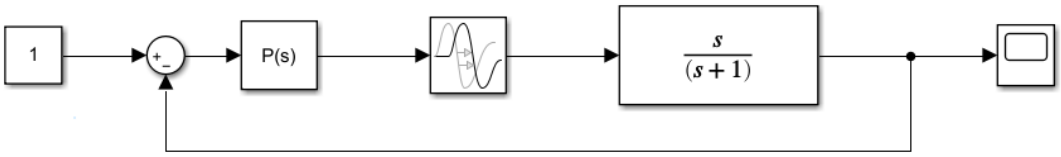
Question1:

To design a PID controller with Ziegler Nichols method for this system we should first use a P action controller and increase the gain so that system oscillates. It is done by setting I and D gains to zero and then K_p is increased so that it reaches the ultimate gain K_u, at which the output of the control loop has stable and consistent oscillations. K_u and the oscillation period T_u, then is used to set the P, I and D gains as the table below:

Ziegler–Nichols method[1]

Control Type	K_p	T_i	T_d	K_i	K_d
P	$0.5K_u$	–	–	–	–
PI	$0.45K_u$	$0.8\bar{3}T_u$	–	$0.54K_u/T_u$	–
PD	$0.8K_u$	–	$0.125T_u$	–	$0.10K_uT_u$
classic PID[2]	$0.6K_u$	$0.5T_u$	$0.125T_u$	$1.2K_u/T_u$	$0.075K_uT_u$
Pessen Integral Rule[2]	$0.7K_u$	$0.4T_u$	$0.15T_u$	$1.75K_u/T_u$	$0.105K_uT_u$
some overshoot[2]	$0.3\bar{3}K_u$	$0.50T_u$	$0.3\bar{3}T_u$	$0.6\bar{6}K_u/T_u$	$0.1\bar{1}K_uT_u$
no overshoot[2]	$0.20K_u$	$0.50T_u$	$0.3\bar{3}T_u$	$0.40K_u/T_u$	$0.06\bar{6}K_uT_u$

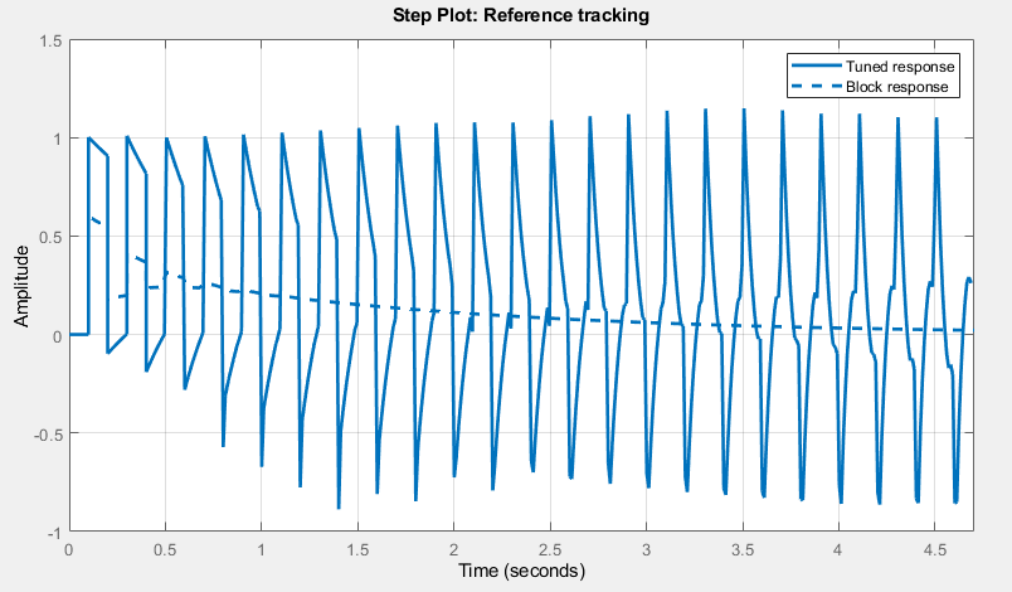
so we set up a closed loop feedback system with the plant and a delay of 0.1 and try to make systems oscillate.



We use pid tune app of matlab to find K_u and T_u.

We can take this state of the system as a consistent oscillating.

From the pid tune app now we can obtain $K_u \approx 1$ and $T_u = 0.2$ and now we can design the PID controller.



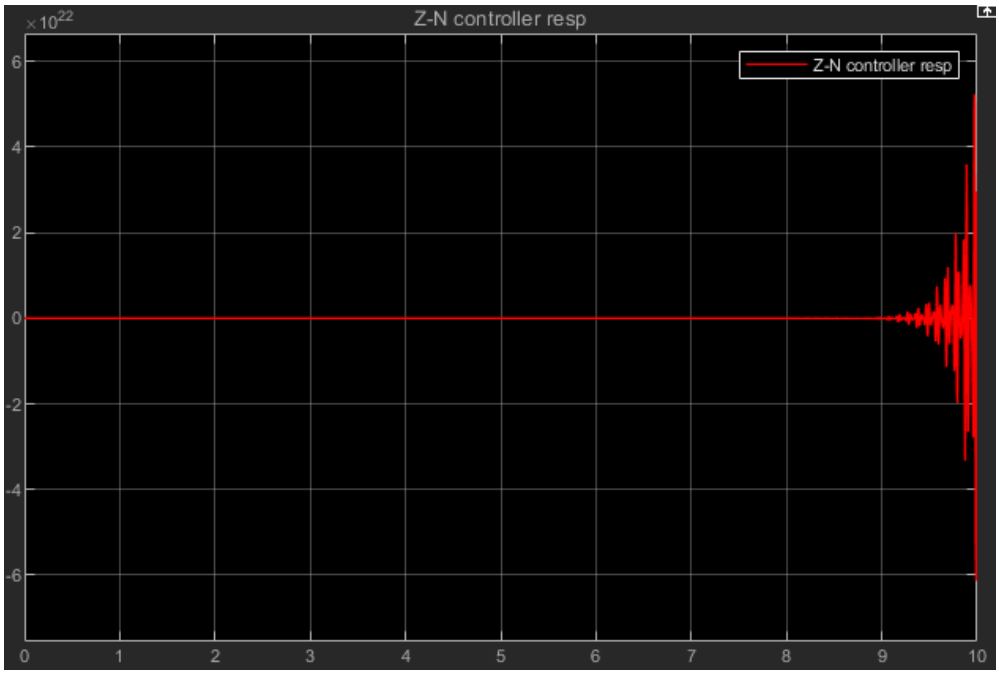
The gains of the PID are as follows:

$$K_p = 0.6K_u = 0.6$$

$$K_i = \frac{1.2K_u}{T_u} = 6$$

$$K_d = 0.075K_uT_u = 0.015$$

Now we use this controller in the feedback system and this would be the result of the step response of the system.



As it can be seen from the figure the system has become unstable after 9 s with this controller. So it is obvious that this controller needs some fine tuning to get a better result. But we stop working on this controller because our main goal is to design a controller which its gains are obtained by a Fuzzy logic inference and the gains change continuously based on the response and the gains are not constant like PID controller. To do this what we need is K_u and T_u which we have concluded before and we can continue to design our fuzzy system.

Fuzzy gain scheduling of PID controllers:

Figure below shows a PID control system with a fuzzy gain scheduler

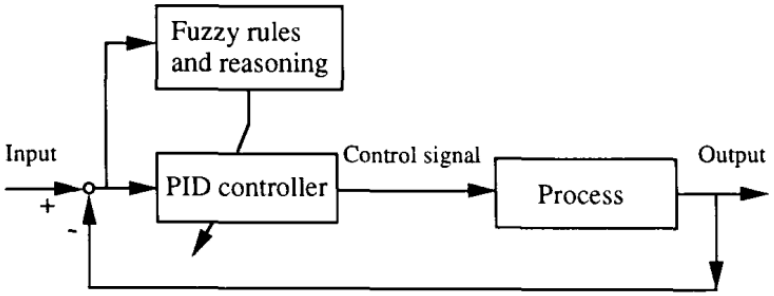


Fig. 1. PID control system with a fuzzy gain scheduler.

First we should normalize the K_p and K_d to the range between 0 and 1 by the following transformations:

$$K_p' = \frac{K_p - K_{pmin}}{K_{pmax} - K_{pmin}}$$

$$K_d' = \frac{K_d - K_{dmin}}{K_{dmax} - K_{dmin}}$$

Which the maximum and minimum of the gains are calculated like this:

$$K_{pmin} = 0.32K_u \qquad K_{pmax} = 0.6K_u$$

$$K_{dmin} = 0.08K_uT_u \qquad K_{dmax} = 0.15K_uT_u$$

Assume that the integral time constant is determined with reference to the derivative time constant by:

$$T_i = \alpha T_d$$

From which we can obtain:

$$K_i = \frac{K_p}{\alpha T_d} = \frac{K_p^2}{\alpha K_d}$$

So the parameters to be obtained by the fuzzy system are K_p' and K_d' and α then the PID gains can be obtained by the equations mentioned above.

Assume that the inputs to the fuzzy system are $e(t)$ and $\dot{e}(t) \approx \Delta e(t)$, so the fuzzy system turner consists of three two-input-one-output fuzzy systems, as shown in the figure below:

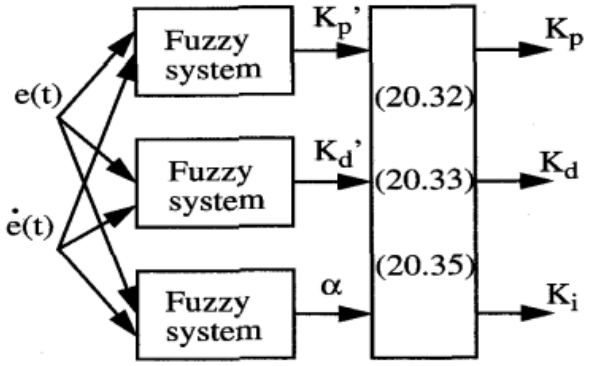
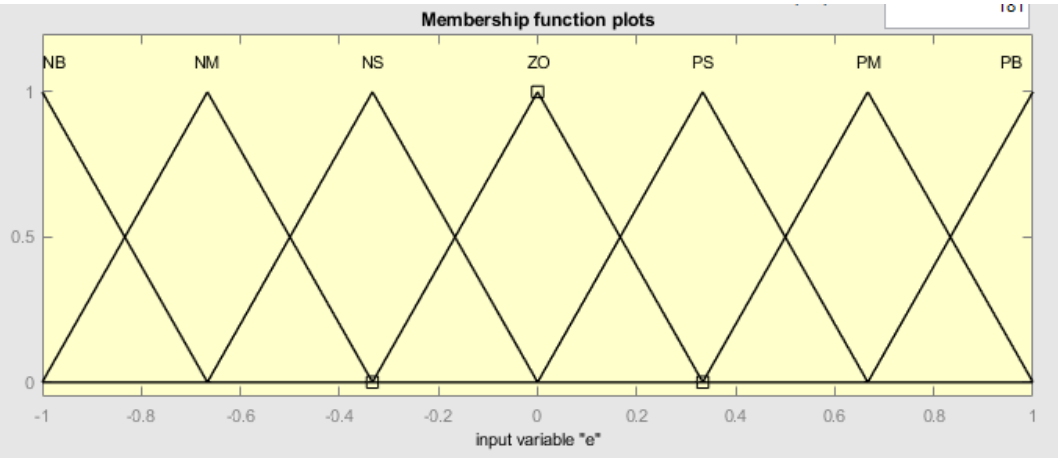
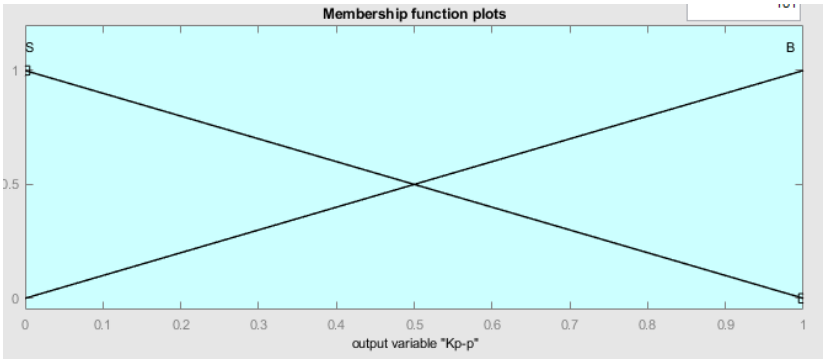


Figure 20.6. Fuzzy system turner for the PID gains.

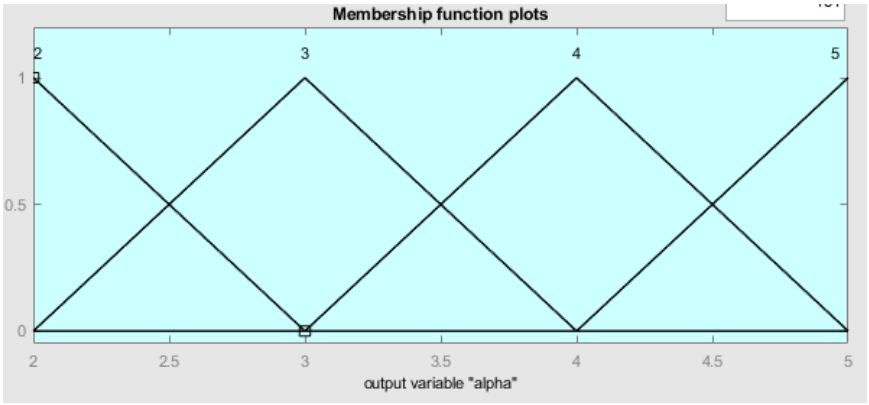
We define 7 fuzzy sets with type triangular for $e(t)$ and $\dot{e}(t)$ as shown in the picture below:



For both K_p' and K_d' we have two membership functions. For each parameter we have a membership function Big and another membership function Small with triangular type which is like this:



And for the parameter α we define 4 triangular membership functions. The membership functions are defined considering the fact that the term α must be in the range $2 \leq \alpha \leq 5$ and therefore this would be arrangement of the MFs:

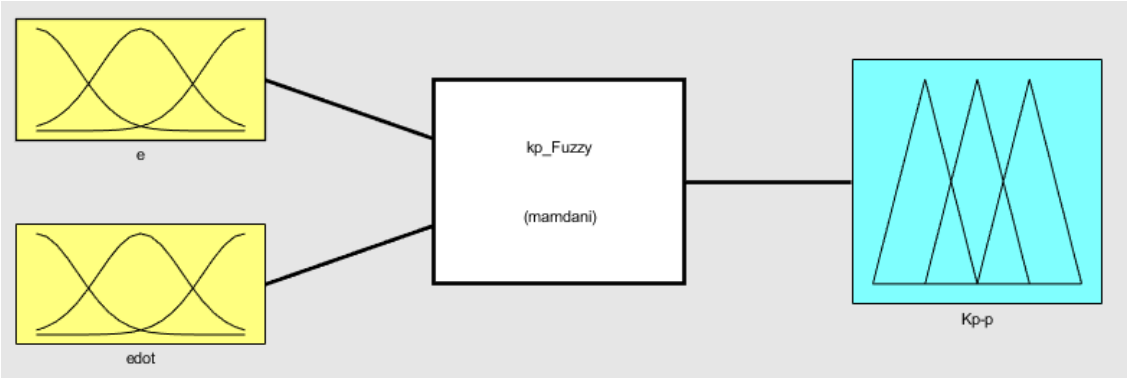


Now that we have membership functions we can drive rules experimentally based on the step response of the system and the tables below are the rules drawn in this way. For K_p':

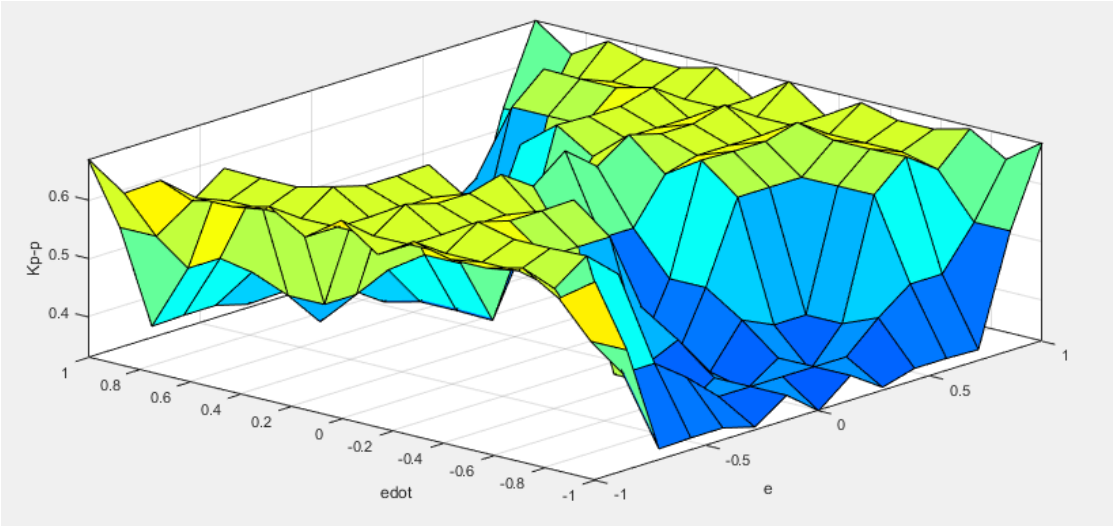
		$\dot{e}(t)$						
		NB	NM	NS	ZO	PS	PM	PB
e(t)	NB	B	B	B	B	B	B	B
	NM	S	B	B	B	B	B	S
	NS	S	S	B	B	B	S	S
	ZO	S	S	S	B	S	S	S
	PS	S	S	B	B	B	S	S
	PM	S	B	B	B	B	B	S
	PB	B	B	B	B	B	B	B

Figure 20.11. Fuzzy turning rules for K_p'.

Now based on these rules we design a fuzzy inference system for obtaining the proper K_p' from $e(t)$ and $\dot{e}(t)$ like this:



The surface derived by these rules for the FIS system designed for K_p' is like this:



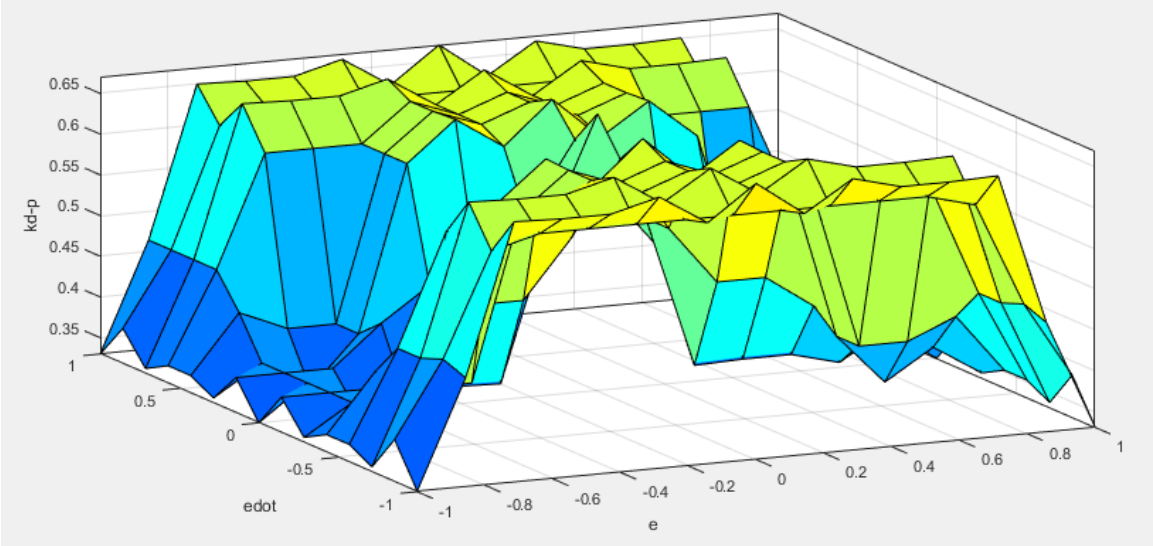
The rules for K_d' also would be like this:

		$\dot{e}(t)$						
		NB	NM	NS	ZO	PS	PM	PB
e(t)	NB	S	S	S	S	S	S	S
	NM	B	B	S	S	S	B	B
	NS	B	B	B	S	B	B	B
	ZO	B	B	B	B	B	B	B
	PS	B	B	B	S	B	B	B
	PM	B	B	S	S	S	B	B
	PB	S	S	S	S	S	S	S

Figure 20.12. Fuzzy turning rules for K_d'.

Based on these rules we design a fuzzy inference system for obtaining the proper K_d' from $e(t)$ and $\dot{e}(t)$ just like the one we designed for K_p' but with different rules.

The surface derived by these rules for the FIS system designed for K_d' would be like this:



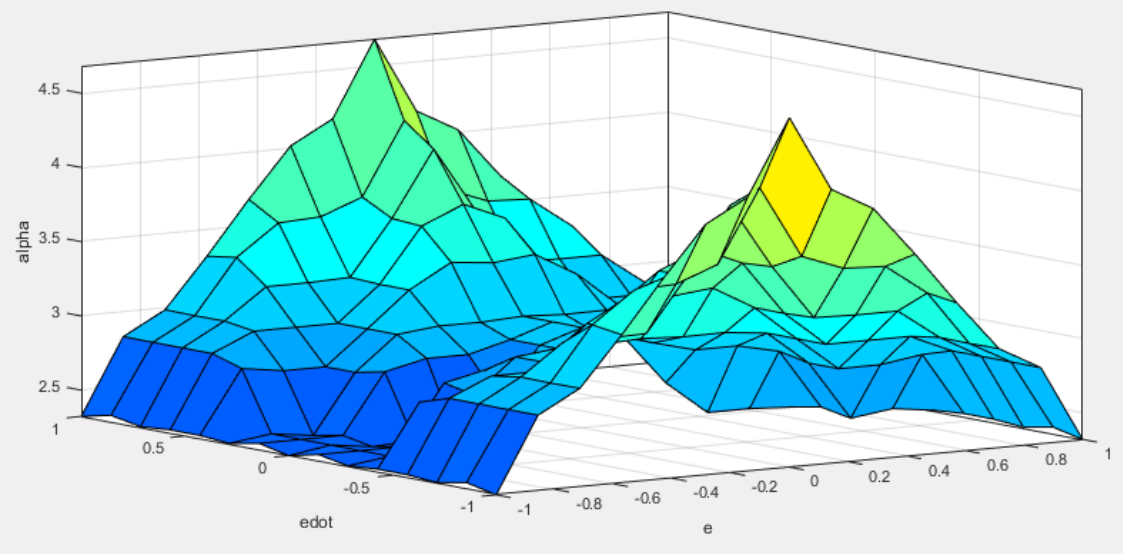
And the table of rules for α is as follows:

		$\dot{e}(t)$						
		NB	NM	NS	ZO	PS	PM	PB
e(t)	NB	2	2	2	2	2	2	2
	NM	3	3	2	2	2	3	3
	NS	4	3	3	2	3	3	4
	ZO	5	4	3	3	3	4	5
	PS	4	3	3	2	3	3	4
	PM	3	3	2	2	2	3	3
	PB	2	2	2	2	2	2	2

Figure 20.13. Fuzzy turning rules for α.

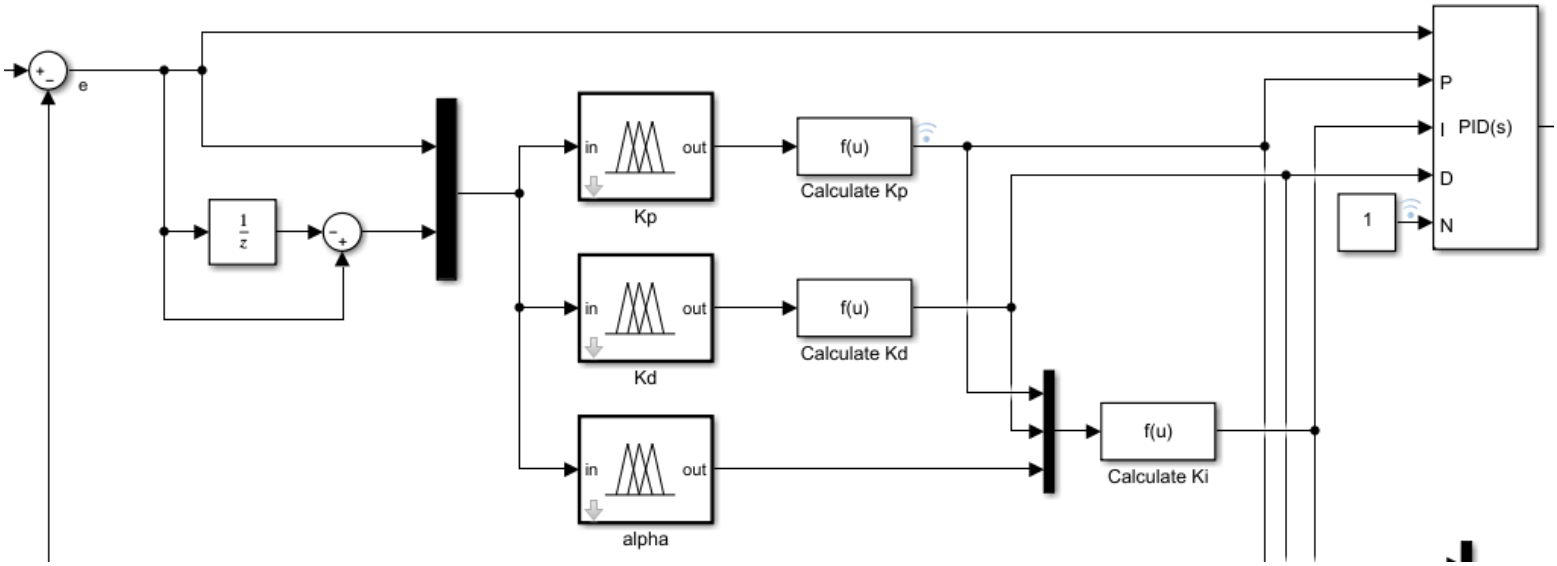
The third FIS which is designed to obtain α is like this:

The surface derived by these rules for the FIS system designed for α would be like this:



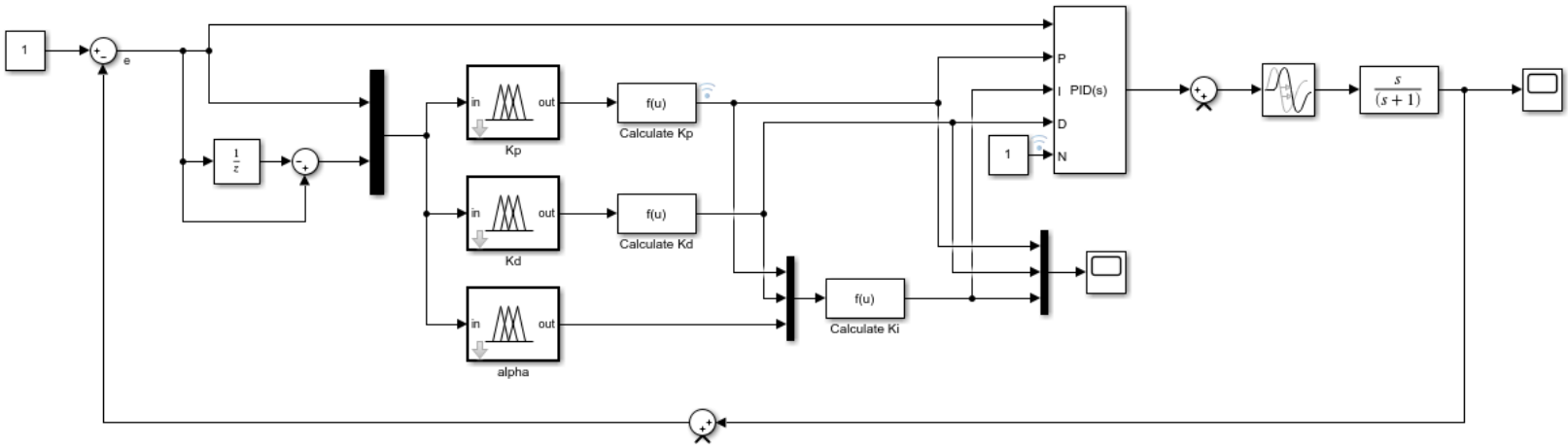
After we designed 3 FIS with Fuzzy logic GUI of the matlab we use the in the Simulink in our feedback closed loop system.

The configuration of the Fuzzy gain scheduling system in the closed loop system would be like this:

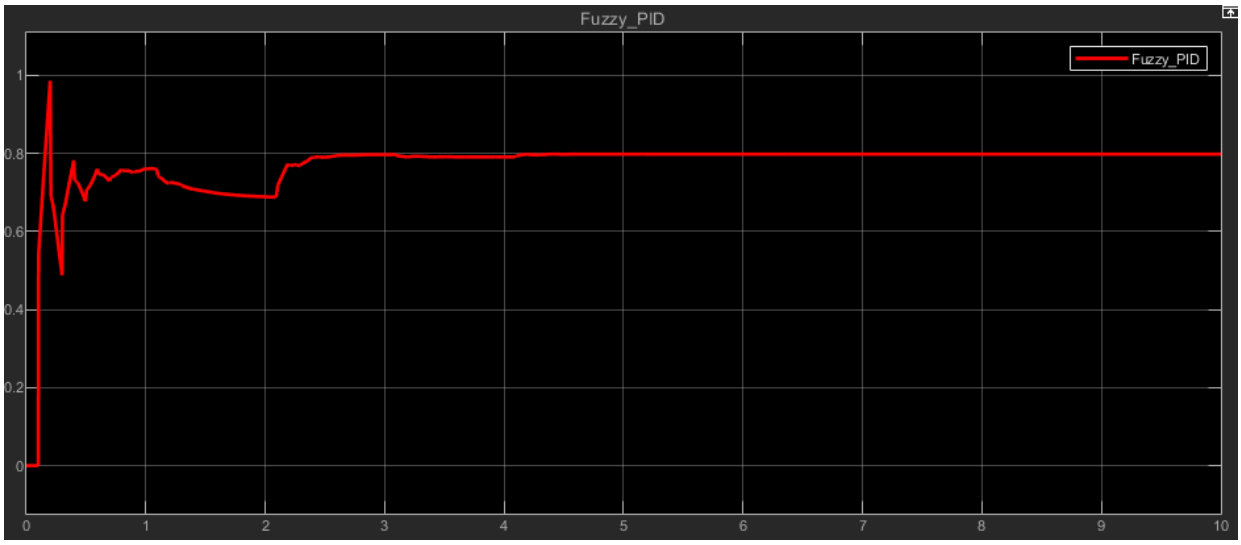


Δe is calculated by using a unit delay block and subtracting current error from previous error. Then $e(t)$ and $\Delta e(t)$ are feed into fuzzy blocks. Output of the FIS blocks related to K_p' and K_d' are deformalized to obtain K_p and K_d then K_i is then calculated. The last step is to update gains of the PID controller block externally by the fuzzy systems.

The whole closed loop system is like this;



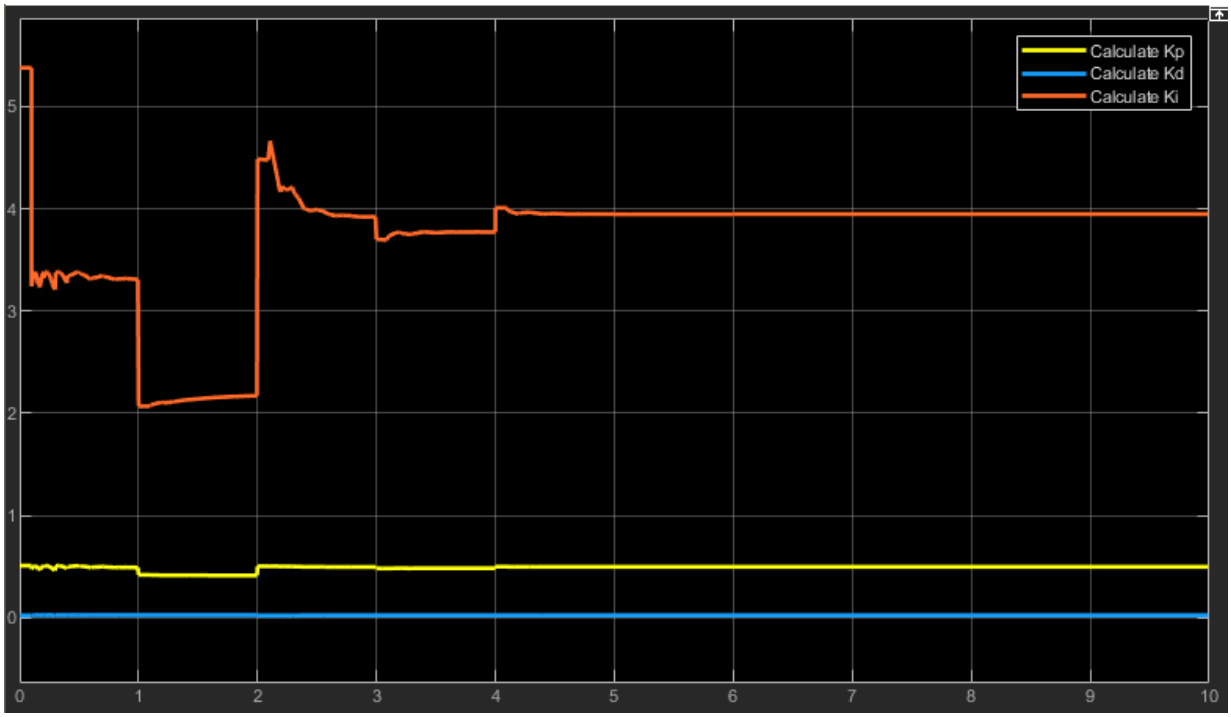
The step response of this system is as follows:



The final value of this response is 0.8 means that this controller leads to steady state error of 0.2.

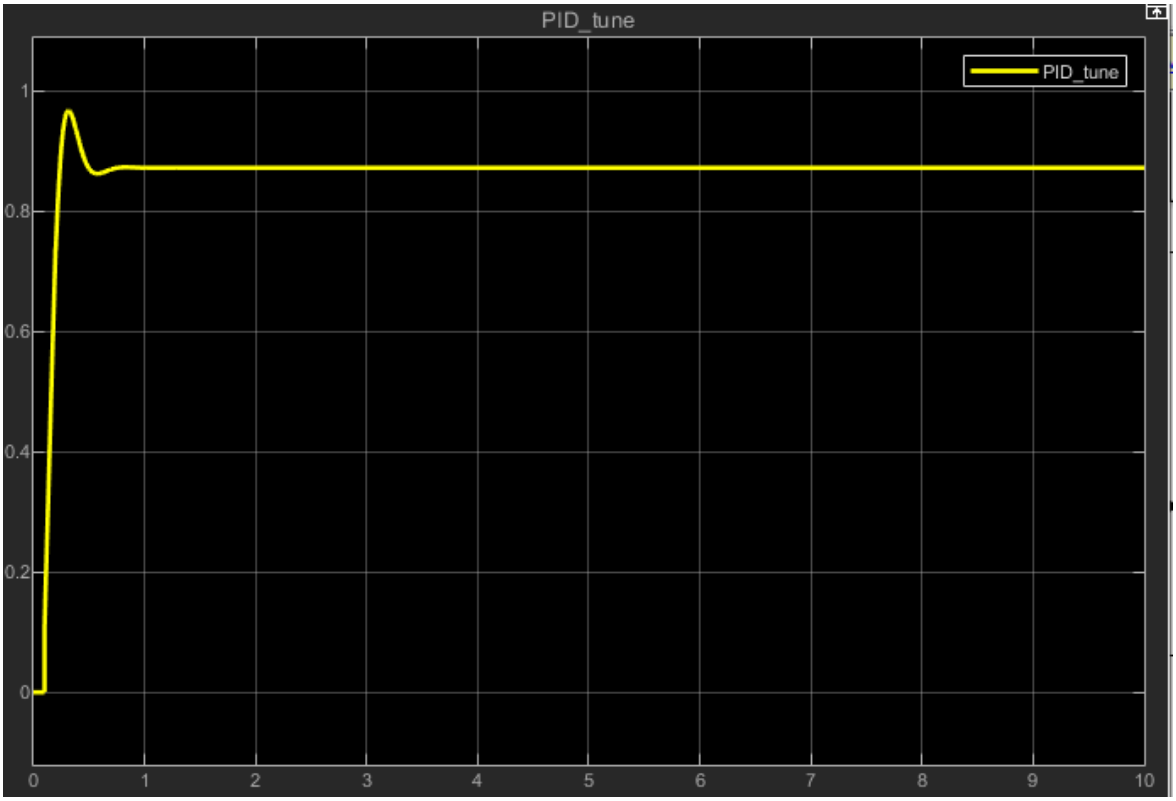
At all the performance of the controller is good. Unlike the Ziegler Nichols PID controller that caused instability the Fuzzy PID has successfully controlled the system but it has steady state error.

The figure below is the plot of gains which are output of Fuzzy blocks. As it can be seen the value of K_i is greater than the others and it has more changes over time.



So far we have designed a Fuzzy PID controller which has an acceptable performance now let's test a PID controller tuned by the matlab PID tuner app.

The step response of the tuned PID is shown in the picture below:

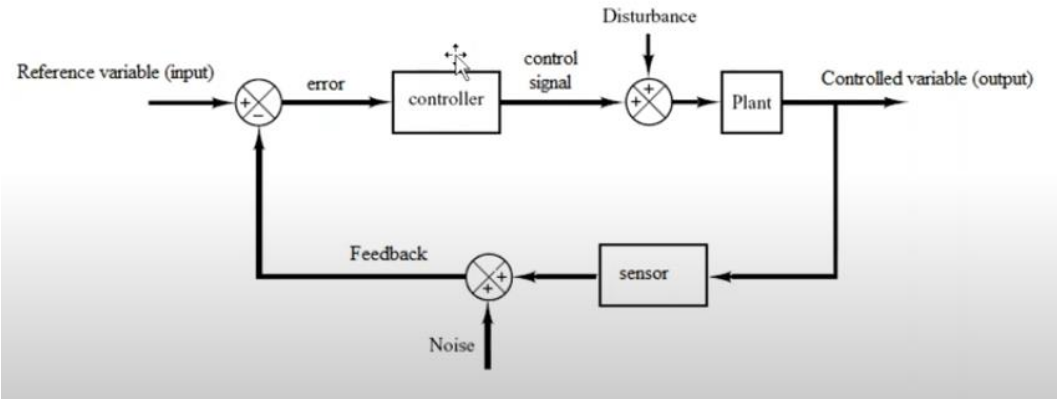


This response also has steady state error but its final value is 0.87 which is 0.07 more than final value of Fuzzy PID.

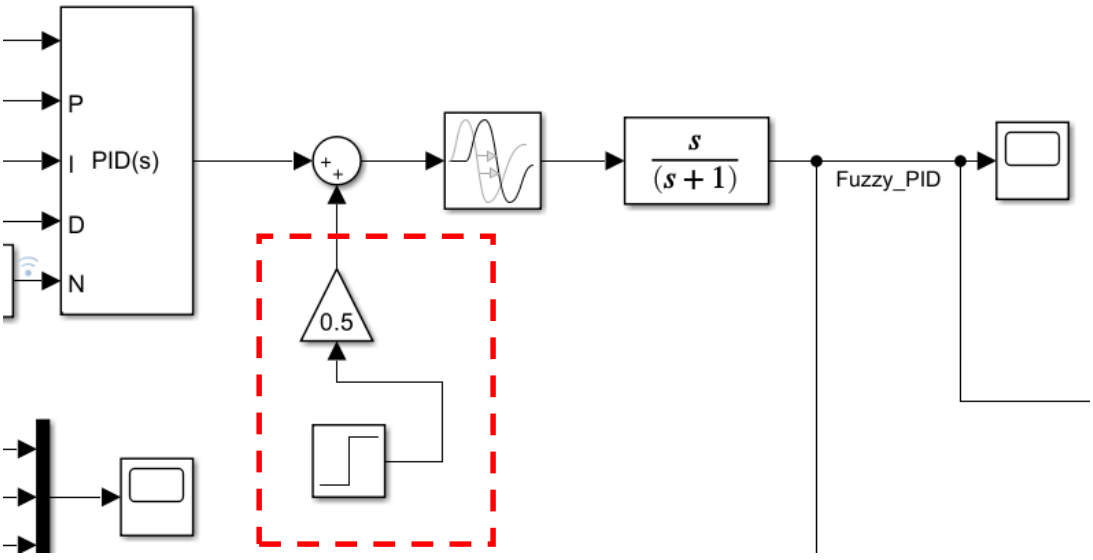
Another result that can be concluded by comparing the responses is that it seems the system with tuned PID has faster response than system with Fuzzy PID.

If we check the gains of tuned PID K_i is 6.8 and K_p is 0.1 and K_d is zero. The value of gains is just like the order of fuzzy controller means it has a high value of K_i a little value of K_p and an almost negligible value of K_d. This means that our designed Fuzzy controller is performing reasonably.

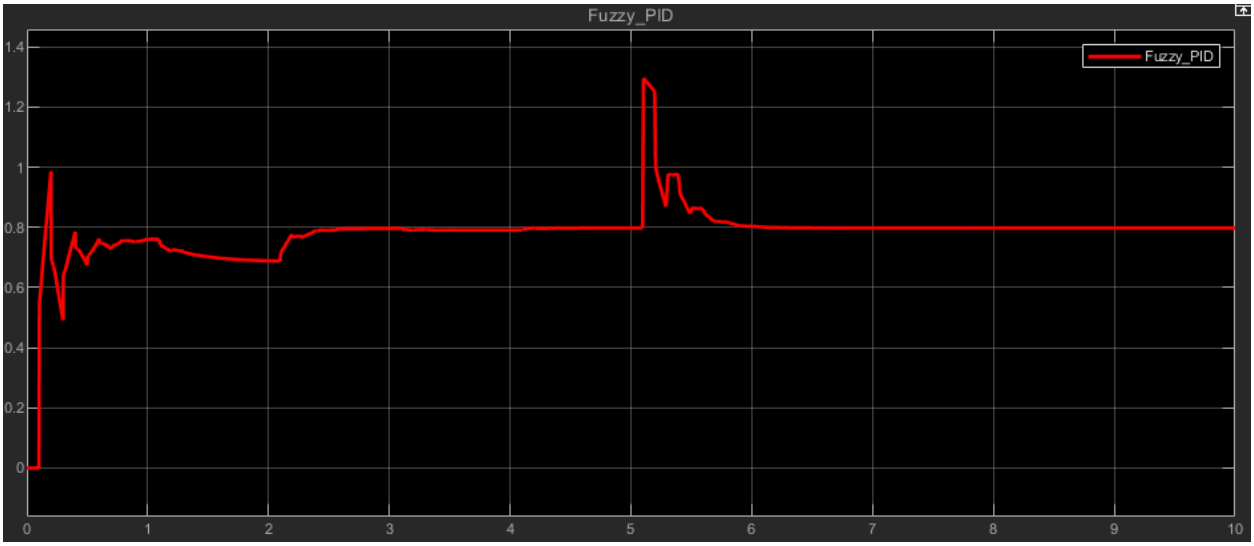
Now in the next section we test the discussed controllers with presence of disturbance and noise. This picture shows how the disturbance and noise should be added to the system.



First we add disturbance to system. To do that we add a half step input to system in the time 5s to the system.

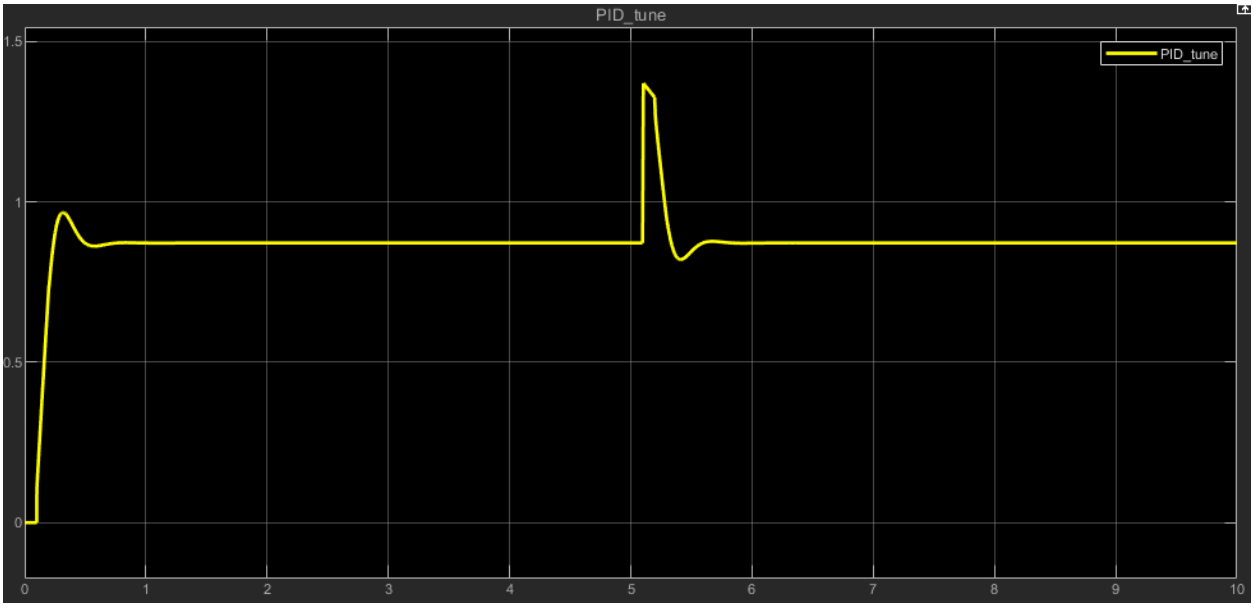


This picture is the response of the Fuzzy PID controller.



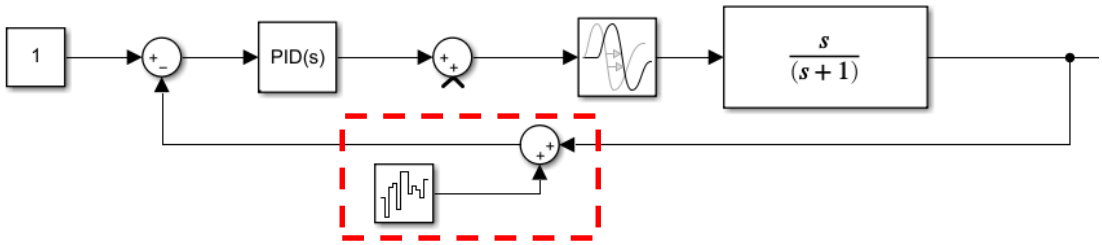
As you can see the controller has successfully controlled the system even with added disturbance and system is not unstable and the disturbance is damped in almost 1 second. The maximum value that this system reached is 1.3.

Now this picture is response of the tuned PID with presence of the disturbance:

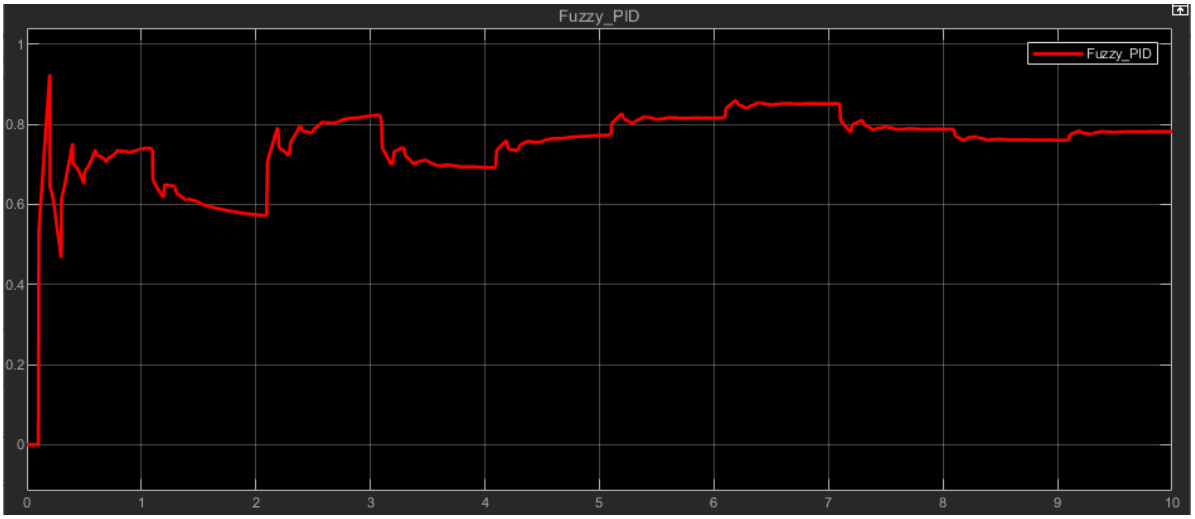


This controller also controlled the disturbance successfully and system is not unstable and the disturbance is damped in almost 0.7s which is at all faster than the fuzzy PID. The maximum value that this system reached is 1.37.

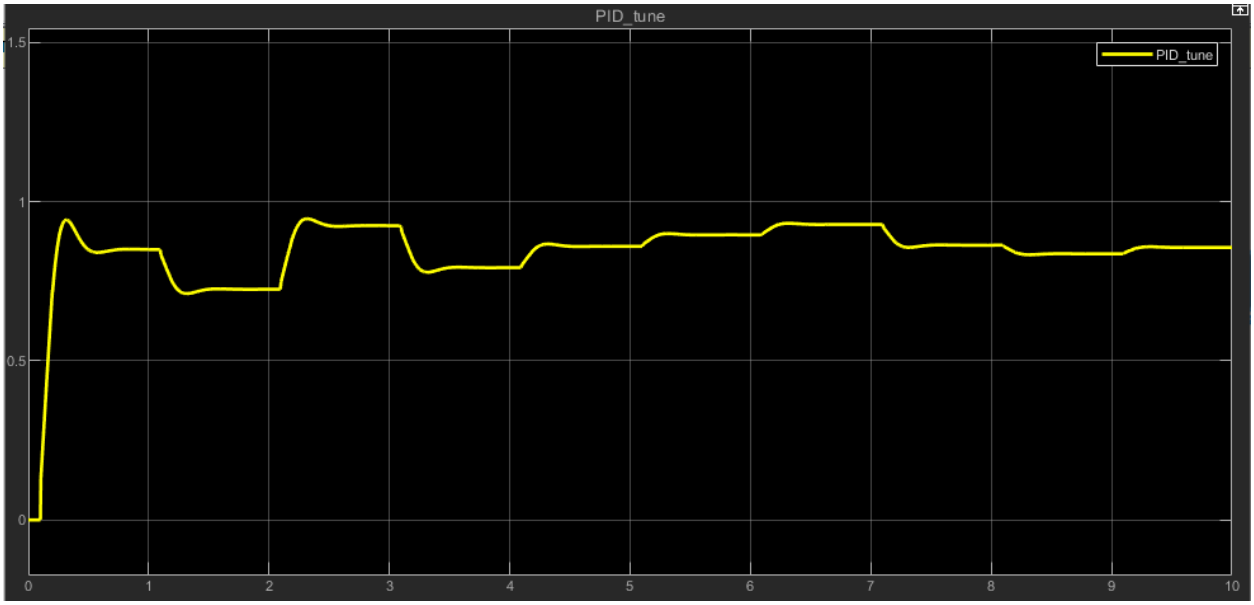
Now we test system in presence of noise. To do this we add a white noise with power factor of 0.005 and sample time of 1s:



This is the response of Fuzzy PID system with presence of white noise:



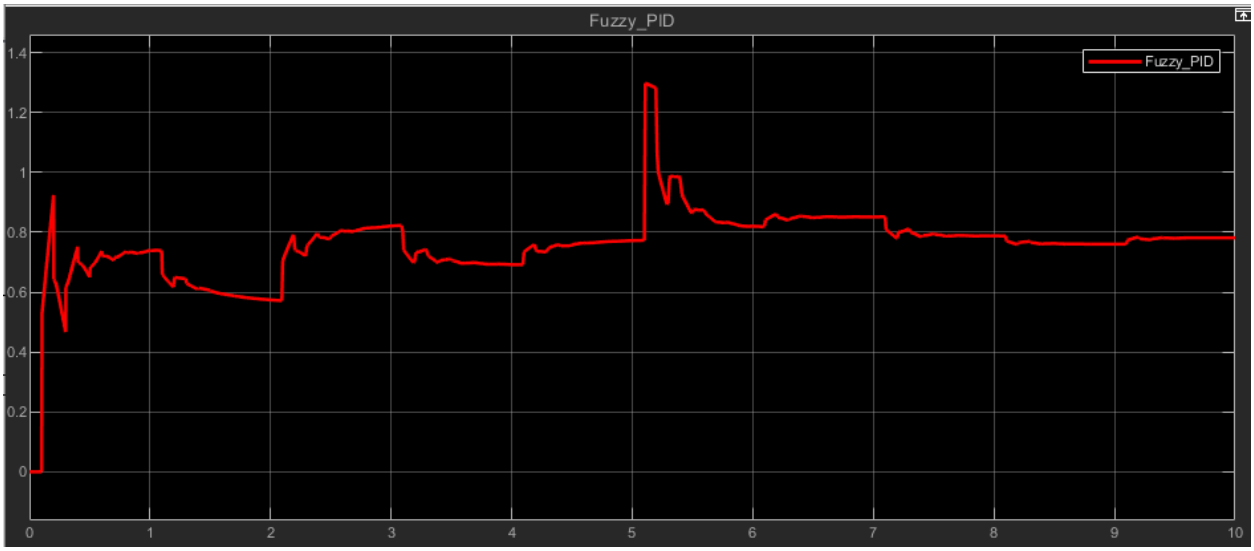
And this is the response of the tuned PID to a noisy measurement of the system.



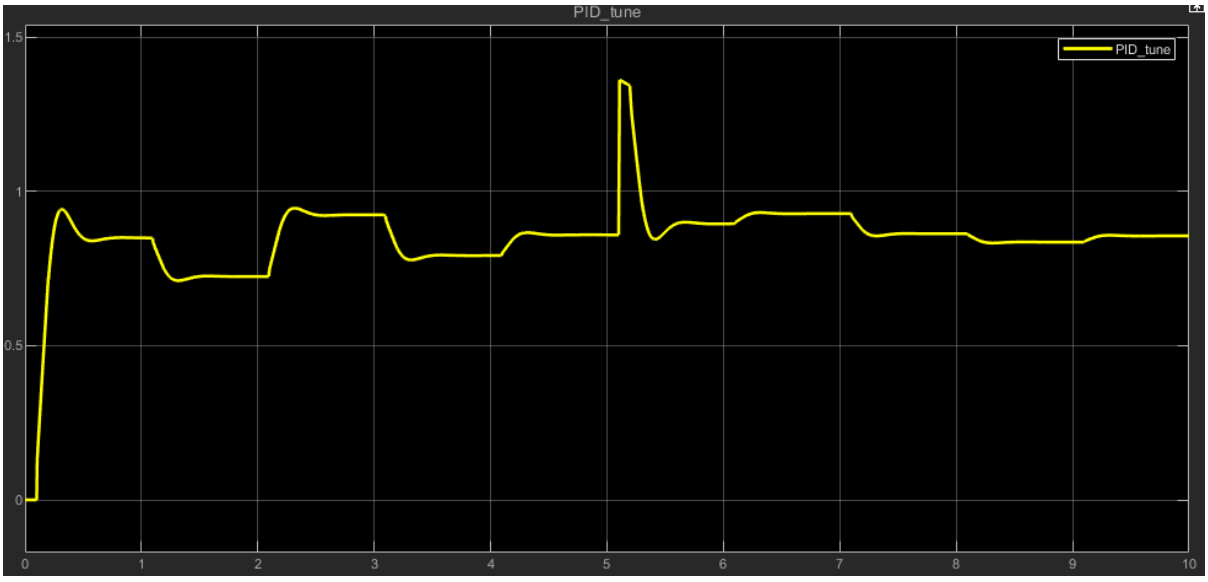
As it is obvious the tuned PID has smoother response but both controllers have successfully controlled the system even with noise.

At last we add both noise and disturbance to system and check its response:

Fuzzy PID controller:



PID tuner:



As last the designed Fuzzy PID controller has acceptable performance, especially that Ziegler Nichols PID couldn’t even control the system, but the controller tuned by matlab seems to perform a little better but the fuzzy controller still can be optimized and improved. B

Question2:

Our goal in this section is to design a controller that simulates the driver's behavior when reversing a component. The component's position is described by three state variables: x , ϕ, and θ, where θ represents the angle of the component relative to the horizontal axis. The control is achieved using the steering angle α, as shown in Figure 1. The steering angle is positive for clockwise movement and negative for counterclockwise movement. For simplicity, we assume that the space between the component and the wall is sufficient, so it is not considered as a state variable. The input and output spaces are defined as $V = [\theta] = [-40.40]$ and $U = [x] \times [\phi] = [0.20] \times [-90.270]$, respectively. Our ultimate goal is to design a controller with inputs $(x.\phi)$ and output (θ).

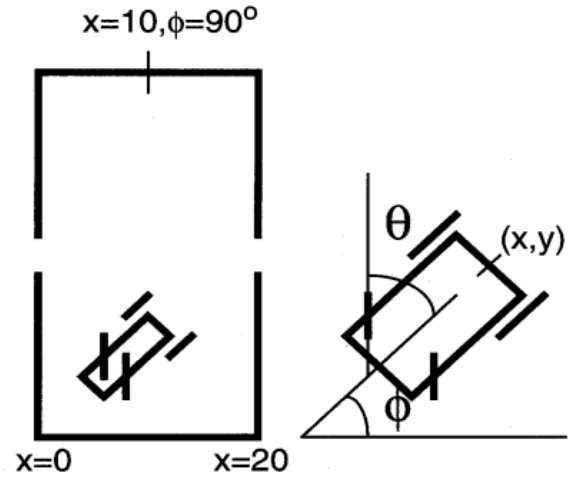


Figure 12.4. The simulated truck and loading zone.

First, we generate input-output pairs $(\boldsymbol{xp}.\boldsymbol{\phi p};\boldsymbol{\theta p})$ through trial and error. Starting from an initial state, we determine the control $\boldsymbol{\theta}$ based on common sense and experience. After several trials, we select the pairs that result in the smoothest successful trajectory. We used 14 initial states to generate these pairs:

$(x_0.\phi_0)$
$= (1.0); (1.90); (1.270); (7.0); (7.90); (7.180); (7.270); (13.0); (13.90); (13.180);$

$(13.270); (19.90); (19.180); (19.270)$

From the whole 14 initial conditions of above we would have almost 250 input-output pair now we can create a Fuzzy system with Lookup Table method with this data.

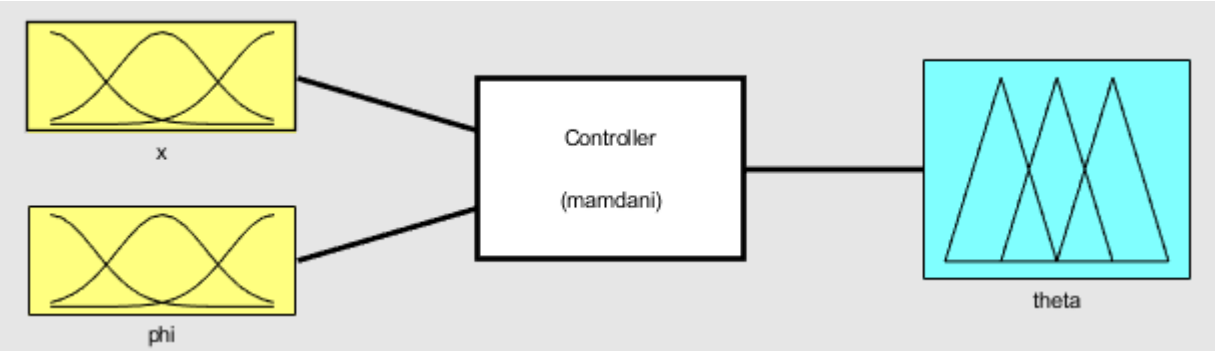
By considering the data that has been gathered a fuzzy system using the table look-up scheme has been developed which the look-up table is as follows:

ϕ \ x	S2	S1	CE	B1	B2
S3	S2	S3			
S2	S2	S3	S3	S3	
S1	B1	S1	S2	S3	S2
CE	B2	B2	CE	S2	S2
B1	B2	B3	B2	B1	S1
B2		B3	B3	B3	B2
B3				B3	B2

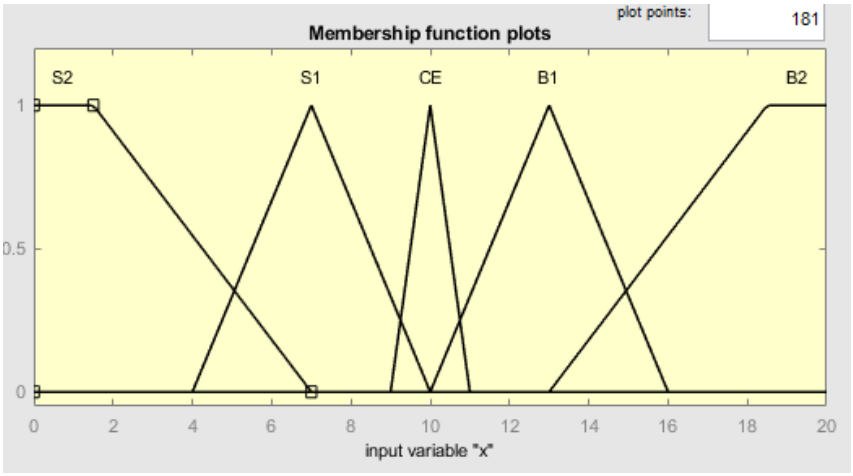
we see that some boxes are empty, so the input-output pairs do not cover all the state space. however, we will see that the rules are sufficient for controlling the truck to the desired position starting from a wide range of initial positions.

We define 7 membership functions for ϕ and 5 membership functions for x and 7 membership function for θ. Type of membership functions are triangular (and few trapezoidal).

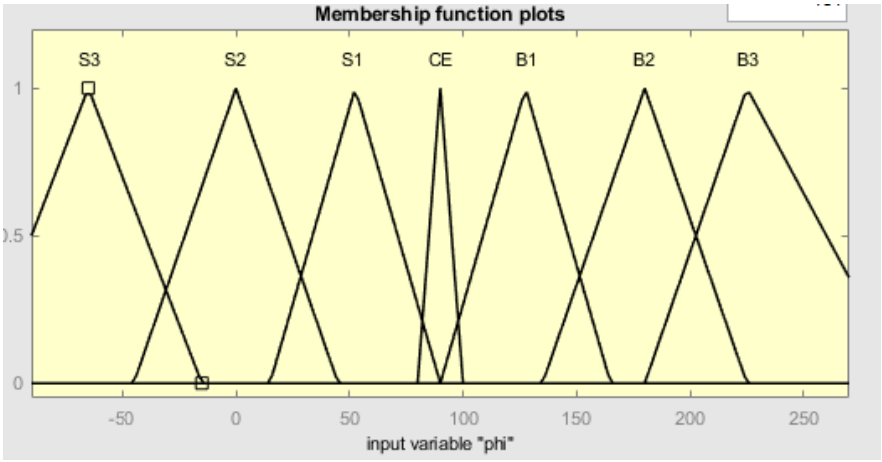
Here in the picture below you can see the FIS system which is created with the above rules and it has two inputs and one output.



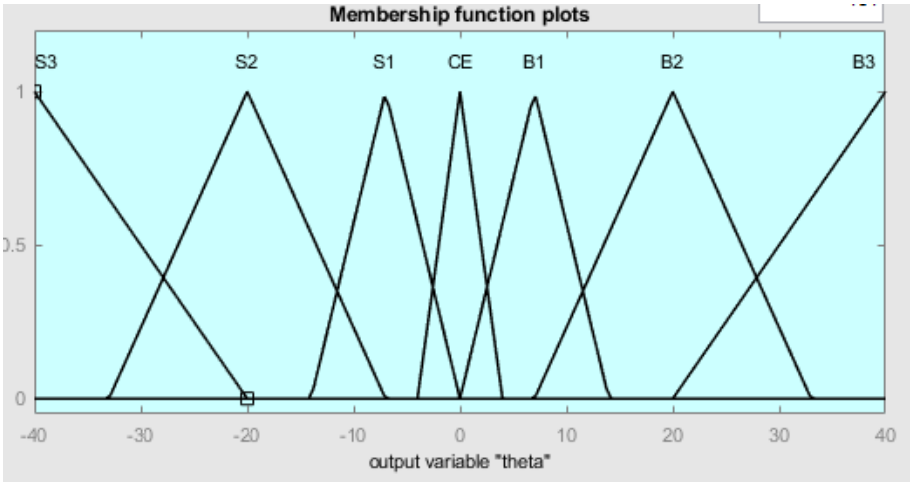
Here are the membership functions:



Membership functions of the input x

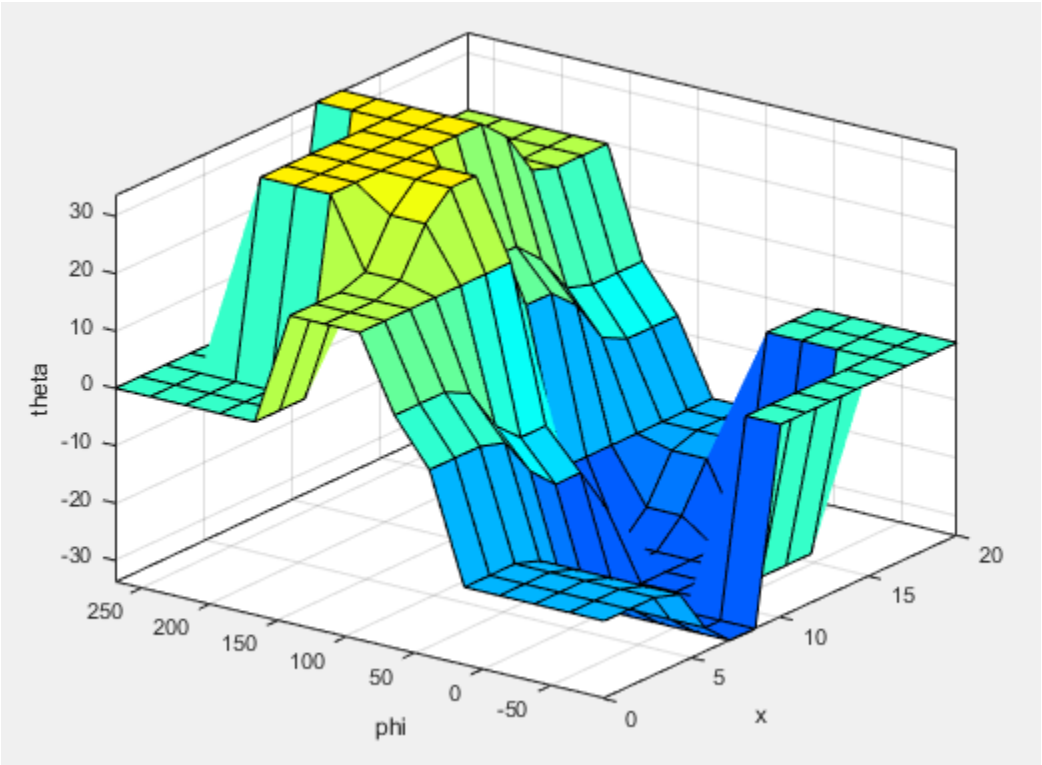


Membership functions of the input ϕ



Membership functions of the output θ

In the surface below you can see the output of fuzzy system based on the inputs:



Now for testing our fuzzy inference system we should set an initial position $(x_0.\phi_0)$ and write a loop that evaluates θ and then compute the next step position by these kinematic equations:

$$x(t+1) = x(t) + cos[\phi(t) + \theta(t)] + sin[\theta(t)]sin[\phi(t)]$$
$$y(t+1) = y(t) + sin[\phi(t) + \theta(t)] - sin[\theta(t)]cos[\phi(t)]$$
$$\phi(t+1) = \phi(t) - sin^{-1}[\frac{2sin(\theta(t))}{b}]$$

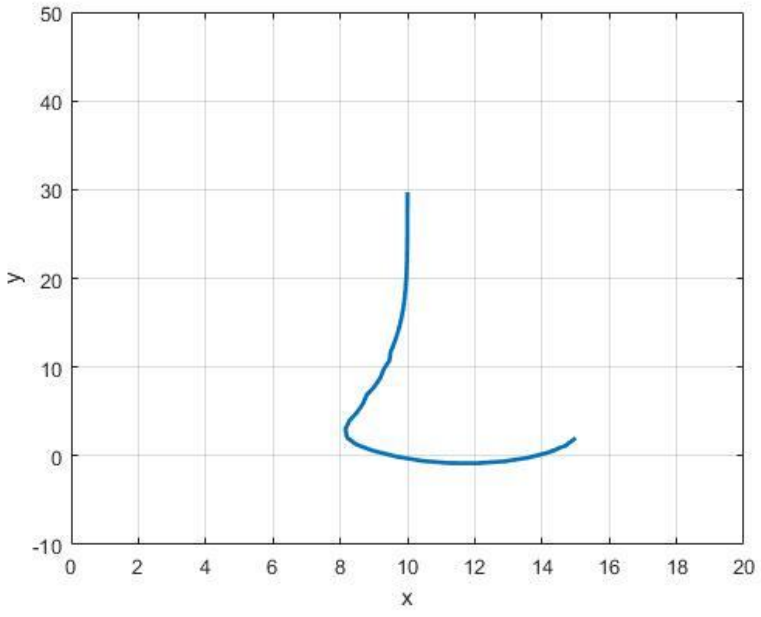
This loop should be continued and position should be updated until the error which is difference between current position and target position decreases to a certain level like 0.01.

The code below is the implementation up the above algorithm:

```
x = zeros(1,n);
phi = zeros(1,n);
y = zeros(1,n);
y(1,1) = 2;

x(1,1) = input('Enter Initial Point for x(0<x<20) & Press Enter Key: ');
```

```
phi(1,1) = input('Enter Initial Point for phi(-90<phi<270 degree) and then
Press Enter Key: ');

x_desired = 10;
phi_desired = 90;

% Cost Function
J = norm([x_desired-x(1,1) phi_desired-phi(1,1)])

t=1;
while(J>=0.01)
    theta = evalfis([x(1,t);phi(1,t)],fis);
    Data_Table(t,:) = [t-1,x(1,t),y(1,t),phi(1,t),theta];
    x(1,t+1) =
x(1,t)+cos((phi(1,t)+theta).*pi/180)+sin(theta*pi/180)*sin(phi(1,t).*pi/180);
...
    phi(1,t+1) = phi(1,t) - (asin(2*sin(theta*pi/180)/b))*180/pi;
    y(1,t+1) = y(1,t)+sin((phi(1,t)+theta)*pi/180)-
sin(theta*pi/180)*cos(phi(1,t)*pi/180);

    J = norm([x_desired-x(1,t+1) ...
        phi_desired - phi(1,t+1)]);
    t = t+1
end

x_truck(1,:) = x(1,1:t);
phi_truck(1,:) = phi(1,1:t);
y_truck(1,:) = y(1,1:t);
```

after this we have whole trajectory of the truck and we can plot it. For example for an initial condition of (15.250) the trajectory would be like this:



As it can be seen from the figure the FIS have successfully controlled truck to the target position and the final position in this case is:

$$x_{Final} = 9.9996$$

$$\phi_{final} = 89.9907$$

$$y_{Final} = 29.6992$$

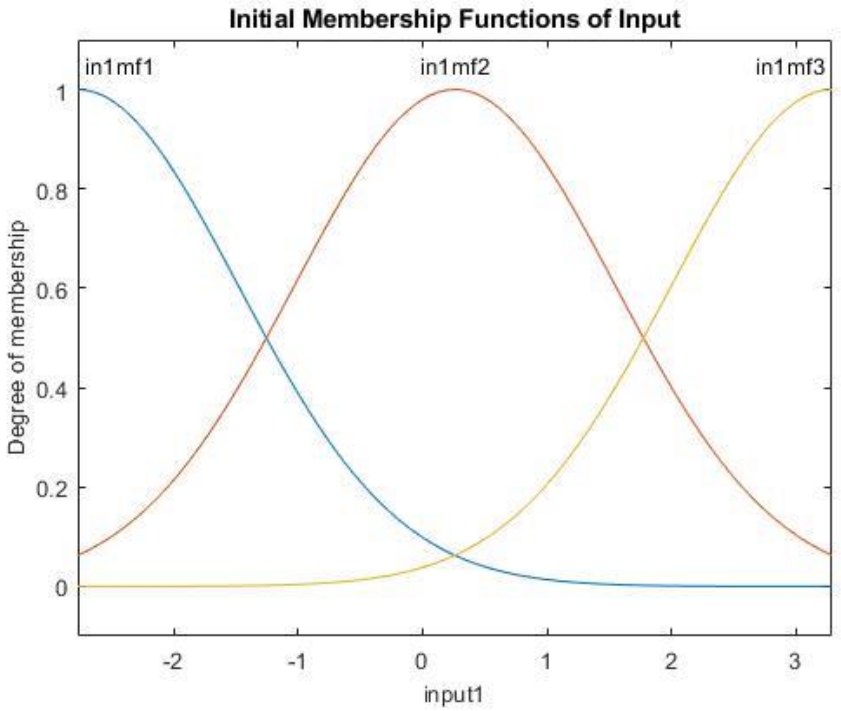
Question3:

The ball beam system is a single input single output system. To identify such system, we can use an ANFIS network.

To do system identification it's better to normalize data first.

First we should create a Fuzzy Inference System which we do with genfis1 method of matlab and then create an ANFIS network with the anfis method of matlab.

We have one input and by trial and error we define 4 Gaussian membership function for this data. The initial configuration of MFs are like the picture below:



The genfis1 function in MATLAB is used to generate an initial Fuzzy Inference System (FIS) structure for Sugeno-type fuzzy systems. It employs **grid partitioning** to create the FIS, which means it divides the input space into a grid of membership functions (MFs) based on the specified number of MFs for each input. This method is particularly useful when the structure of the input-output relationship is relatively simple and well-understood. The genfis1 function requires the user to specify the number of MFs for each input variable, as well as the type of MFs (Gaussian, triangular, …). The output MFs are typically linear or constant in Sugeno-type systems.

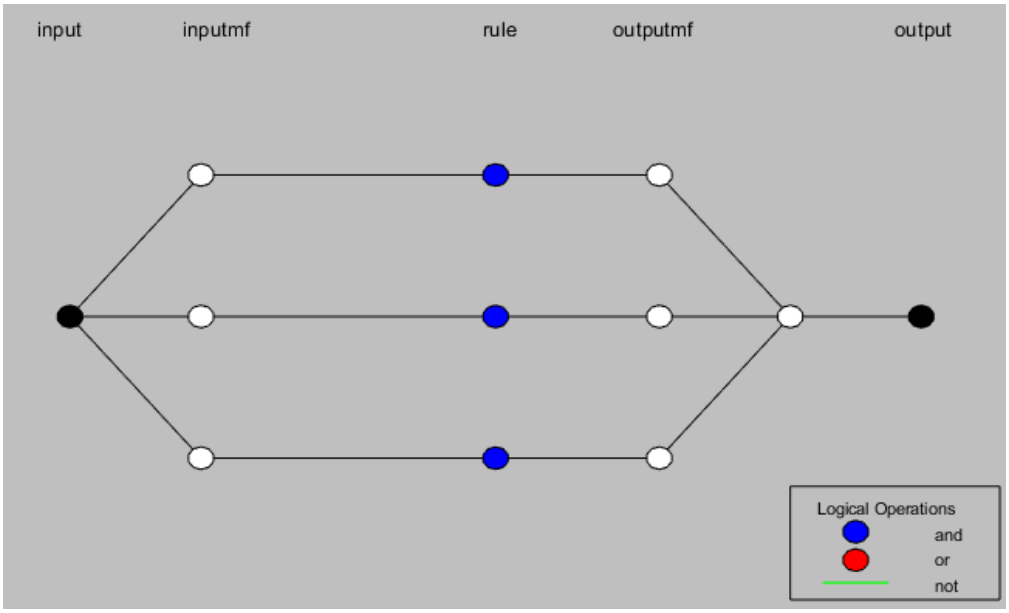
The FIS generated by genfis1 consists of a set of fuzzy rules that map the input variables to the output variable. Each rule is defined by the combination of MFs for the input variables. For example, if there are two inputs with three MFs each, genfis1 will create a rule base with $3 \times 3 = 9$ rules. The rules are of the form:

$$if\ x_1\ is\ A_1\ and\ x_2\ is\ A_2 . then\ y = f(x_1 . x_2)$$

where A_1 and A_2 are fuzzy sets (MFs) for the inputs x_1 and x_2, and $f(x_1 . x_2)$ is a linear or constant function of the inputs.

Next we create an ANFIS network with use of the FIS we created before.

The structure of the ANFIS network would be like this:



The input membership functions are Gaussian and output membership functions are constant.

ANFIS uses a hybrid learning algorithm that combines gradient descent and least squares estimation to tune the parameters of the fuzzy inference system. The learning process consists of two main phases:

Forward Pass (Least Squares Estimation):

- In this phase, the consequent parameters (the coefficients of the linear output functions in Sugeno-type systems) are estimated using the least squares method.

- This step minimizes the error between the predicted output and the actual output.
- The premise parameters (the parameters of the input MFs) are fixed during this phase.

Backward Pass (Gradient Descent):

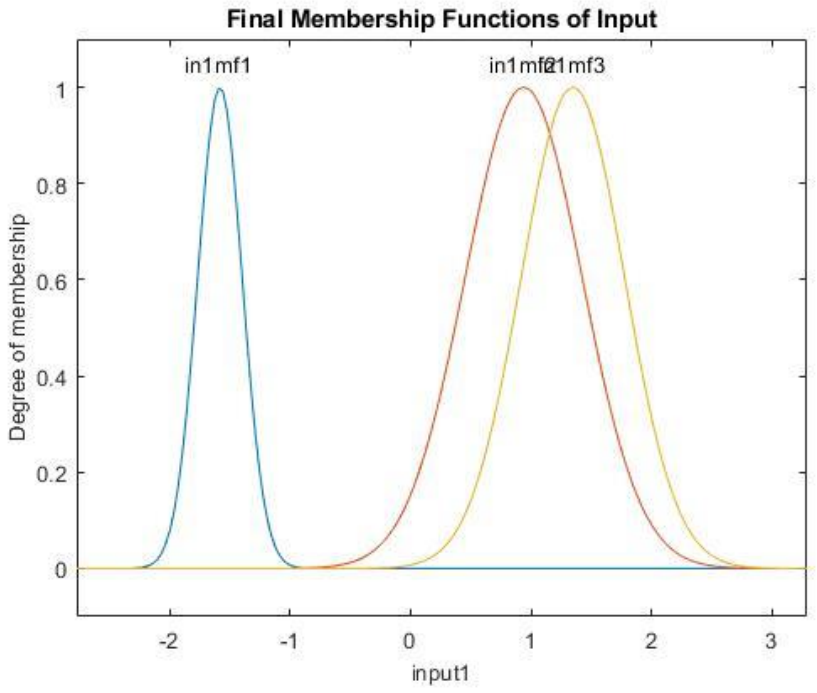
- In this phase, the premise parameters (the parameters of the input MFs, such as the center and width of Gaussian MFs) are updated using gradient descent.
- This step adjusts the shape and position of the MFs to further reduce the error.
- The consequent parameters are fixed during this phase.

The parameters of ANFIS can be divided into two categories:

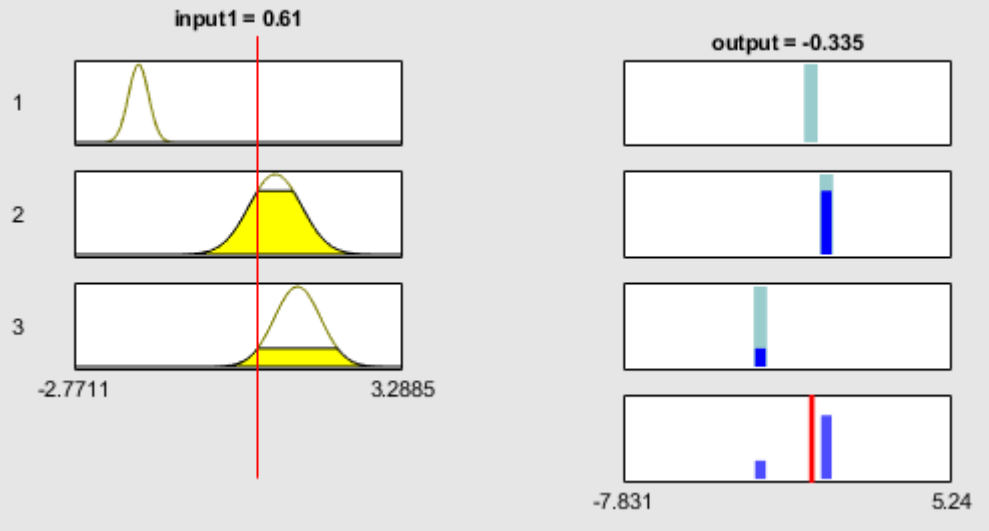
Premise Parameters which are the parameters of the input MFs, such as the center ($\boldsymbol{c}$) and width ($\boldsymbol{\sigma}$) of Gaussian MFs or the vertices of triangular MFs.

Consequent Parameters which are the coefficients of the linear or constant output functions in Sugeno-type systems.

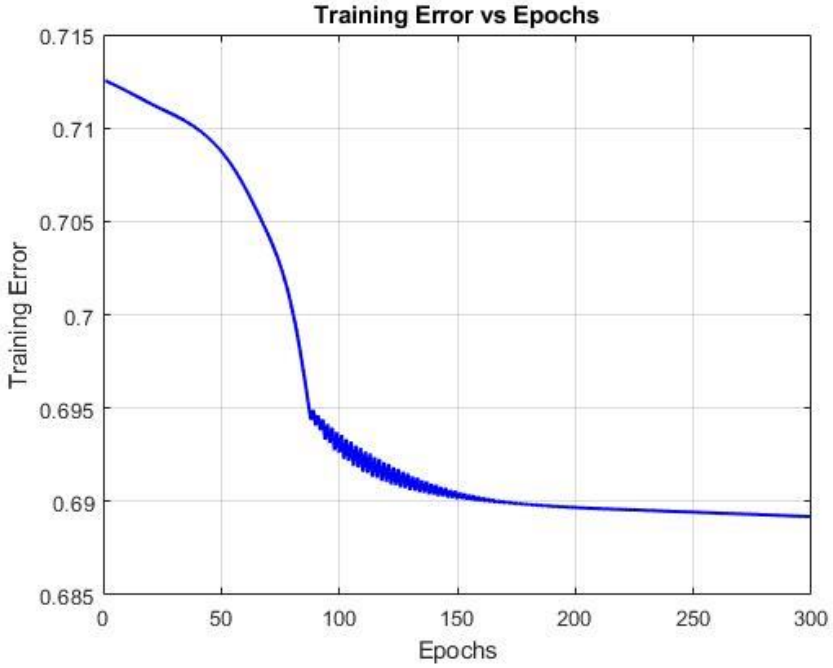
After training phase, the configuration of membership function would become like this:



The picture below is the arrangement of the rules and how they affect the output.



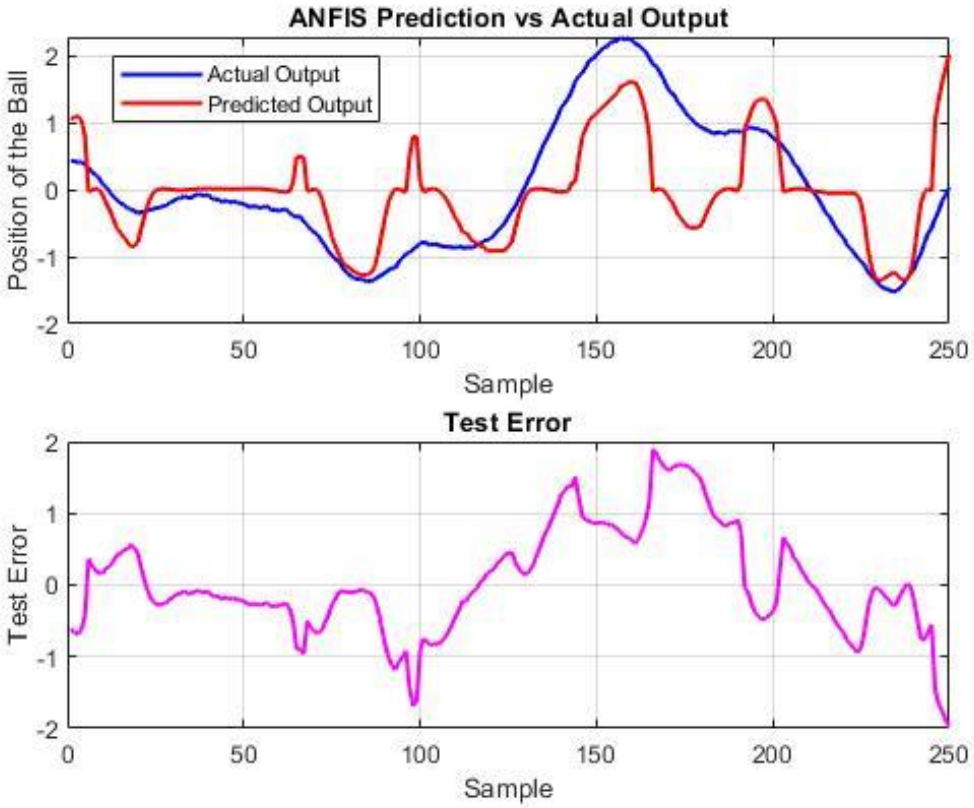
The plot of error based on epoch in training phase is in the figure below:



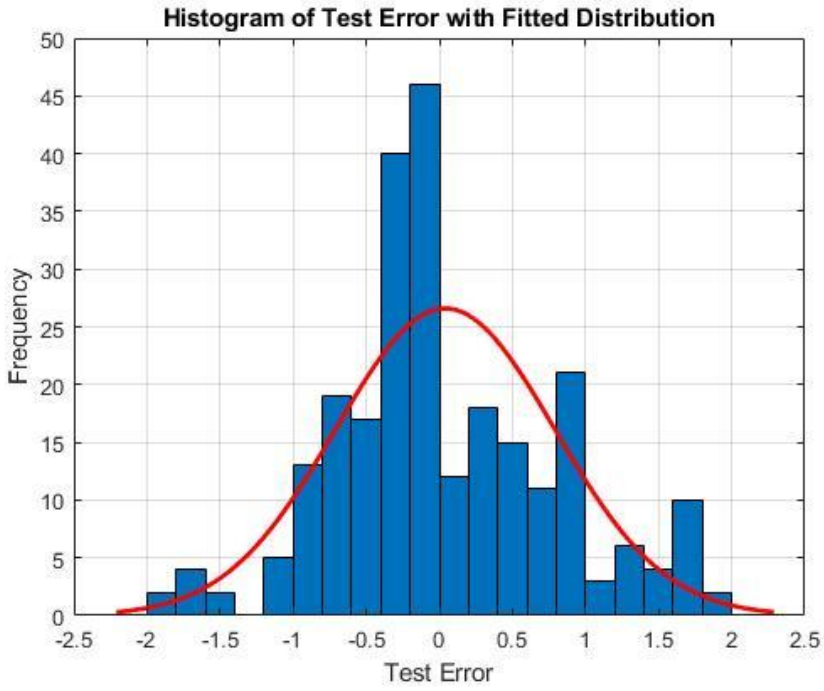
The minimal training RMSE here would be 0.689188.

Now we use test data to evaluate the model.

In the figure below you can see the original data and the fitted curve generated by ANFIS and also the error between them.



The error histogram is also like this:



The test data RMSE of this model is 0.749453.

Question4:

The goal is to identify a nonlinear component in a control system using ANFIS.

Differential equation of the system is as follows:

$$y(k+1) = 0.3y(k) + 0.6y(k-1) + f(u(k))$$

- $y(k)$ and $u(k)$ are the output and input of the system at time index k.
- $f(u(k))$ is an unknown nonlinear function that needs to be identified.

The unknown nonlinear function $f(u)$ is defined as:

$$f(u) = 0.6sin(\pi u) + 0.3sin(3\pi u) + 0.1sin(5\pi u)$$

Based on the paper to identify the plant, a **series-parallel model** is used, which is governed by the following difference equation:

$$\hat{y}(k+1) = 0.3y(k) + 0.6y(k-1) + F(u(k))$$

$F(u(k))$ is the function implemented by ANFIS, which is used to approximate the unknown nonlinear function $f(u(k))$.

So the input to ANFIS is $u(k)$, the input to the plant and the output of ANFIS is an approximation of $\boldsymbol{f(u(k))}$.

From the plant dynamics, we can isolate $\boldsymbol{f(u(k))}$ as:

$$f(u(k)) = y(k+1) - 0.3y(k) - 0.6y(k-1)$$

So we can use this equation to create a training set for our network.

We define a sinusoidal input for training data as the paper have said:

$$u(k) = sin(\frac{2\pi k}{250})$$

The training data for ANFIS consists of input-output pairs $(u(k).f(u(k))$.

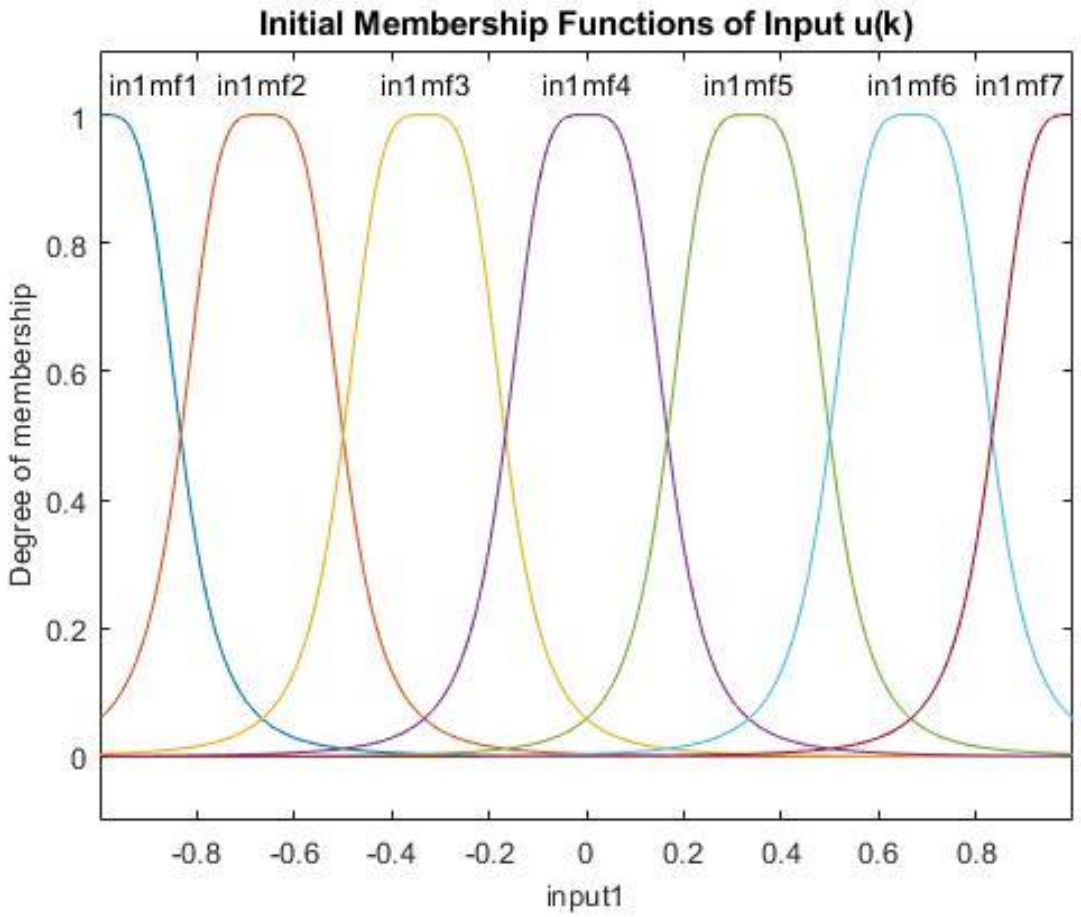
The code below is for creating training data based on the differential equation of the system

```
y_train = zeros(1, k_max); % Initialize output vector
y_train(1) = 0; % Initial condition y(1)
y_train(2) = 0; % Initial condition y(2)
```

```
for k = 2:k_max-1
    y_train(k+1) = 0.3*y_train(k) + 0.6*y_train(k-1) + f(u_train(k)); % Plant
dynamics
end

% Prepare training data for ANFIS: [u(k), f(u(k))]
f_u_train = zeros(1, k_max-2); % Initialize f(u(k)) vector
for k = 2:k_max-1
    f_u_train(k-1) = y_train(k+1) - 0.3*y_train(k) - 0.6*y_train(k-1); %
Compute f(u(k))
end
input_data_train = u_train(2:k_max-1)'; % Input: u(k)
output_data_train = f_u_train'; % Output: f(u(k))
```

Once trained, ANFIS can predict $f(u(k))$ for any given $u(k)$ and in this way we have identified the nonlinear term in the dynamic differential equation of our system.

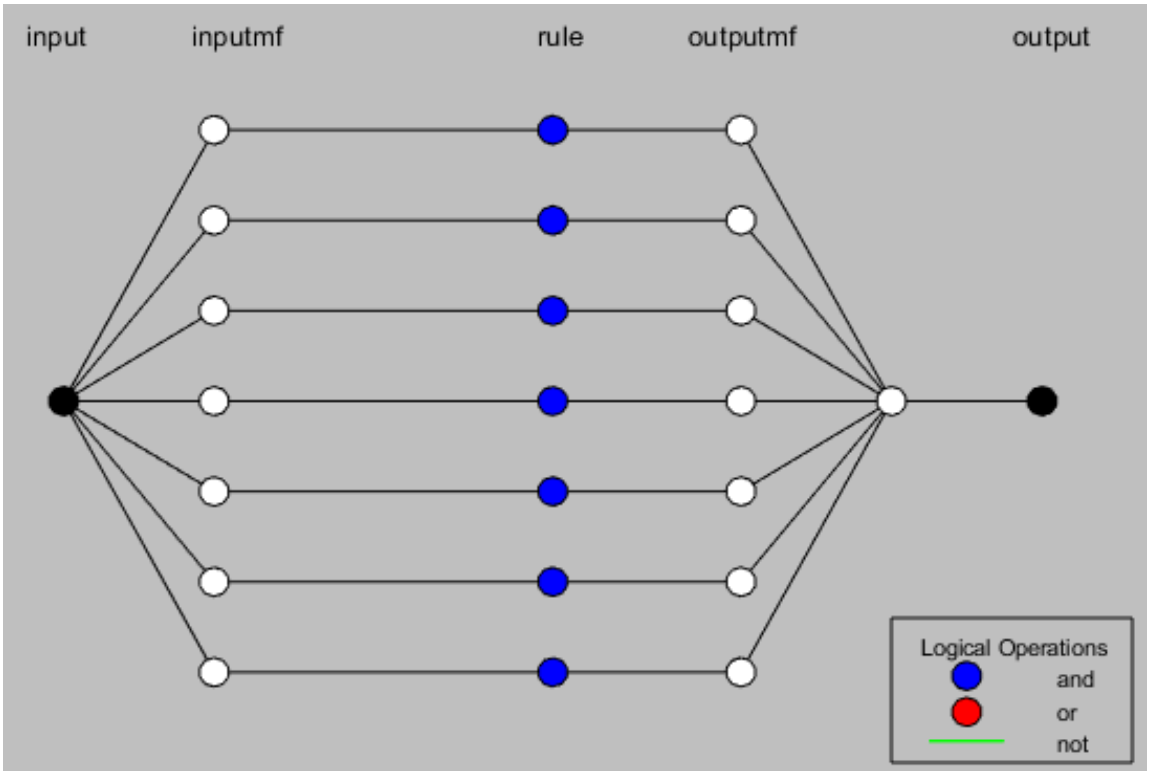
To do this first we create a FIS system and we define 7 generalized bell shape membership functions. Which are like the figure below:



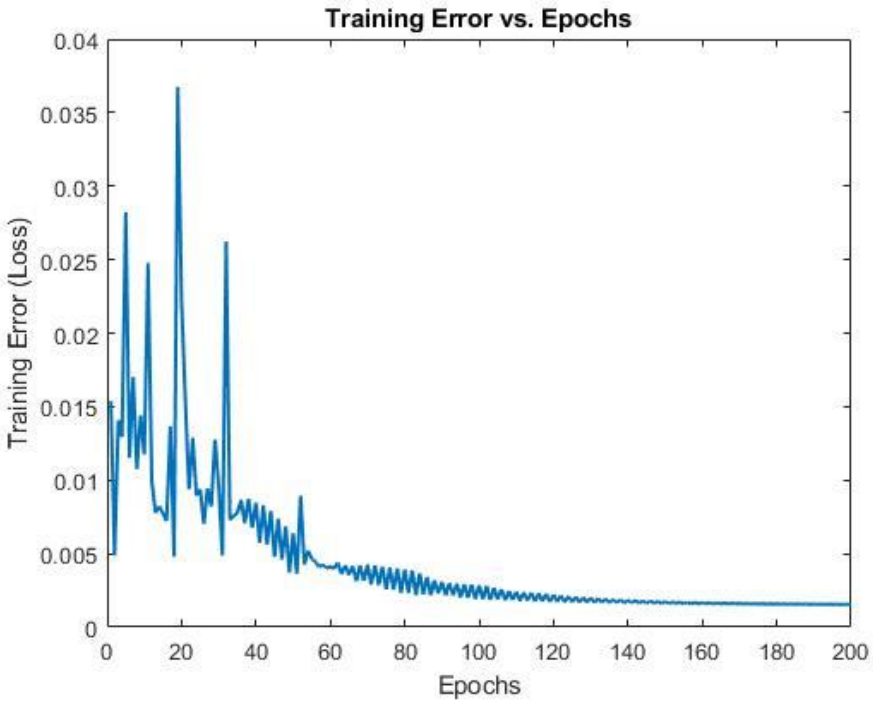
We use genfis method of matlab to create our FIS.

The FIS generated by genfis is designed to capture the relationship between the input and output data in a fuzzy logic framework. The grid partitioning method ensures that the input space is systematically explored, and the rules are created to cover all possible combinations of input membership functions. This approach is particularly useful when there is no prior knowledge of the system being modeled, as it automatically generates a rule base that can be fine-tuned during the training process. The membership functions are initialized with parameters that define their shape, such as the center and width for Gaussian functions, and these parameters are later optimized during the ANFIS training phase. By creating an initial FIS structure, genfis provides a structured and interpretable model that can be further improved using data-driven learning techniques, making it a crucial step in the development of ANFIS-based systems.

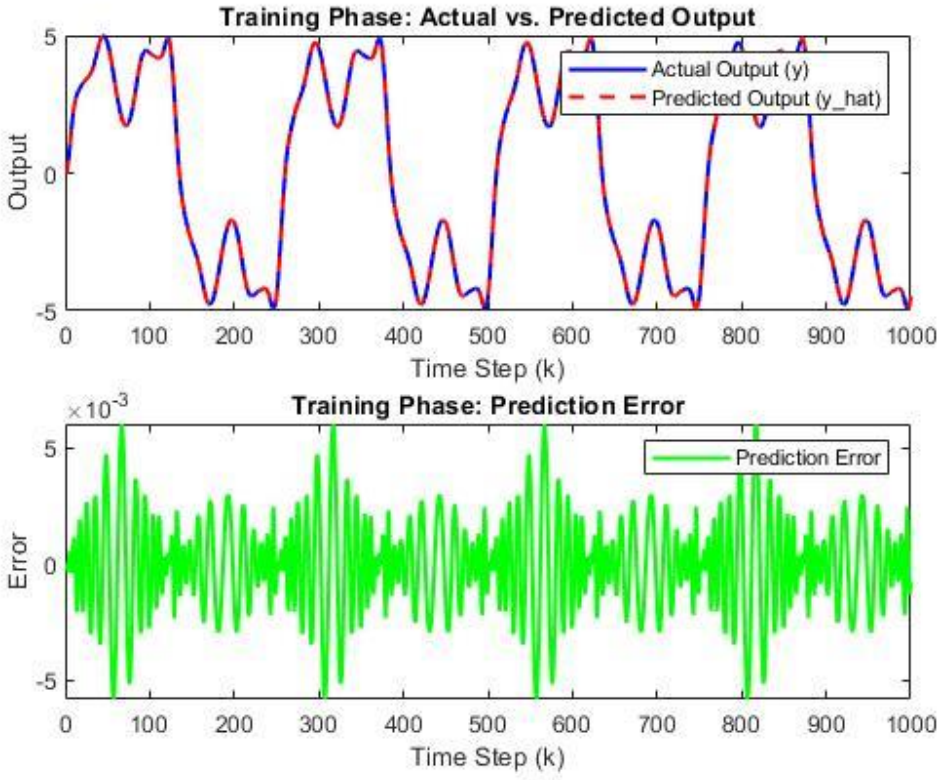
Next step is to use anfis method of matlab to build a network and train it. The structure of the network is like this:



The training is done in 200 epochs and plot of error based on epoch of the network in presented in the figure below:

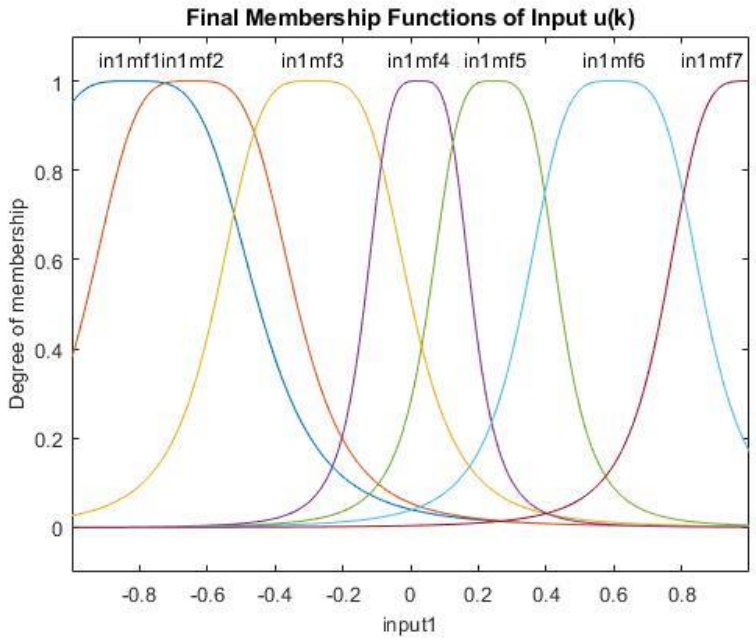


Minimal training RMSE for the ANFIS we created is 0.00154362 which is a very good result. We can also see the performance of the network on the training set and its error in the picture below:

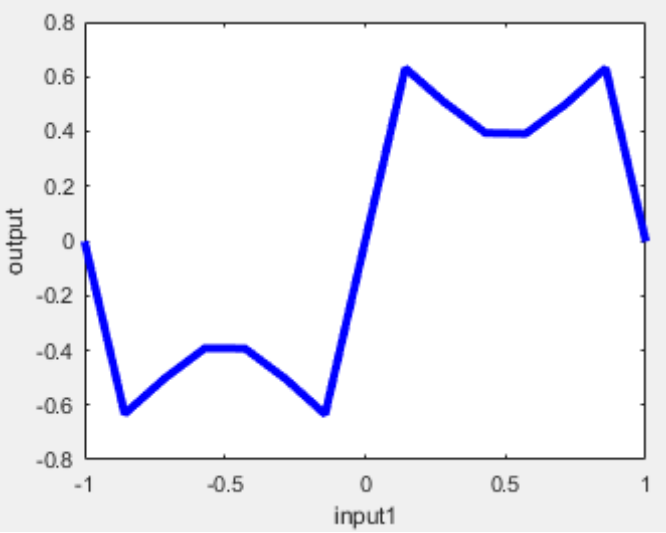


As you can see the actual data curve and predicted data curve have almost negligible difference.

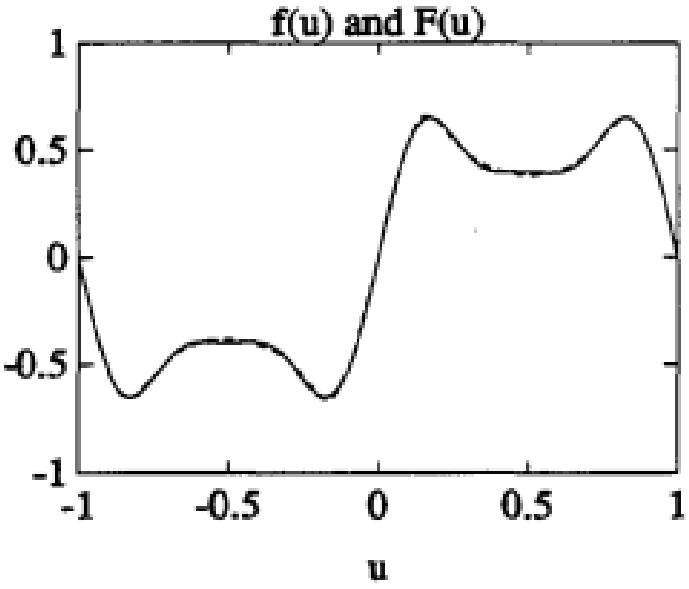
Now that we have trained out ANFIS we can see the final shape of membership functions which their parameters (center c and width σ) are optimized through learning by gradient decent method.



If we look at the surface of the FIS we used in this network which is shape of the identified nonlinear function $f(u)$ and we compare it with the shape reported by the paper we can see that they are largely similar.



Surface of designed FIS



$f(u)$ reported by the paper

We can take this similarity as a validation of our work. This means our designed ANFIS network is a good approximation of the function $f(u)$.

Now we test our network by changing the input to system to see if the ANFIS approximation can correctly estimate the value of $f(u)$ in another case or not.

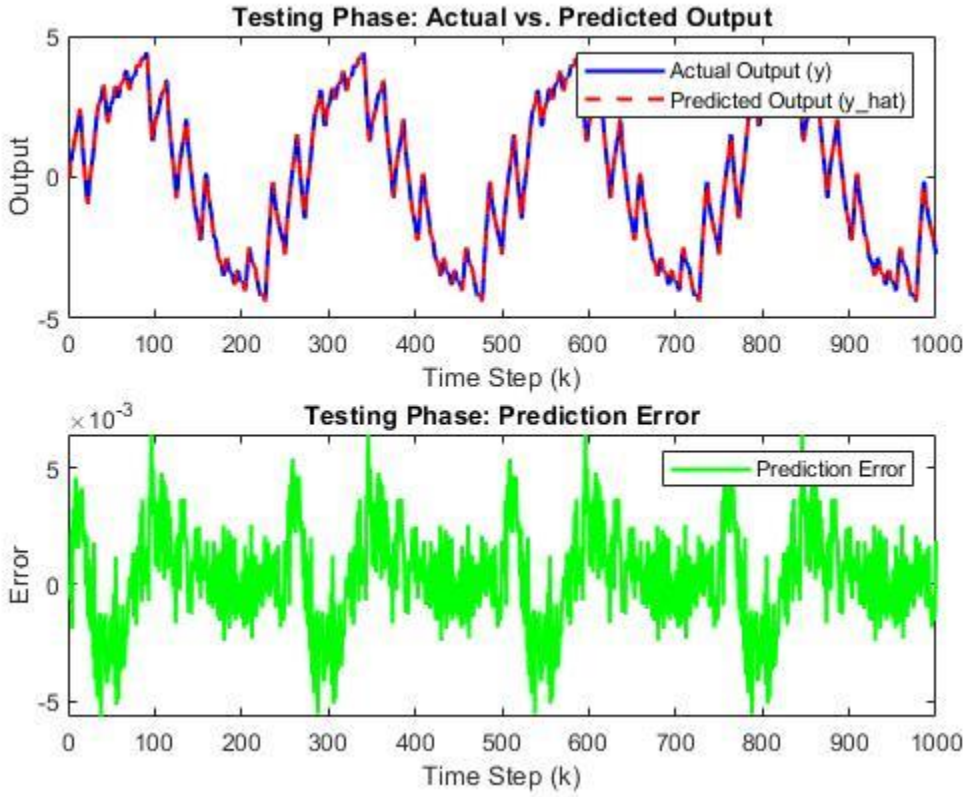
As paper said we use an input like this to create a test set:

$$u(k) = 0.5sin(\frac{2\pi k}{250}) + 0.5sin(\frac{2\pi k}{25})$$

The code below is related to creating test data:

```
% Test the trained ANFIS with a new input signal
k_max_test = 1000; % Total time steps for testing
u_test = 0.5*sin(2*pi*(1:k_max_test)/250) + 0.5*sin(2*pi*(1:k_max_test)/25);
% Test input signal u(k)

% Generate plant output y(k) for the test input
y_test = zeros(1, k_max_test); % Initialize test output vector
y_test(1) = 0; % Initial condition y(1)
y_test(2) = 0; % Initial condition y(2)
for k = 2:k_max_test-1
    y_test(k+1) = 0.3*y_test(k) + 0.6*y_test(k-1) + f(u_test(k)); % Plant
dynamics
end
```

In the picture below you can see the actual data and predicted data on same plot and also the error which is very small value in order of 10^{-3}.



Question5:

This problem is a regression task and goal is to estimate value of NO_2 in air to evaluate quality of air.

We want to examine two methods for doing this task. First we use an ANFIS network and then we use a neural network with RBF hidden layer and we compare their results.

ANFIS:

To use data for training we should first do some preprocessing on dataset.

First we should find missing values and remove the rows which have missing value.

If we carefully look at dataset there are many terms with value of -200 which seems to be a little unusual to be a valid data.

4/8/2004	19:00:00	6.3	1618	974	29.1	1530	326	579	171	2167	1791	18.0	32.9	0.6737
4/8/2004	20:00:00	4.3	1319	544	15.8	1172	232	746	136	1699	1425	15.8	39.8	0.7096
4/8/2004	21:00:00	1.6	1045	138	7.0	857	92	968	93	1410	922	15.3	41.5	0.7194
4/8/2004	22:00:00	1.4	1058	92	6.3	824	95	993	99	1411	971	14.1	46.0	0.7363
4/8/2004	23:00:00	2	-200	137	-200.0	-200	129	-200	106	-200	-200	-200	-200	-200
4/9/2004	0:00:00	2.4	-200	189	-200.0	-200	154	-200	109	-200	-200	-200	-200	-200
4/9/2004	1:00:00	1.8	-200	159	-200.0	-200	118	-200	97	-200	-200	-200	-200	-200
4/9/2004	2:00:00	1	-200	80	-200.0	-200	69	-200	83	-200	-200	-200	-200	-200
4/9/2004	3:00:00	1	-200	66	-200.0	-200	-200	-200	-200	-200	-200	-200	-200	-200
4/9/2004	4:00:00	1	-200	87	-200.0	-200	97	-200	79	-200	-200	-200	-200	-200
4/9/2004	5:00:00	0.9	-200	79	-200.0	-200	145	-200	84	-200	-200	-200	-200	-200
4/9/2004	6:00:00	1.5	-200	150	-200.0	-200	169	-200	86	-200	-200	-200	-200	-200
4/9/2004	7:00:00	2.6	-200	196	-200.0	-200	250	-200	111	-200	-200	-200	-200	-200
4/9/2004	8:00:00	2.9	-200	299	-200.0	-200	215	-200	117	-200	-200	-200	-200	-200
4/9/2004	9:00:00	2.7	-200	254	-200.0	-200	237	-200	122	-200	-200	-200	-200	-200
4/9/2004	10:00:00	2.4	-200	226	-200.0	-200	190	-200	119	-200	-200	-200	-200	-200
4/9/2004	11:00:00	2.6	-200	262	-200.0	-200	219	-200	121	-200	-200	-200	-200	-200
4/9/2004	12:00:00	4.1	-200	505	-200.0	-200	294	-200	127	-200	-200	-200	-200	-200
4/9/2004	13:00:00	4.3	-200	512	-200.0	-200	253	-200	135	-200	-200	-200	-200	-200
4/9/2004	14:00:00	2.2	-200	159	-200.0	-200	176	-200	119	-200	-200	-200	-200	-200
4/9/2004	15:00:00	3.4	-200	337	-200.0	-200	243	-200	137	-200	-200	-200	-200	-200
4/9/2004	16:00:00	3.6	-200	357	-200.0	-200	214	-200	139	-200	-200	-200	-200	-200
4/9/2004	17:00:00	2.5	-200	194	-200.0	-200	141	-200	117	-200	-200	-200	-200	-200
4/9/2004	18:00:00	2.8	-200	278	-200.0	-200	166	-200	127	-200	-200	-200	-200	-200
4/9/2004	19:00:00	3.4	-200	303	-200.0	-200	197	-200	137	-200	-200	-200	-200	-200
4/9/2004	20:00:00	3	-200	234	-200.0	-200	191	-200	127	-200	-200	-200	-200	-200

The dataset contains air quality measurements, including columns like CO(GT), PT08.S1(CO), NMHC(GT), C6H6(GT) and … which are real-world datasets, and in such datasets missing values can occur due to sensor malfunctions, data collection errors, or other issues.

The value -200 is not a physically meaningful value for air quality measurements. For example, concentrations of gases like CO, NO2, or benzene cannot be negative. Sensor readings are typically positive values or zero (if no gas is detected).

So the rows in the dataset which have value of -200 should be removed. To remove them we replace every -200 with NAN and then remove them.

The code below is to remove missing values.

```
% Handle missing values (replace -200 with NaN)
inputData(inputData == -200) = NaN;
outputData(outputData == -200) = NaN;

% Remove rows with missing values
validRows = ~any(isnan(inputData), 2) & ~isnan(outputData);
inputData = inputData(validRows, :);
outputData = outputData(validRows, :);
```

Next step is normalization. We first calculate mean and standard deviation of data and then use normalization function to normalize it. Here is the code:

```
% Manually calculate mean (mu) and standard deviation (sigma) for input and
output data
inputMu = mean(inputData, 1);  % Mean of each column (1x12 array)
inputSigma = std(inputData, 0, 1);  % Standard deviation of each column (1x12
array)

outputMu = mean(outputData, 1);  % Mean of output data (scalar)
outputSigma = std(outputData, 0, 1);  % Standard deviation of output data
(scalar)

% Normalize the input and output data using MATLAB's normalize function
inputDataNorm = normalize(inputData);  % Normalize input data
outputDataNorm = normalize(outputData);  % Normalize output data
```

Next we should split the dataset and keep 60% for training 20% for test and 20% for validation.

Here is the code to split the dataset.

```
% Split the data into training, testing, and validation sets
rng(1);  % For reproducibility
n = size(inputDataNorm, 1);
trainIndices = 1:round(0.6*n);
testIndices = round(0.6*n)+1:round(0.8*n);
valIndices = round(0.8*n)+1:n;

trainInput = inputDataNorm(trainIndices, :);
trainOutput = outputDataNorm(trainIndices, :);

testInput = inputDataNorm(testIndices, :);
testOutput = outputDataNorm(testIndices, :);

valInput = inputDataNorm(valIndices, :);
valOutput = outputDataNorm(valIndices, :);
```

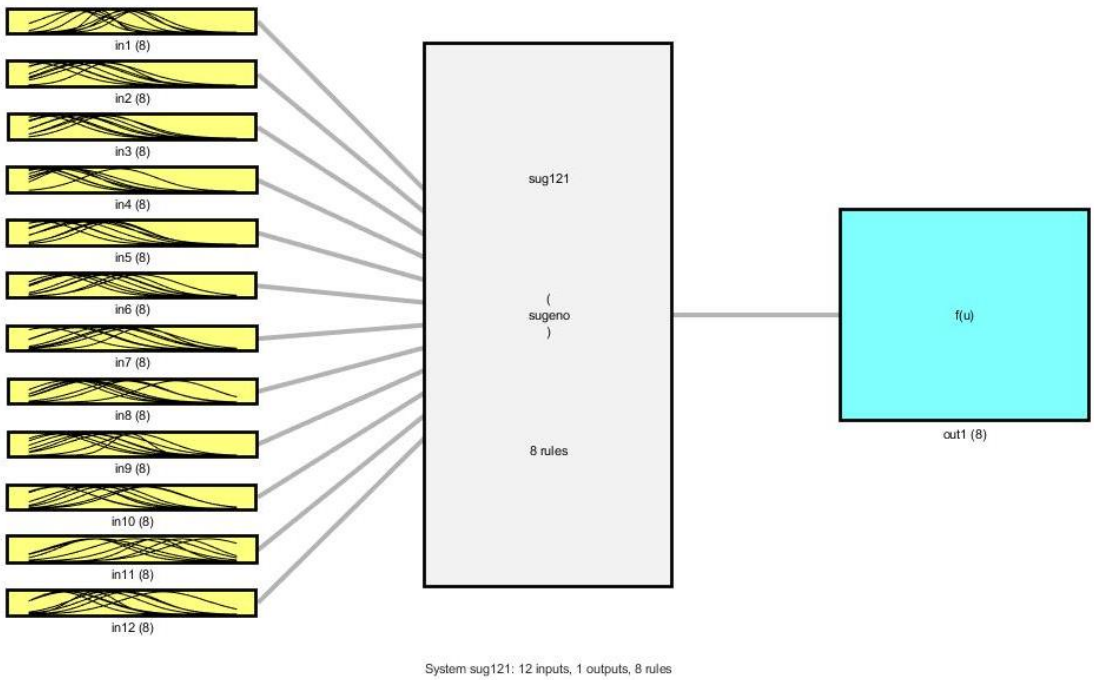
The next step is to create a FIS to be used further in creating ANFIS network. To do this we use genfis2 method with radius 0.5.

genfis2 uses subtractive clustering to automatically generate fuzzy rules from the input-output data. Subtractive clustering is a data-driven approach that identifies

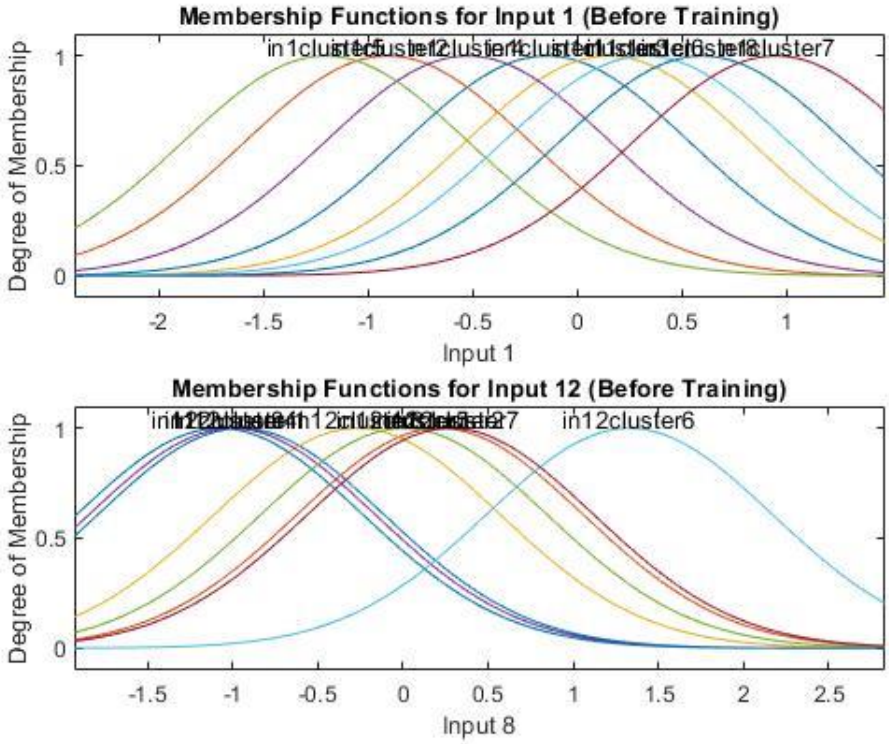
natural groupings (clusters) in the data. Each cluster corresponds to a fuzzy rule in the FIS. The algorithm identifies cluster centers in the input-output data space, where each cluster center represents a region where the data points are densely packed. Each cluster center is used to create a fuzzy rule, meaning that if there are n clusters, n fuzzy rules are generated. For each input variable, Gaussian membership functions are typically created around the cluster centers. These membership functions define how the input space is partitioned. The key advantage of subtractive clustering is that it automatically determines the number of rules and the structure of the FIS based on the data, without requiring prior knowledge of the system.

Each fuzzy region corresponds to a rule in the FIS. The input space is divided into overlapping fuzzy regions, each associated with a membership function. These regions are defined by the cluster centers and their influence ranges. The radius parameter in genfis2 controls the size of the influence range for each cluster. A smaller radius results in more clusters (and thus more rules), while a larger radius results in fewer clusters. This partitioning allows the FIS to model complex, nonlinear relationships between inputs and outputs.

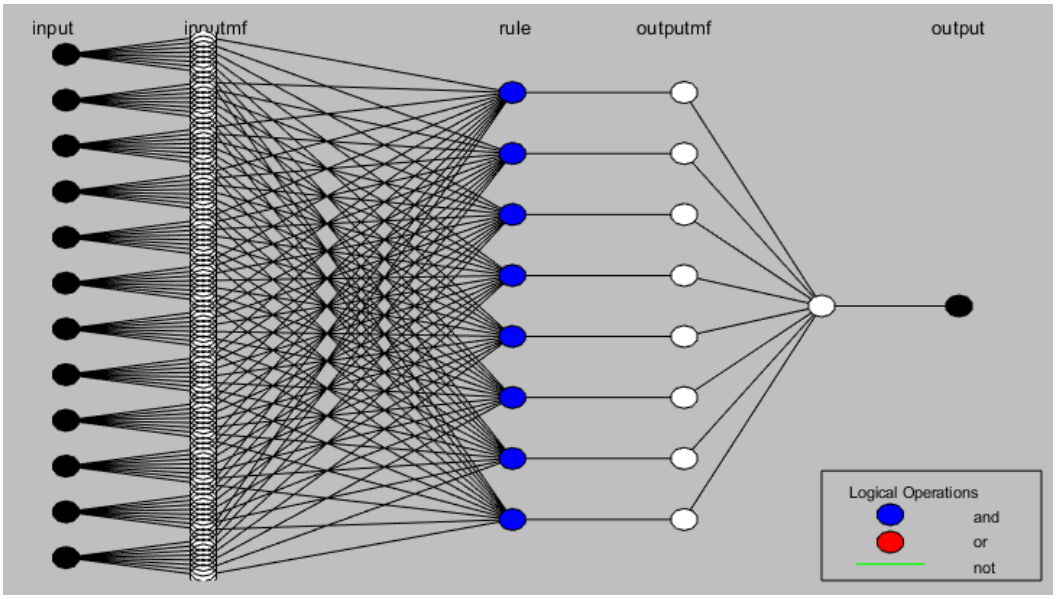
The structure of created fuzzy system would be like this. We have 12 inputs with Gaussian membership function and output membership function is constant.



The membership functions which are defined for input 1 and input 12 are brought in below pictures. Consider that these membership functions are generated by clustering of the data and not yet optimized. The optimization of parameters are dine in the training phase.

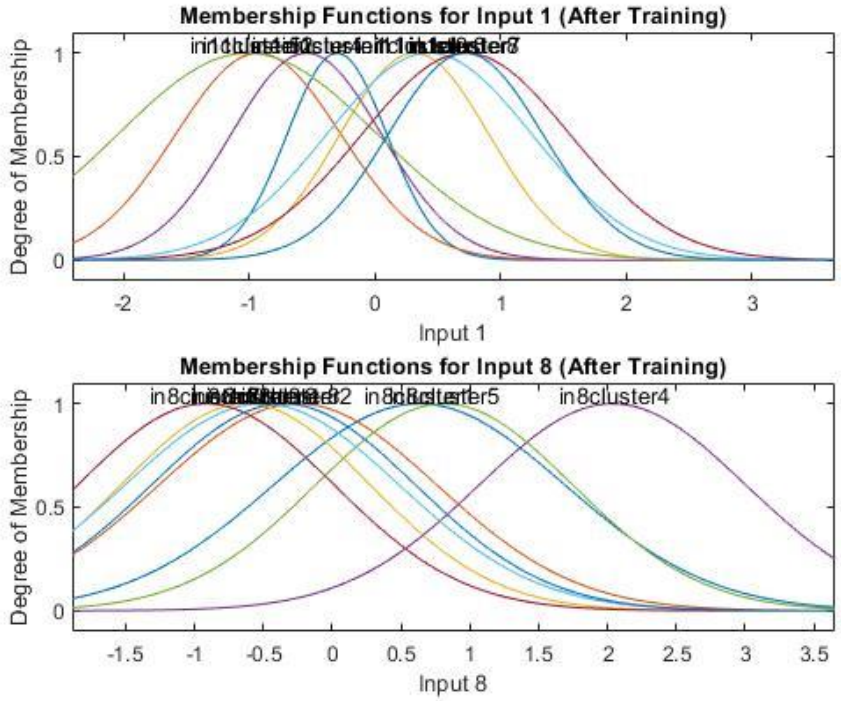


In the next step we create an ANFIS network. The structure of network is shown in the figure below:



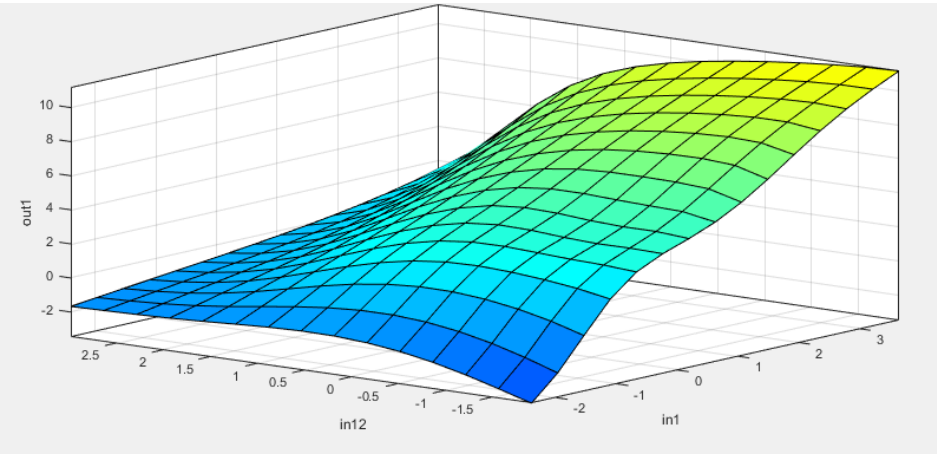
We perform training in 100 epochs and use both train data and validation data for checking the model not to be overfitted.

Now that training is performed we can check how the membership function of inputs are changed and how their center and width are optimized.



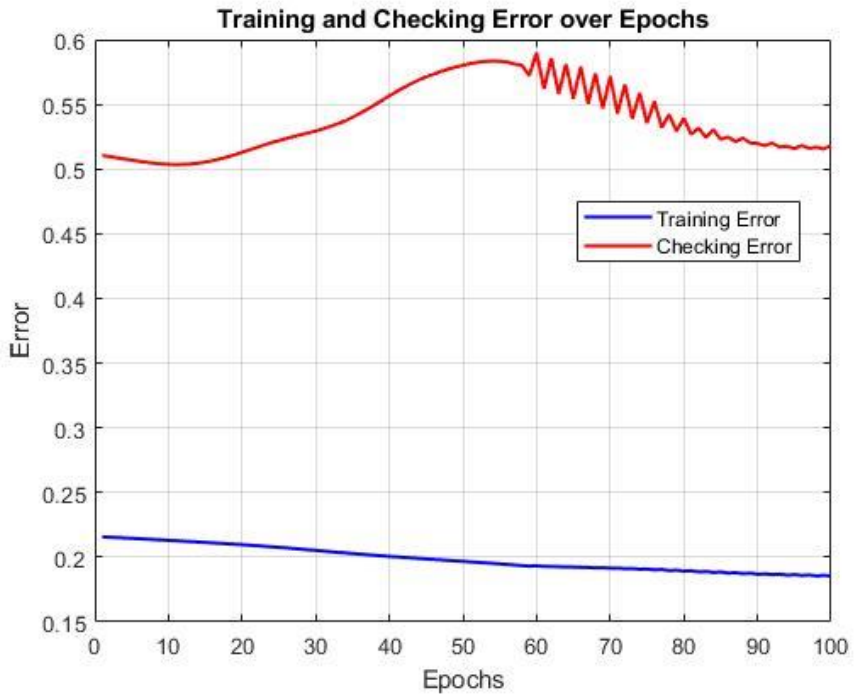
Comparing membership functions before and after training shows that they are effectively changed and their parameters optimized in a good way.

Now it’s not bad to check the surface which shows the relation of input 1 and 12 with output which is created based on the rules.



The surface shows that for high values of input1 and low values of input12 the output would be high and for high values of input12 and low values of input1 the output is low.

Here is the plot of error based on epochs for training and validation data:

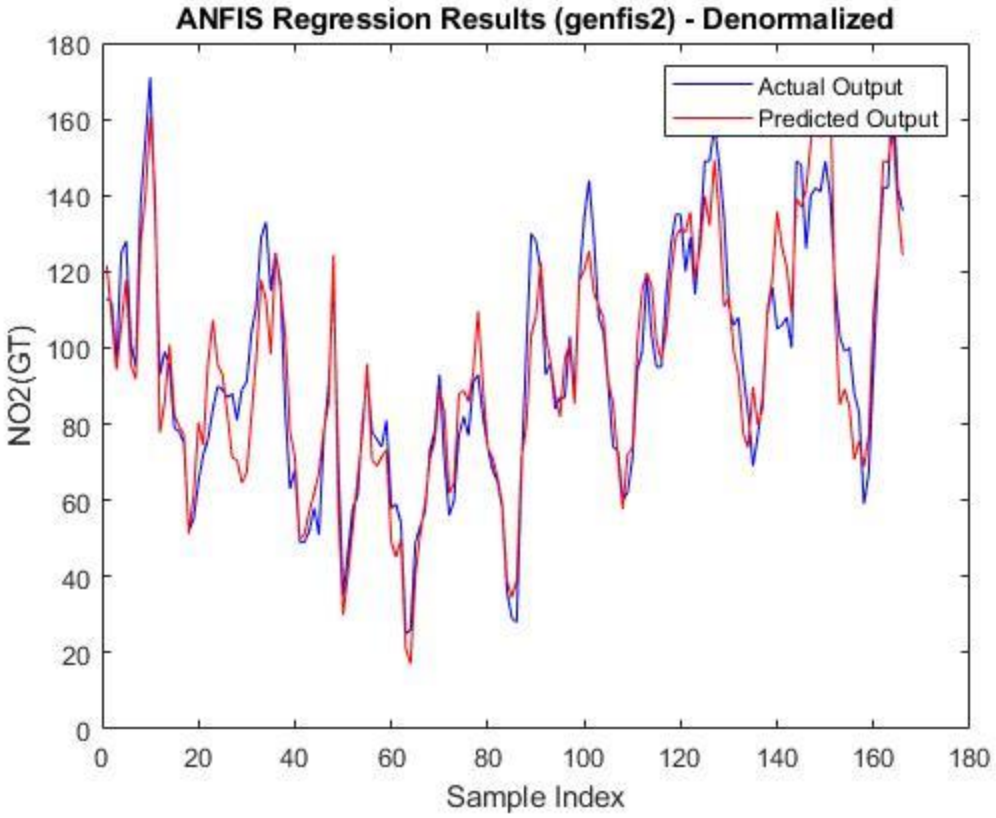


We also test the model with test data and the table below is summary of result of the network we trained.

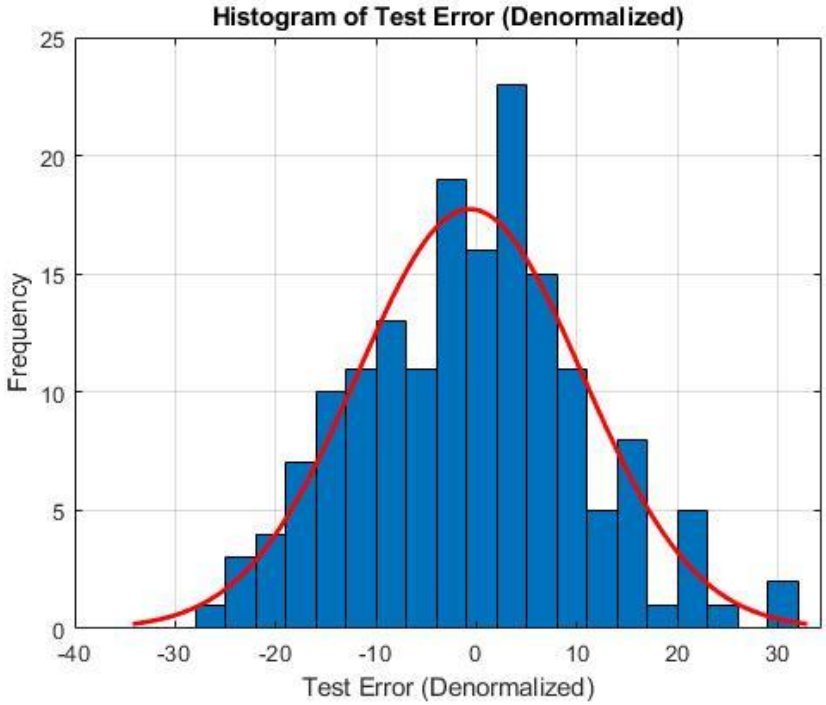
	train	validation	test
MSE	34.0797	266.2363	125.0327
RMSE	5.8378	16.3167	11.1818

The picture below is test result of the model and it show the true value and predicted value in same figure.

From this picture we can see that models performance is good and it does task of regression with a little value of error.



The histogram of error is like this too:



PBF:

The preprocessing here is same as what we did in ANFIS network.

For implementing neural network with RBF layer we use matlab and newrb function. We create a network with 20 neurons and spread of 5 for RBFs(It controls the width of the RBFs).

The newrb function in MATLAB is used to create and train a Radial Basis Function (RBF) neural network. The architecture of an RBF network consists of three layers: an input layer, a hidden layer with radial basis activation functions, and an output layer with linear activation functions. The key principle behind RBF networks is the use of radial basis functions in the hidden layer to transform the input data into a higher-dimensional space, where the problem becomes linearly separable or easier to model.

The input layer of an RBF network receives the input data and passes it to the hidden layer. The hidden layer consists of neurons that use radial basis functions, typically Gaussian functions, to compute their outputs. Each neuron in the hidden layer has a center and a width (spread) parameter, which determine the shape and range of the radial basis function. The Gaussian function computes the distance between the input vector and the neuron's center, and the output of the neuron is a function of this distance. This allows the hidden layer to capture local features of the input data, making RBF networks particularly effective for problems where the relationship between inputs and outputs is nonlinear.

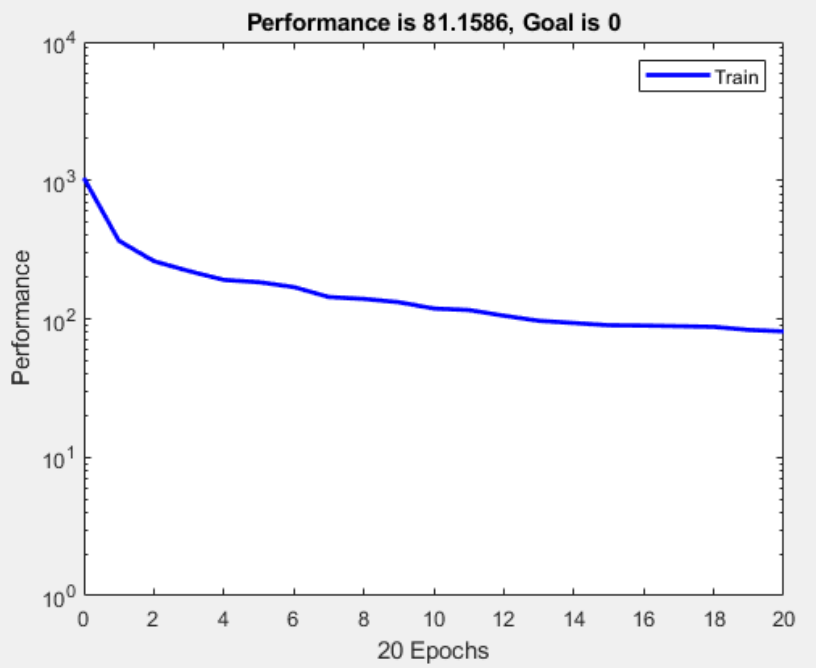
The newrb function in MATLAB automates the process of creating and training an RBF network. It takes the input data, target outputs, and other parameters such as the goal error and spread constant as inputs, and it returns a trained RBF network. The function uses an iterative process to add neurons to the hidden layer until the network's performance meets the specified goal error or the maximum number of neurons is reached.

The code below is for crating and training RBF network.

```
% Define the RBF neural network
% The RBF network will have one hidden layer with RBF neurons
% The output layer will be a linear layer

% Number of RBF neurons in the hidden layer
numRBFNeurons = 20;

% Create the RBF network
```

```
net = newrb(X_train', Y_train', 0, 5, numRBFNeurons, 1);

% Train the network (RBF networks are trained in one step, no further
training is needed)

% Evaluate the network on the validation set
Y_val_pred = sim(net, X_val');

% Calculate the Mean Squared Error (MSE) on the validation set
mse_val = mean((Y_val' - Y_val_pred).^2);
fprintf('Validation MSE: %f\n', mse_val);

% Evaluate the network on the test set
Y_test_pred = sim(net, X_test');

% Calculate the Mean Squared Error (MSE) on the test set
mse_test = mean((Y_test' - Y_test_pred).^2);
fprintf('Test MSE: %f\n', mse_test);
```

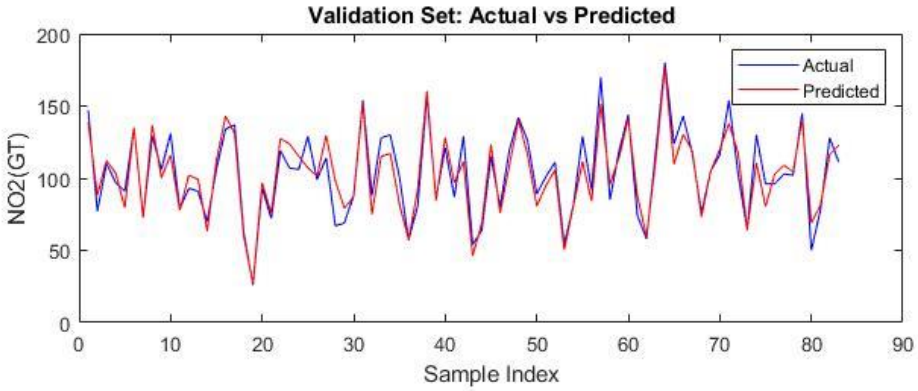
plot of train set error based on epoch is shown in the figure below.

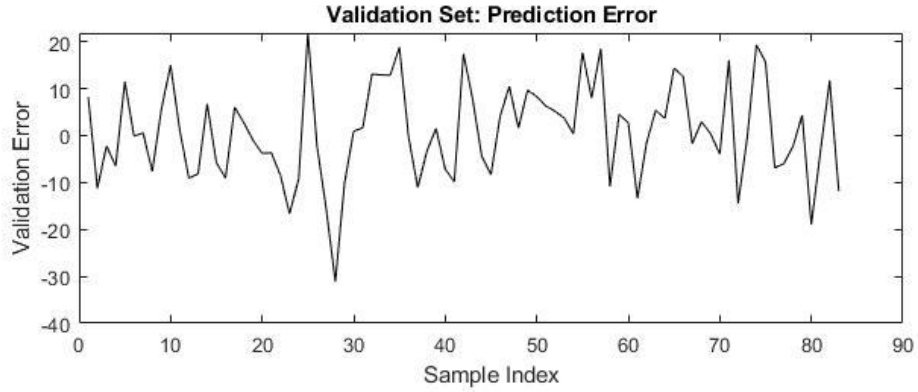


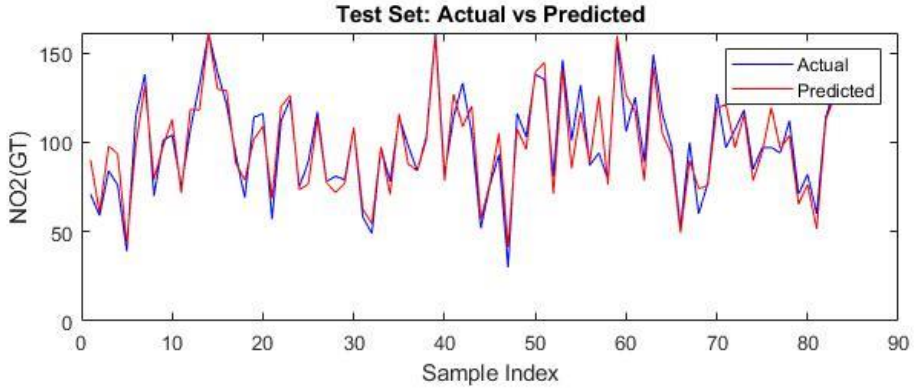
We test this model and table below is the summary of this networks results:

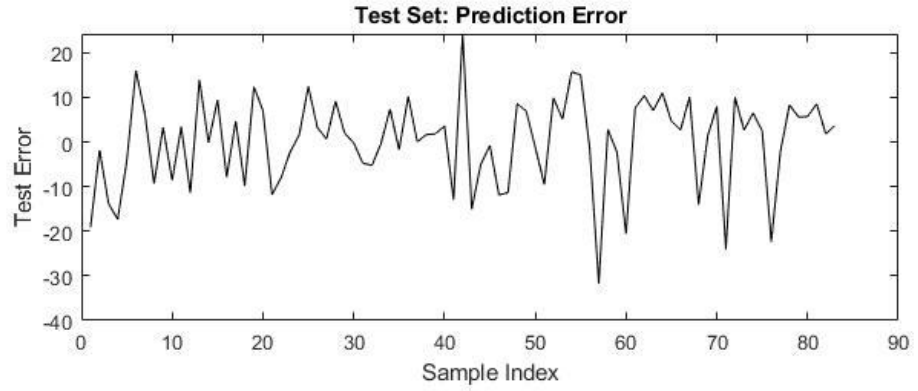
	train	validation	test
MSE	81.1586	89.218663	109.890980
RMSE	9.0088	9.4456	10.4829

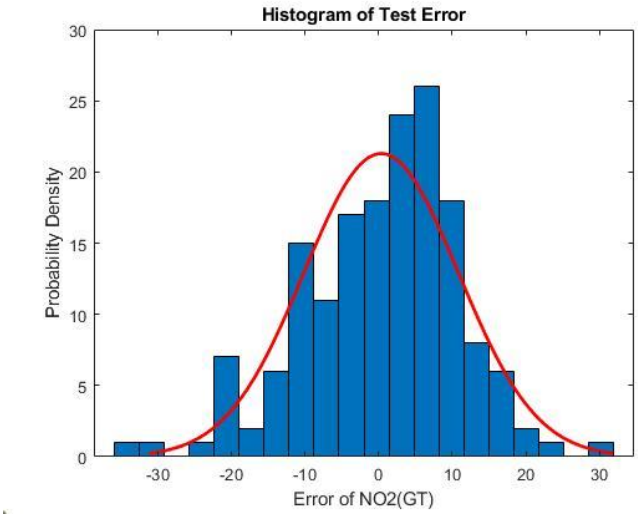
The figures below are result of evaluation of the model on validation and test datasets.


Validation Set: Actual vs Predicted
Actual
Predicted
NO2(GT)
Sample Index


Validation Set: Prediction Error
Validation Error
Sample Index


Test Set: Actual vs Predicted
Actual
Predicted
NO2(GT)
Sample Index


Test Set: Prediction Error
Test Error
Sample Index



Conclusion:

The result of both models on test dataset is almost same and the RMSE of this two network has a little difference but the RBF network has better performance.

The superior performance of the Radial Basis Function (RBF) network over the Adaptive Neuro-Fuzzy Inference System (ANFIS) in this regression task can be attributed to the inherent characteristics and architectural differences between the two models. RBF networks excel in capturing local patterns in data due to their use of radial basis functions in the hidden layer, which are particularly effective for modeling nonlinear relationships. The Gaussian activation functions in the hidden layer allow the RBF network to focus on specific regions of the input space, making it highly adaptable to complex, localized variations in the data. In contrast, ANFIS combines fuzzy logic and neural networks to model global relationships, which may not be as effective when the data exhibits strong local nonlinearities. The RBF network's ability to approximate functions with fewer parameters and its straightforward training process, which involves determining centers and widths followed by linear optimization of output weights, often leads to faster convergence and better generalization on datasets with intricate patterns.