PRACTICE. NEURO-FUZZY LOGIC SYSTEMS

MATLAB TOOLBOX GUI

Dmitry Bystrov, Jerker Westin

Practice "Neuro-Fuzzy Logic Systems" are based on Heikki Koivo "Neuro Computing. Matlab Toolbox GUI" .

Simulink for beginners section gives introduction to Matlab Toolbox, present users GUI for Matlab command window and Simulink.

Fuzzy basics section describes the basic definitions of fuzzy set theory, i.e., the basic notions, the properties of fuzzy sets and operations on fuzzy sets. Are presented examples, exercises and laboratory works.

Neural networks basics section gives introduction to neural networks. Are described the basic theoretical concepts. Backpropagation and radial basis function networks are reviewed with details. Are presented examples and laboratory works.

ANFIS (Adaptive Neuro-Fuzzy Inference System) basic concepts are given in finally section.
Are reviewed GENFIS1 and ANFIS commands, is presented exercise.

Contents

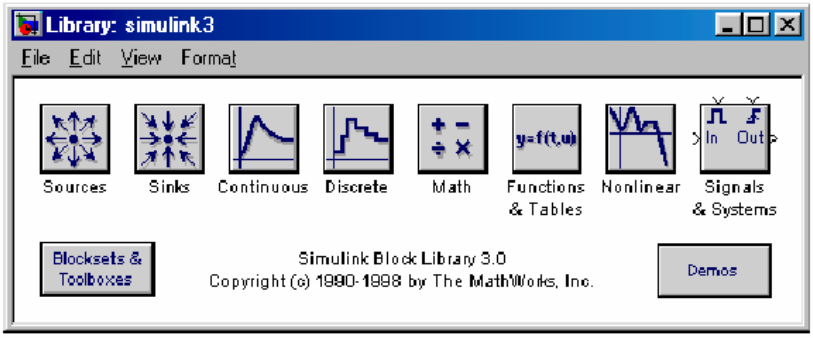
Chapter 1

Simulink for Beginners :

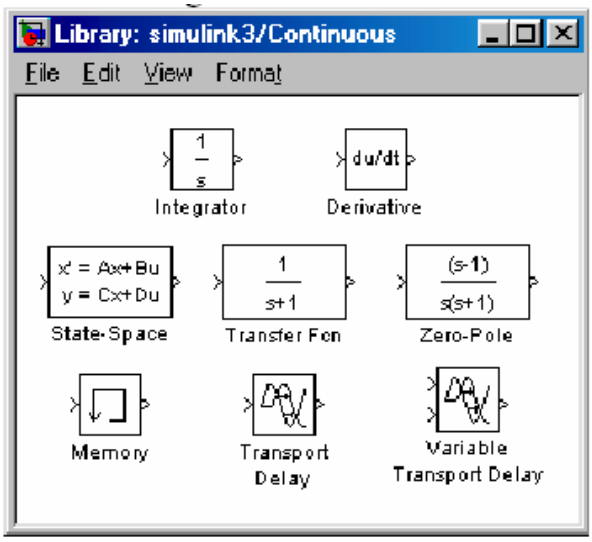
- To begin your SIMULINK session open first MATLAB ICON by clicking mouse twice and then type

»simulink3

You will now see the Simulink block library

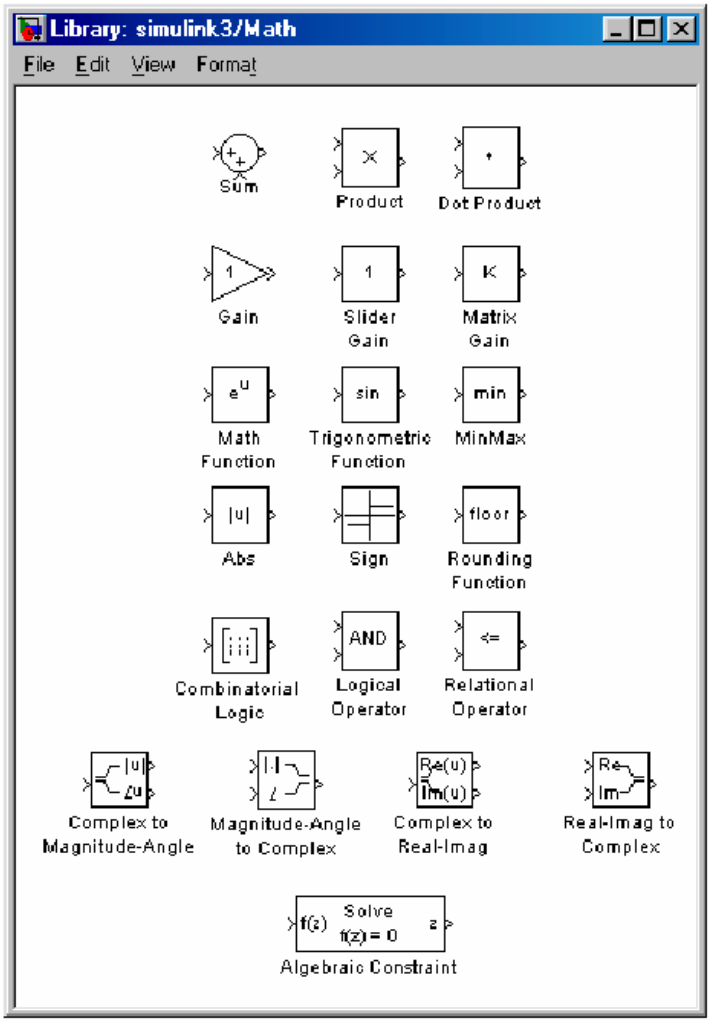


- Browse through block libraries. E.g., if you click Continuous, you will see the following:



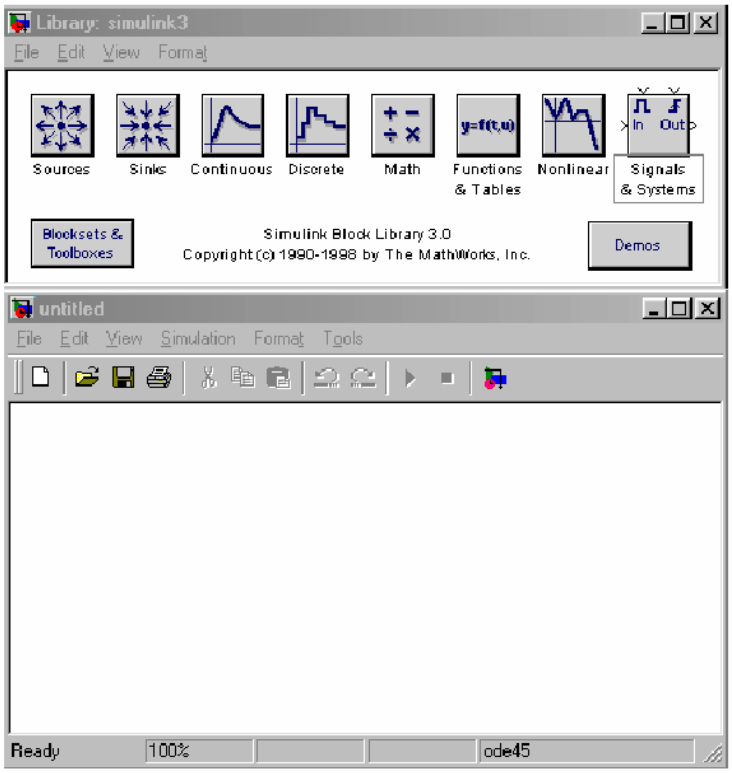
Observe the description of the integrator at the bottom and also the block that will be used for integrator, $\frac{1}{s}$. This operator notation comes from linear systems domain, where s is Laplace variable. Roughly, s corresponds to derivative operator, $\frac{d}{dt}$, and its inverse $\frac{1}{s}$ to integration, $\int$.

- Mathematics block library is shown below



After browsing through other block libraries, we are now ready to start generating a simple Simulink diagram.

- Choose in menu selection File, then New and Model. This file will be called **untitled**.



Untitled file, where the SIMULINK configuration is constructed using different blocks in the library. With a mouse you can configure your model into the empty space.

1.1. MATLAB Command window

Once you have defined your system in SIMULINK window, you can simulate also on the MATLAB Command window. Save your model – currently it has the name *Untitled*, so use that. Go to MATLAB command window and type » `help sim`. The following lines will help you to understand how to simulate.

```
» help sim
```

SIM Simulate a Simulink model
SIM('model') will simulate your Simulink model using all simulation parameter dialog settings including Workspace I/O options.

The SIM command also takes the following parameters. By default time, state, and output are saved to the specified left hand side arguments unless OPTIONS overrides this. If there are no left hand side arguments, then the simulation parameters dialog Workspace I/O settings are used to specify what data to log.

[T,X,Y] = SIM('model',TIMESPAN,OPTIONS,UT)
[T,X,Y1,...,Yn] = SIM('model',TIMESPAN,OPTIONS,UT)

T : Returned time vector.
X : Returned state in matrix or structure format. The state matrix contains continuous states followed by discrete states.

Y : Returned output in matrix or structure format.
For block diagram models this contains all root-level outport blocks.
Y1,...,Yn : Can only be specified for block diagram models, where n must be the number of root-level outport blocks. Each outport will be returned in the Y1,...,Yn variables.

'model' : Name of a block diagram model.
TIMESPAN : One of:
TFinal,
[TStart TFinal], or
[TStart OutputTimes TFinal].
OutputTimes are time points which will be returned in T, but in general T will include additional time points.
OPTIONS : Optional simulation parameters. This is a structure created with SIMSET using name value pairs.
UT : Optional extern input. UT = [T, U1, ... Un] where T = [t1, ..., tm]' or UT is a string containing a function u=UT(t) evaluated at each time step. For table inputs, the input to the model is interpolated from UT.

Specifying any right hand side argument to SIM as the empty matrix, [], will cause the default for the argument to be used.

Only the first parameter is required. All defaults will be taken from the block diagram, including unspecified options. Any optional arguments specified will override the settings in the block diagram.

See also SLDEBUG, SIMSET.

Overloaded methods

```
help network/sim.m
```

The simplest way to start is to use sim command in the following way.

```
» [t,x,y]=sim('untitled');
```

Next you can use plot command to see the result

```
» plot(t,x)
```

Chapter 2

Fuzzy Basics

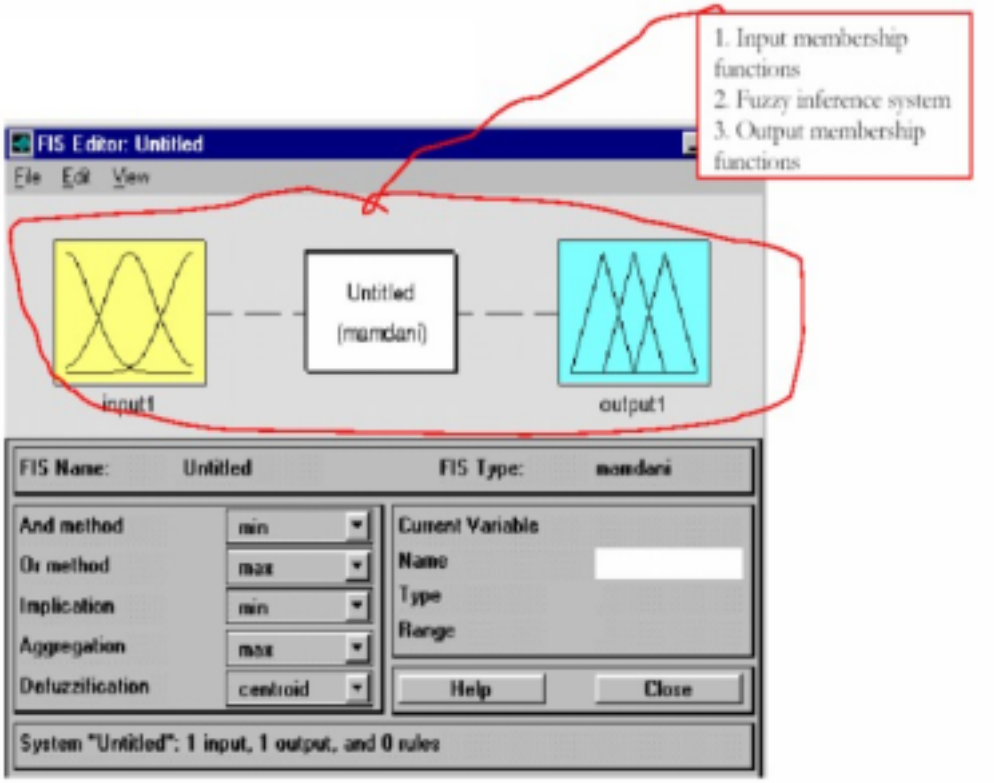
2.1. EXAMPLE : Let the room temperature **T** be a fuzzy variable. Characterize it with three different (fuzzy) temperatures: *cold,warm, hot*.

SOLUTION:

1. First define the temperature range, e.g. [0^0,40^0]

2. When MATLAB is open, then open *GUI (GUI =Graphical User Interface)* by typing ***fuzzy***

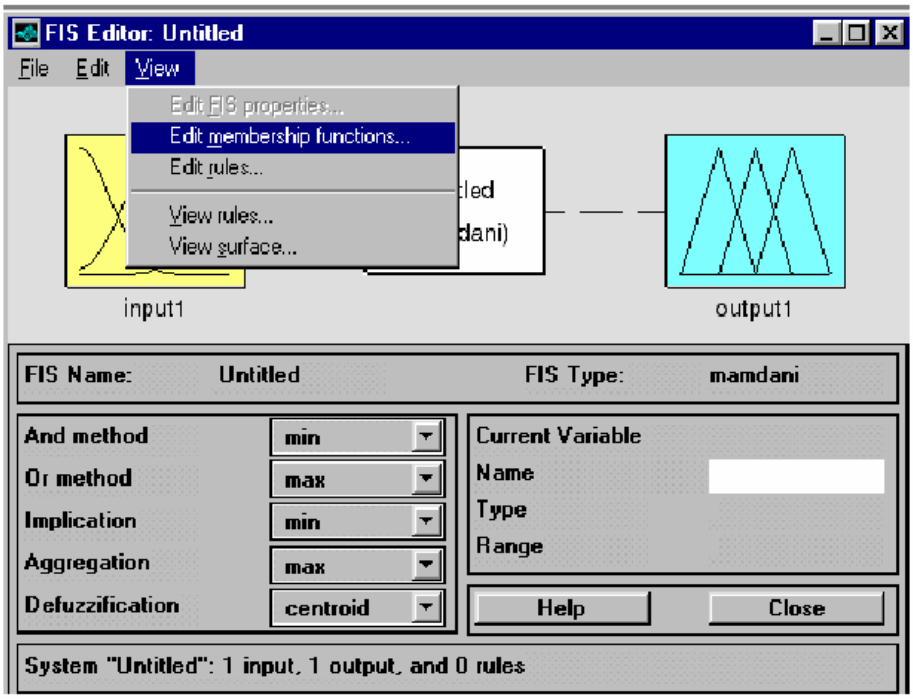
The result is shown below:



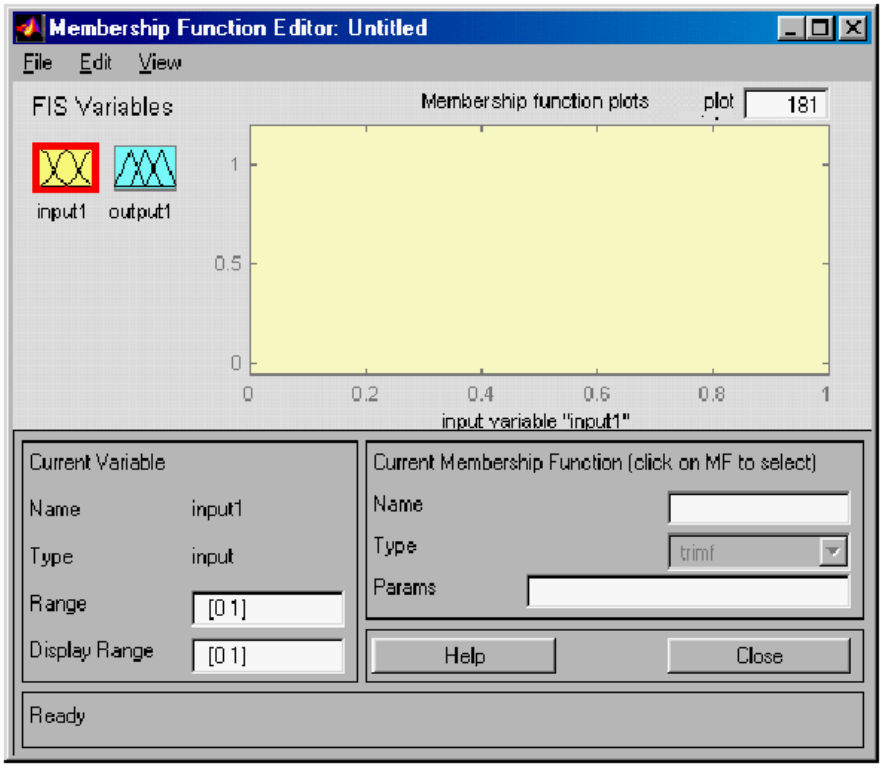
GUI of Fuzzy toolbox. Study the content of GUI to have an overall understanding before proceeding.

Next activate input membership block by moving mouse on top of it and by clicking once. Activation is shown by a *red boundary* on the block. If you click twice, the **MEMBERSHIP FUNCTION EDITOR** opens up. This is needed when defining membership functions.

Another way to do the same is to use ***View*** menu (cf. Figure below). Input block is again activated first.



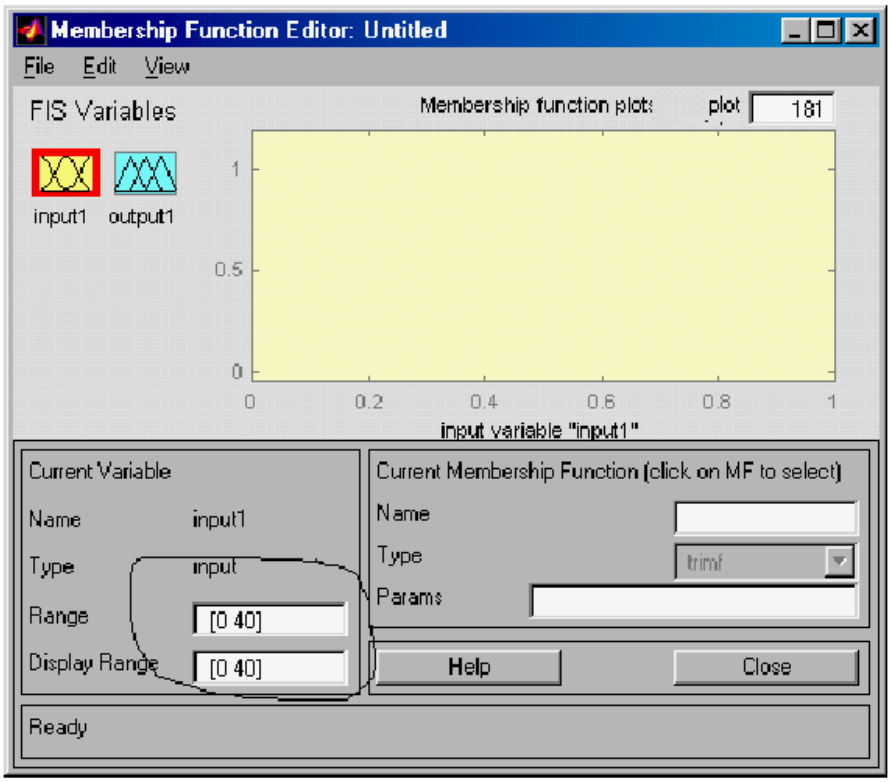
Editing membership functions from ***View*** menu



Display to edit input membership functions.

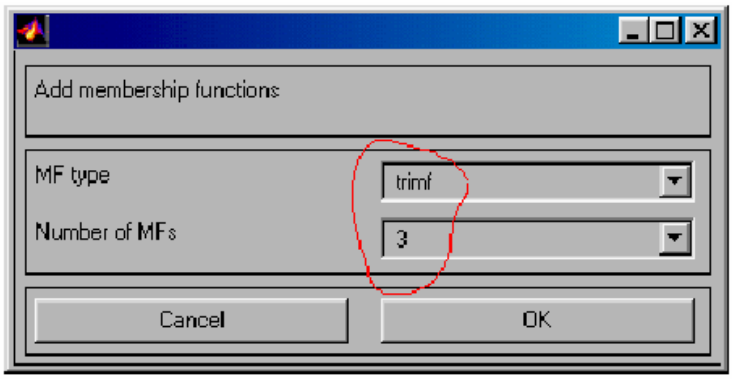
First change *Range* to the one given in the problem or to $[0^0, 40^0\ ]$. This is done by moving the cursor to the *Range* area and clicking once. Then you can correct the figures in the range domain as you would correct text in a text document. Note that you have to leave space between the numbers.

Next click *Close* or anywhere in the light grey area above with the mouse. The result is shown below.



Change the range of input variable to $[0^0, 40^0]$.

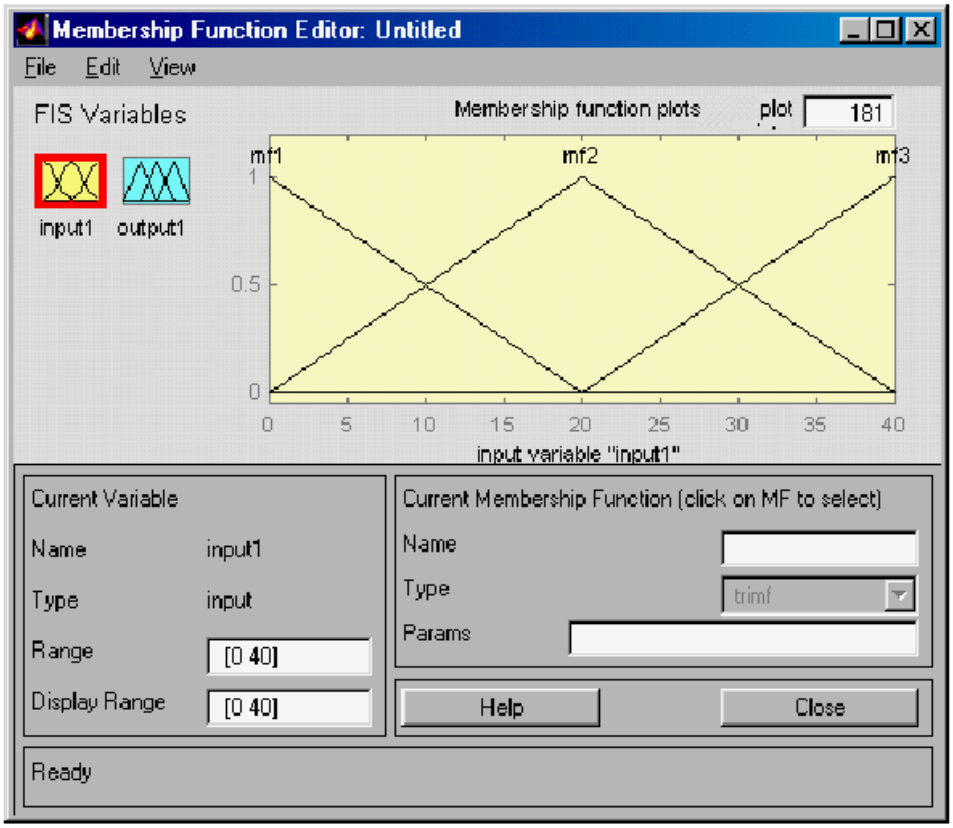
Now you are ready to define new membership functions. Choose ***Add MF's*** from EDIT menu. The following display is shown.



Type and number of membership functions can be chosen.
Default value is three triangular (*trimf*) (3) membership functions as seen on the display. These will divide the range into three equal parts.

Click OK, if you are happy with the default values, otherwise make your own choice. (Try e.g. five *Gaussian* membership functions. Different membership functions can be found under *MF type*.)

The result is shown below. Note that the default names for the membership functions are *mf1, mf2* and *mf3*.



The result of choosing three, triangular membership functions, which by default are named *mf1, mf2* and *mf3*.

2. 2. Fuzzy Logical Operations

T-norm

Triangular norm

Mapping $T:[0,1]\times[0,1]\to[0,1]$ is called **T-norm if** it satisfies the following criteria:

$T(a,b)=T(b,a)$ **commutativity**

$T(T(a,b),c)=T(a,T(b,c))$ **associativity**

$a\le c\wedge b\le d=T(a,b)\le T9c,d)$ **nondecreasing**

$T(a,1)=T(1,a)=a, T(0,0)=0$ **boundary**

REMARK: In fuzzy control *T-norm* is used to connect different propositions in connection of ***and***-operation. The arguments are then the corresponding membership functions.

The most common T-norms are

1. Minimum $\min(a,b)$

2. Product ba

T-conorm

Triangular conorm

Mapping $T^*:[0,1]\times[0,1]\to[0,1]$ is called **T-conorm, if** it is T-norm and in addition satisfies

$$T^*(a,b)=T^*(0,a)=a, T^*(1,1)=1$$

T-conorm is generally used to connect propositions in connection of *or*-operation.
The most common T-conorms:

1. Maximum $\max(a,b)$
2. In probability theory $a+b-ab$
3. For triangular membership functions sum $(a+b)$ is used (although it does not satisfy the above conditions)

2.3. Fuzzy Reasoning

1. **Generalized Modus Ponens (GMP)** (forward chaining)
2. Generalized Modus Tollens (GMT) (backward chaining)

(GMP)

Fact 1: x is $\overline{A}$

Premise 2: If x is A then y is B

Conclusion: y is $\overline{B}$

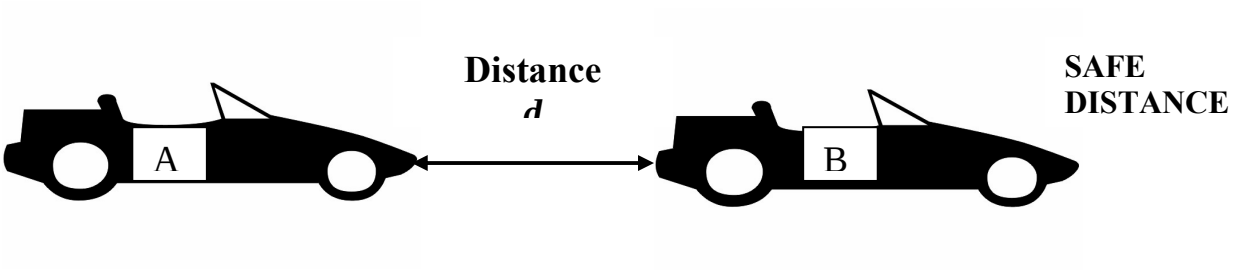
(GMT)

Fact 1: y is $\overline{B}$

Premise 2: If x is A then y is B

Conclusion: x is $\overline{A}$

2.3.1. Example : *Car driving*



Cars A and B on a highway

You are driving car A on a highway. You want to keep a **safe distance** to car B in front of you. Design a (simplified) fuzzy-logic system, which satisfies the requirements.

Proceed as follows:

a. Determine the required fuzzy variables (input/output) and their ranges.
b. Form the rule base.
c. Use fuzzy reasoning to check the operability of the rulebase.

SOLUTION:

a. **Fuzzy variables**. Start with a simple case

INPUT: Distance d

OUTPUT: Breaking power b *(*gas pedal*)*

Three (3) membership functions are chosen for both input and output

Membership functions for INPUT:

Distance d (meters): **short, medium, long**

Membership functions for OUTPUT:

Breaking power b (%): **large, medium, none**

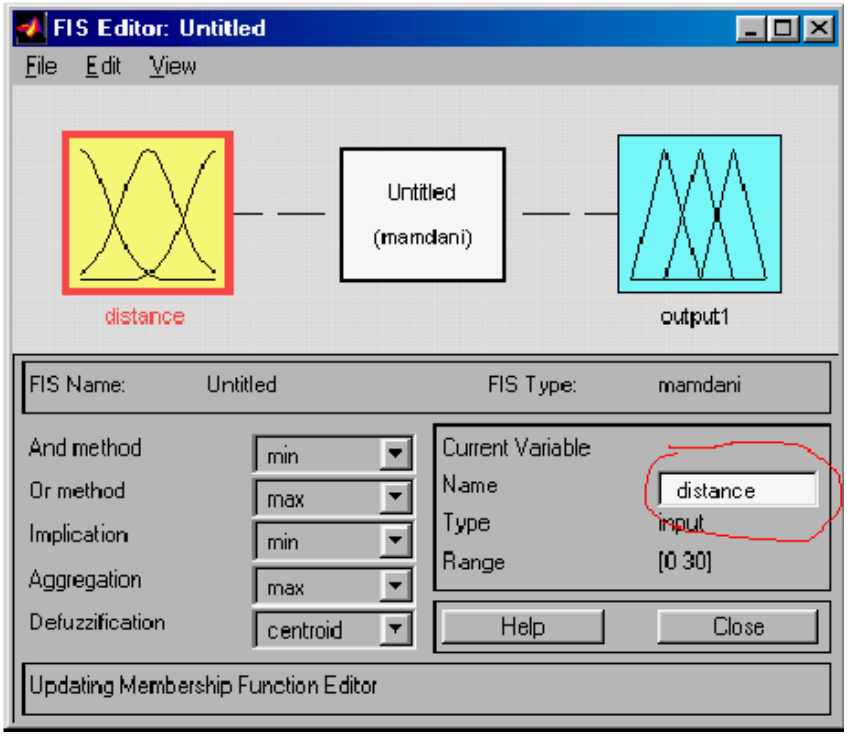
We will use the fuzzy toolbox to define the fuzzy system by giving numerical values for the variables indicated

In MATLAB type

» fuzzy

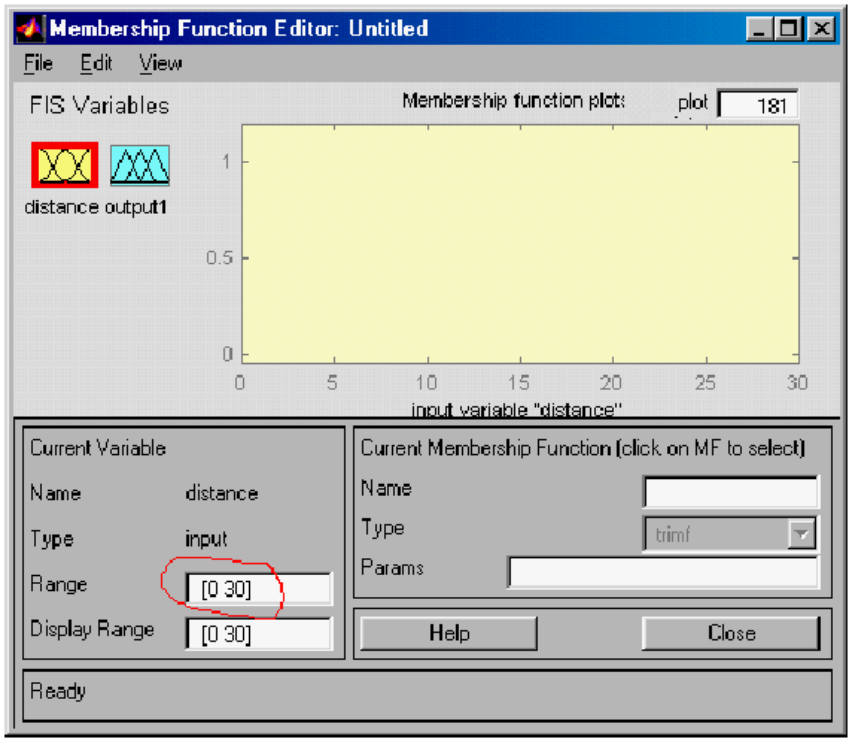
This opens the GUI. Activate the input window

Give a name to the fuzzy input variable. Call it *distance*. Click *Close*.



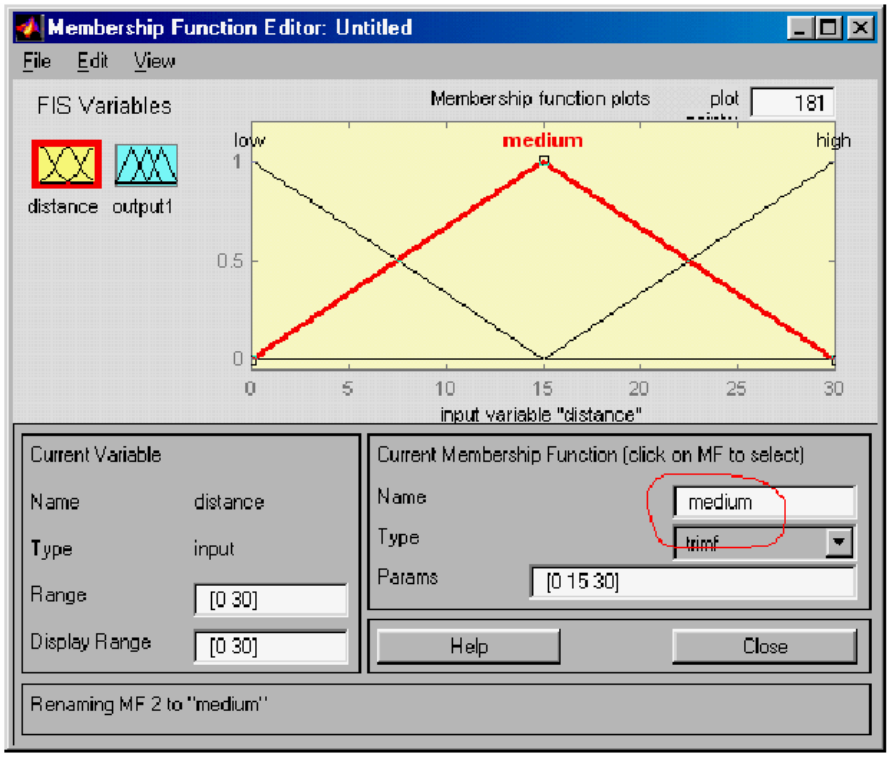
Naming the input variable in GUI.

Next click the input block twice with mouse to open the membership function window. First define the range, say from 0 to 30 m/s.



Set the range in GUI

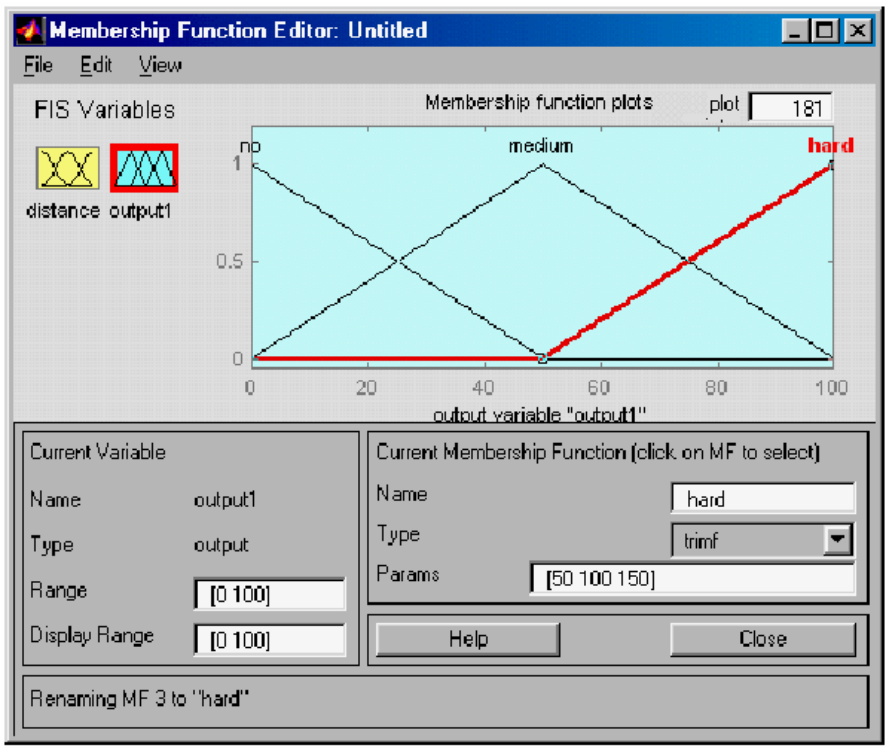
Next choose from Edit ***Add MFs***. Pick the default values: 3 triangular membership functions. Give a name to each: Call them *high*, *medium*, and *short*. When you are finished, click *Close*.



Three triangular membership functions have been chosen for the input variable *distance*. The middle one has been activated and renamed as *medium*.

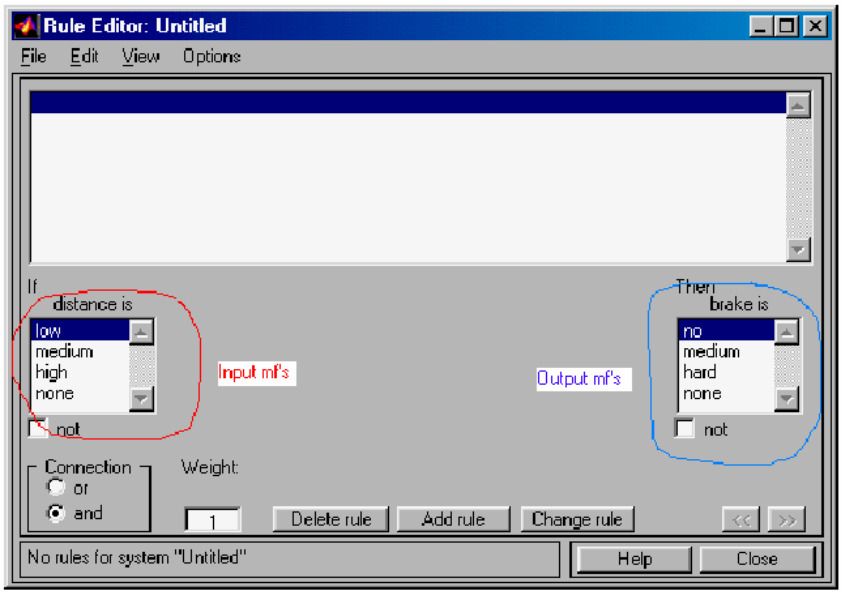
Repeat the same procedure with the output *b*, breaking power.

Define the name of the output, break, and its range. Use three membership functions: ***hard, medium*** and ***no.*** The following GUI display is obtained.



Three triangular membership functions are chosen for the output variable breaking. The last one has been activated and renamed as *hard.*

What is missing from the fuzzy system now is the rule base. Open *View* menu and click *Edit rules*. Then the following display opens.



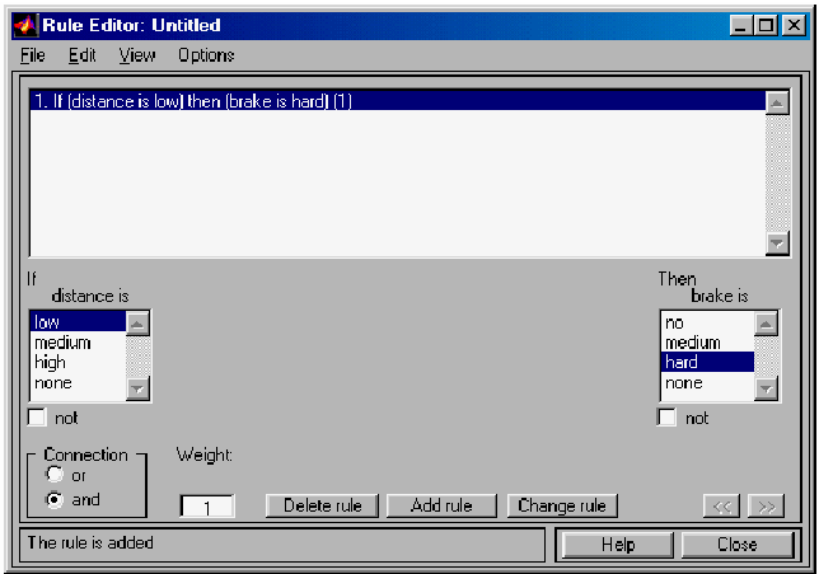
Rule Editor display. On the right, the input side. On the left, the output side

The left-hand side contains the membership functions of the input, *distance*. The right-hand side has the membership functions of the output, *brake*. If the input
side has several variables, which are connected either by *and* or *or,* the *Connection* block is in the lower left-hand corner. In this case we only have one input variable, so the connective is not used. The weight factor (default value = 1), indicates the importance of the rule in question.

The construction of the rule base is the hardest part of the design task. Here a simple-minded rule base is constructed based on driving experience. Typical rule is

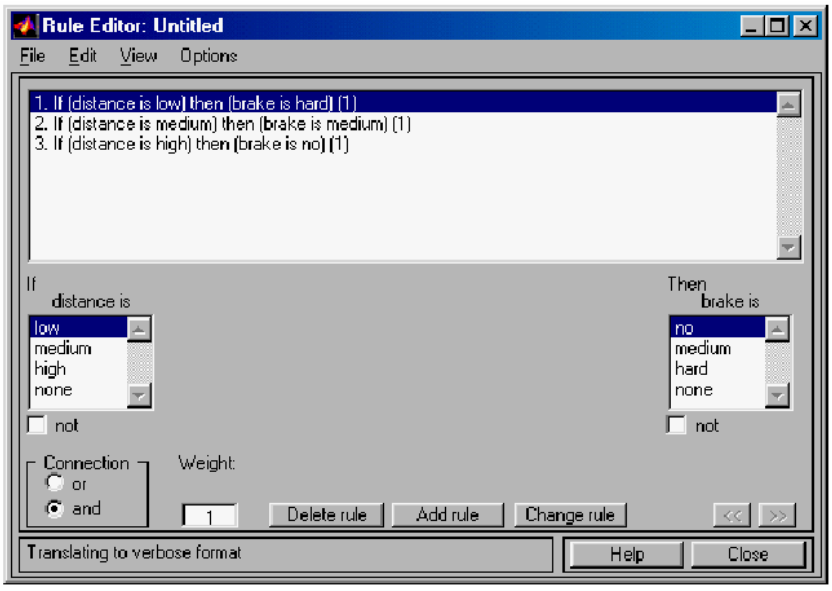
If distance is low, then brake is hard

With mouse choose the membership function *low* for the distance and *hard* for brake. This is done in the figure above. Then click ***Add rule***. The result is seen below.



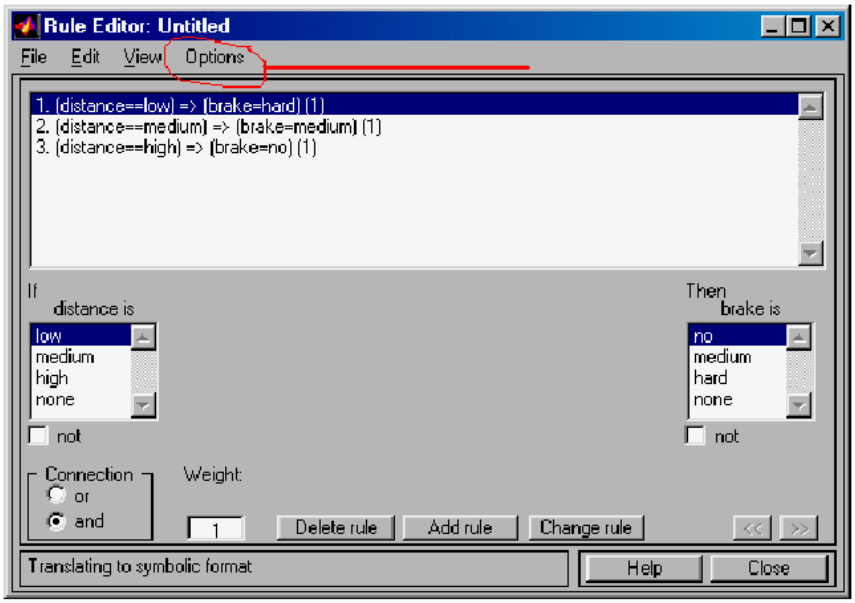
Setting up a rule with Rule Editor

Let us set two other rules, one for *medium* distance and the other for *long* distance. Our simple, rule base is now complete. Click *Close*.



Complete rule base of three rules. Note the weighting parameter (1) at the end of each rule, i.e., all rules have the same weighting.

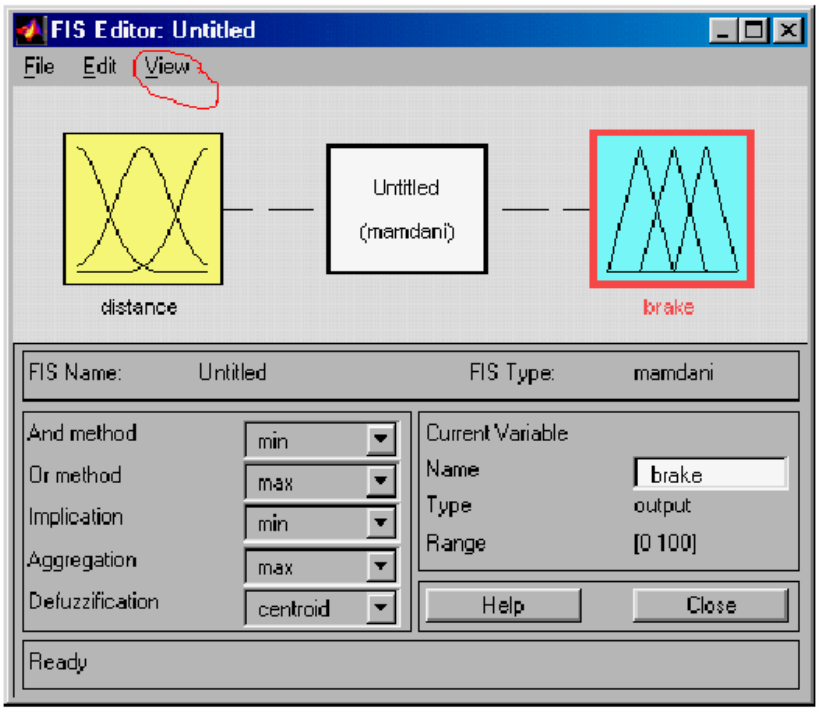
Now the design of the fuzzy system is complete. The Toolbox provides two more interesting ways expressing the rule base.



The rule base can be expressed in two other ways in GUI. Here verbose format is shown.

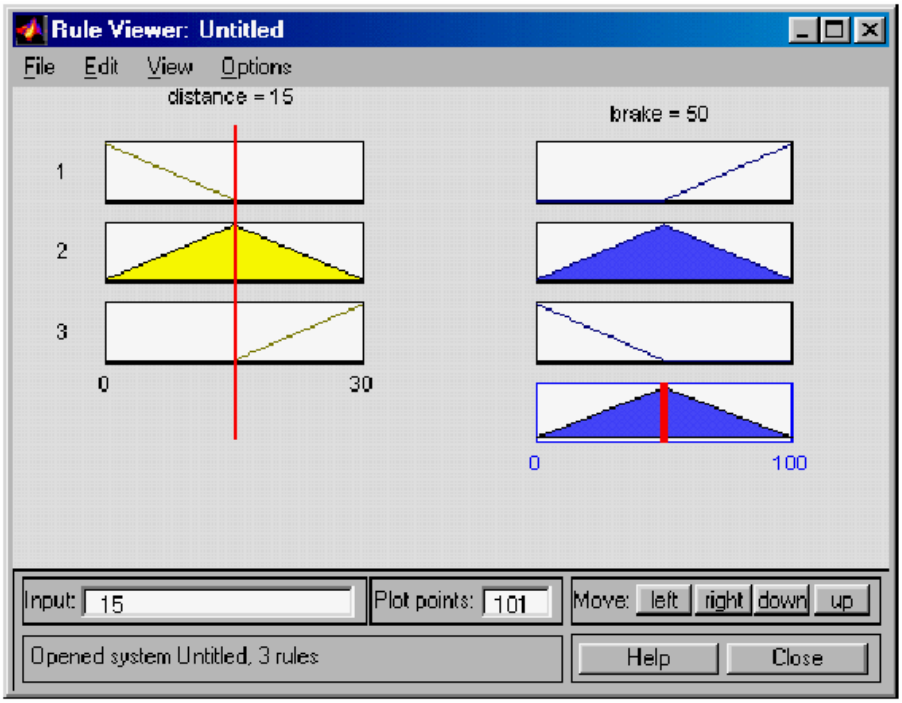
Check under *Options* and there choose *Format*. Under that you can see that the rules as shown are given *verbose*. The other forms are *symbolic* and *indexed*. Check in what form the rule base is given in each case.

Viewing rules gives you the overall picture of the developed fuzzy system. From the main FIS editor, choose from *View* menu *View rules.*



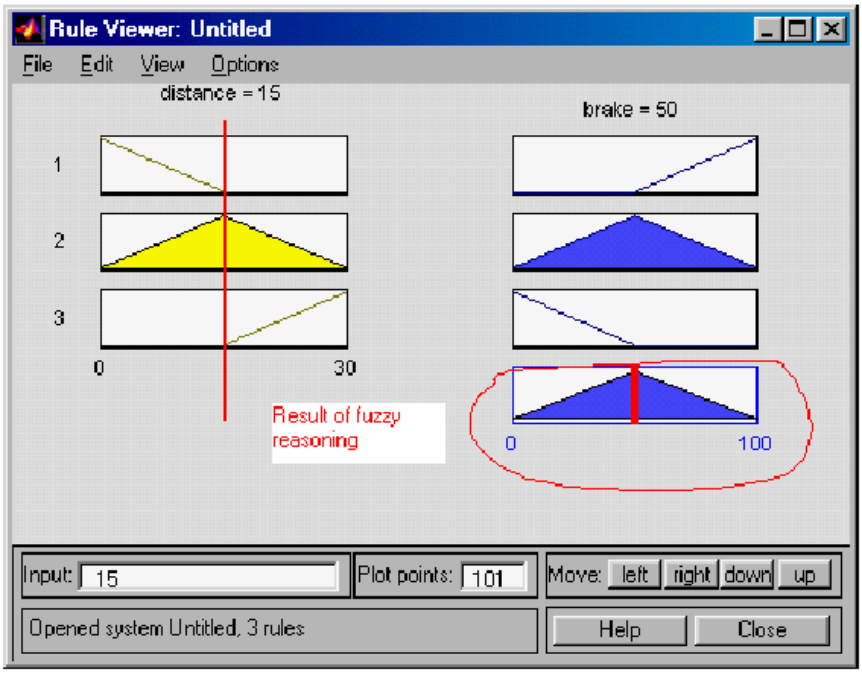
By choosing *View* the rule base can be viewed differently.

The display *View rules* is shown below.



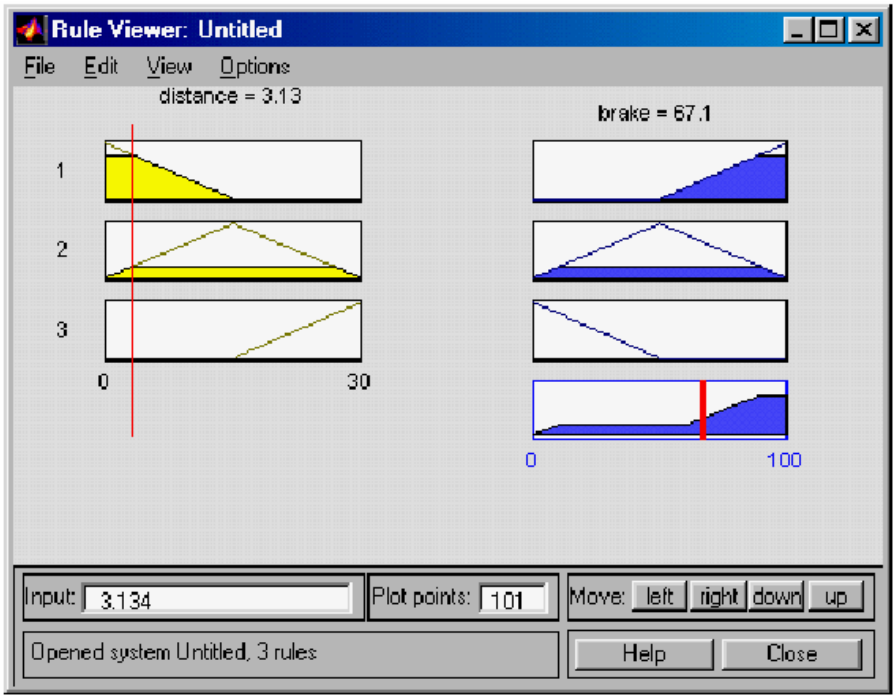
The rule base can be displayed graphically. The red line on the left indicates the value of the input, 15 m. Similarly the bar on the right hand side, indicates the output value, 50%.

On the left-hand side you can see the input, *distance* side and on the left the output, *brake* side. There are three rules and the corresponding triangular membership functions displayed. In the right-hand side lower corner is the result of fuzzy reasoning. At this point it is a fuzzy set. Applying defuzzification method, in the figure center of gravity has been chosen, a crisp value is obtained.



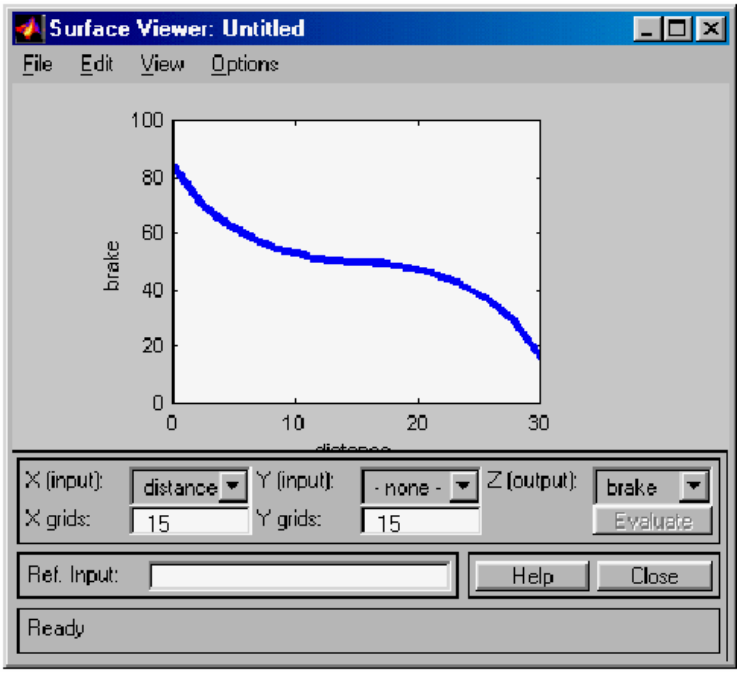
Result of fuzzy reasoning is brake = 50%

Different input values can be tried by moving the red, vertical line on the left-hand side on next Figure.



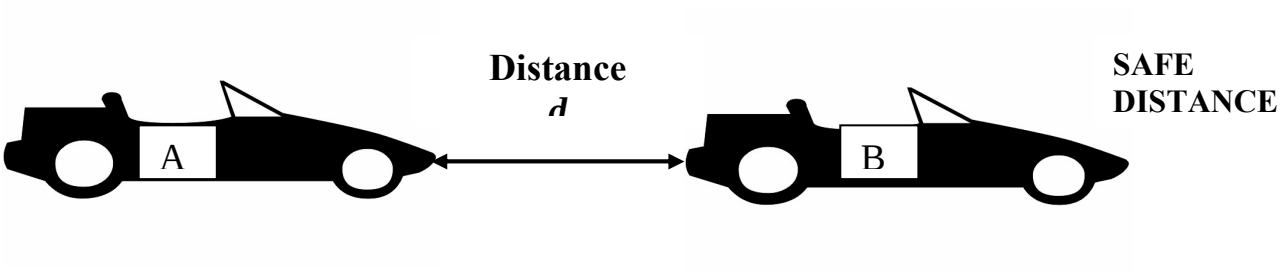
Changing the input value results in different output values.

Finally, the input-output mapping can be observed by viewing surface. Choose *View* menu and under it *View surface.* It is clear that our map is nonlinear. This is where the power of fuzzy systems is strong on next Figure



The fuzzy system viewed as input-output mapping

What is missing ?

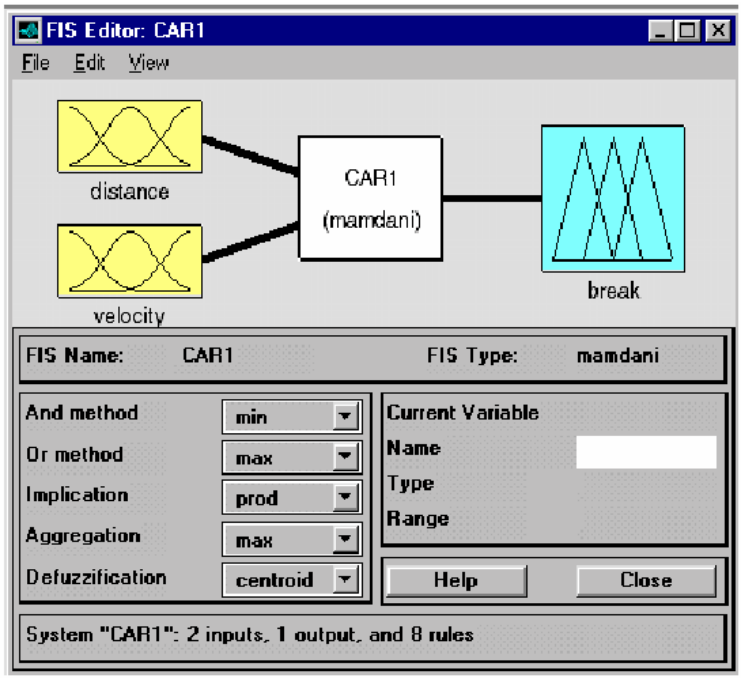


SPEED!
(relative speed between vehicles A and B).

SOLUTION:

Set up a new rule base with two inputs distance and speed, one output, braking power.

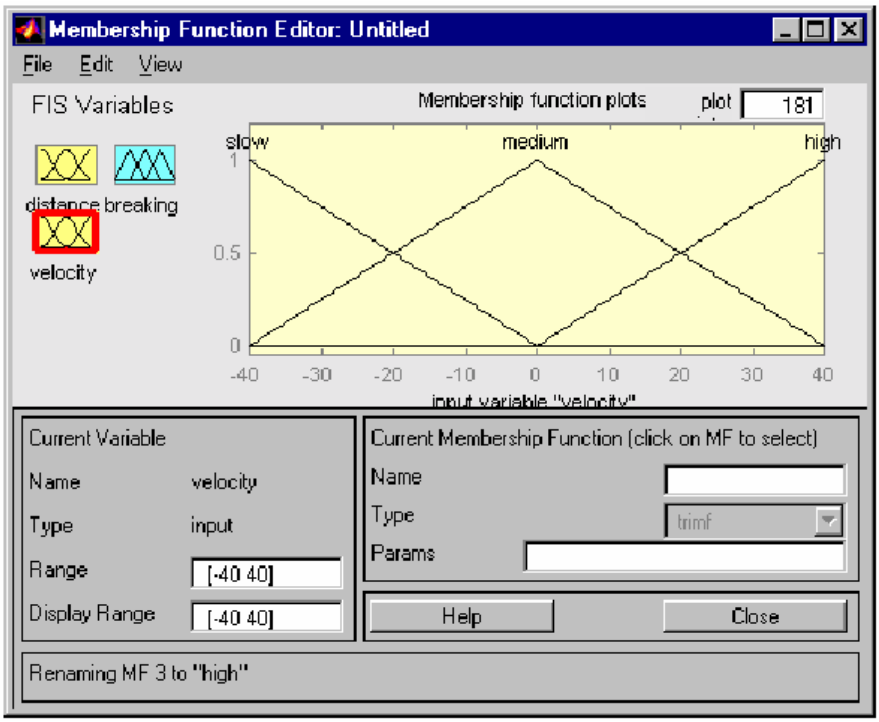
Here only the main steps are shown.



Fuzzy system with two inputs, distance and velocity, and one output, break

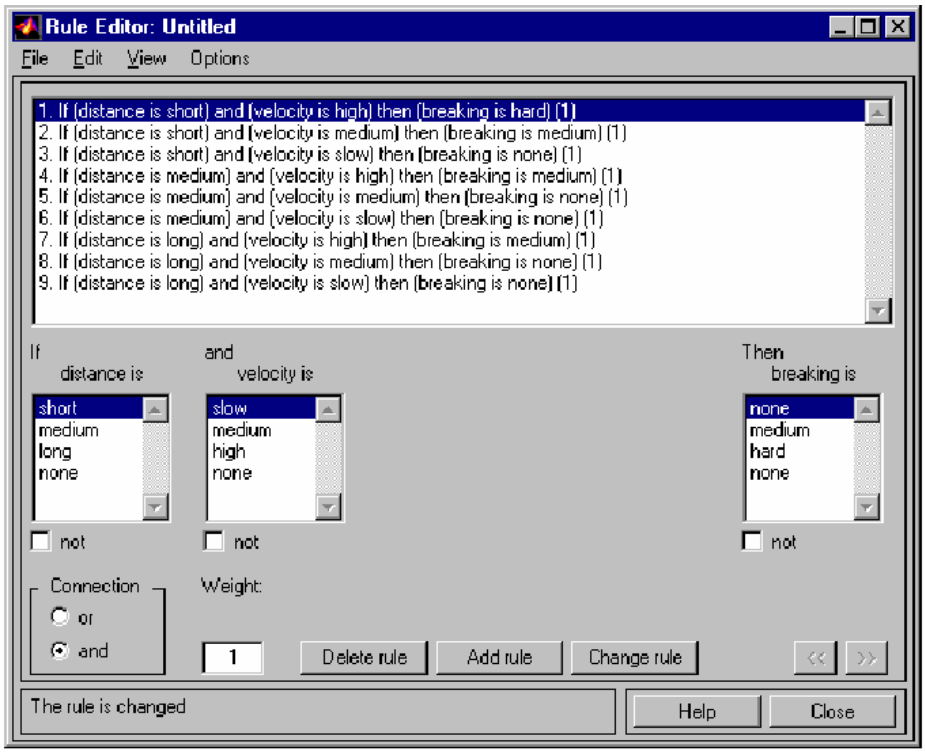
The second input can be added by opening *Edit* menu. Otherwise the steps are as before. Remember to give a name for the added input. Call it *velocity*.

Only *velocity* membership functions are shown because they are new. Note that the speed range is from -40 to 40 km/h and the range has been divided into three membership functions: *slow, medium,* and *high* speed.

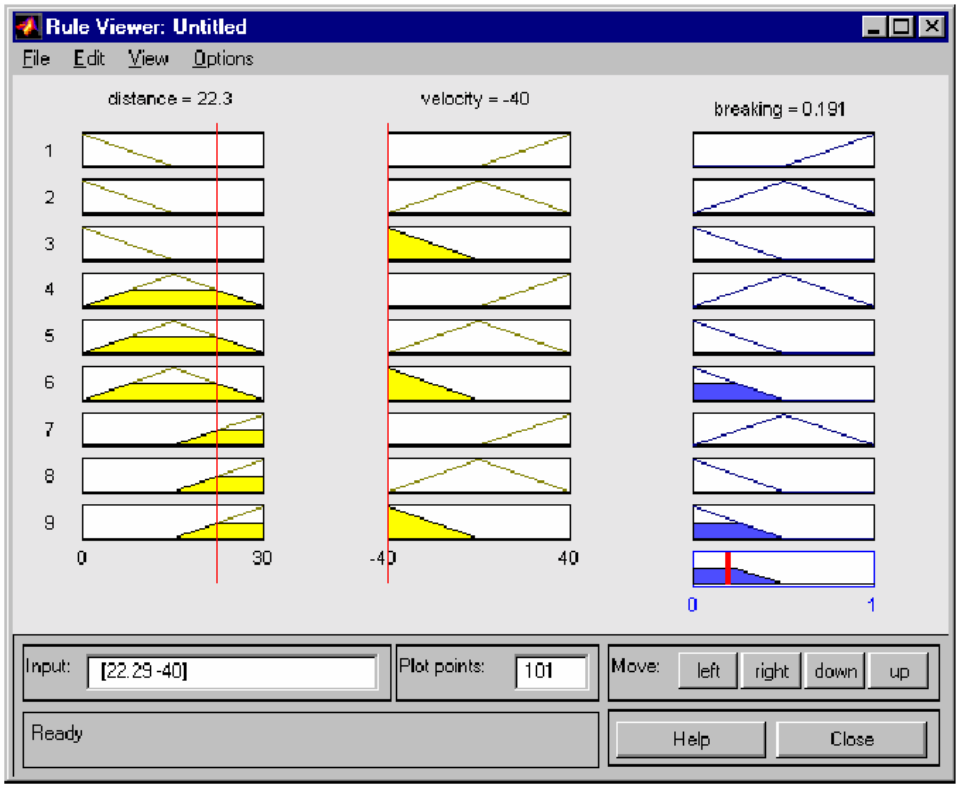


Membership functions for the new input, velocity.

Eight rules have been constructed, which are easy to understand on next Figure. Of course, the rules are not unique. Other rules could be used as well. The rule base
must be viewed and tested to see its effectiveness and how well the system specifications are satisfied.



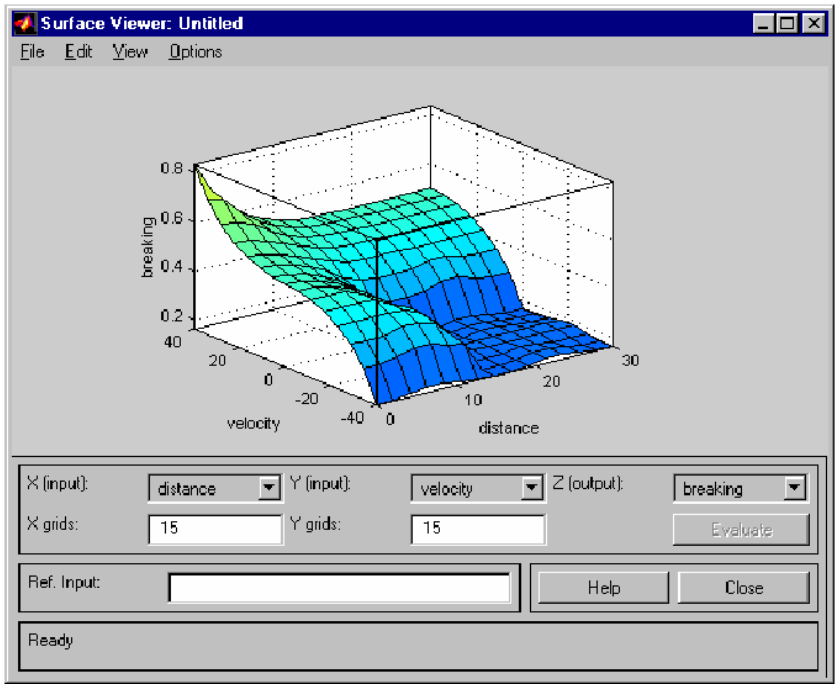
Viewing the rules is shown in next Figure



The rule base is viewed graphically. Different inputs can be chosen to see what the output will be.

Again you can test the system by choosing different values of inputs. If the result is not satisfactory, then refine the system. E.g. it seems that the area of short distance and high speed does not give strong enough braking response.

Finally, the surface view of the rules is shown



The surface view of the constructed rule base

Note that the right hand-side corner is flat with value zero over a fairly large area. You can change that by introducing a new membership function **little** for breaking power. You also have to change the rule base. This is done below.

2.4. Laboratory Work № 1 (LAB. 1)

The quality of service in restaurant is evaluated according to scale 0-10, where 10 means excellent and 0 poor service. The question is, how much tip we should give, that the relation between tip and service is correct.

There are three main rules:

i) If the service is poor, the tip is cheap.
ii) If the service is good, the tip is average.
iii) If the service is excellent, the tip is generous.

According to US practice, the cheap tip is 5%, average 15% and generous 25% of the total bill.

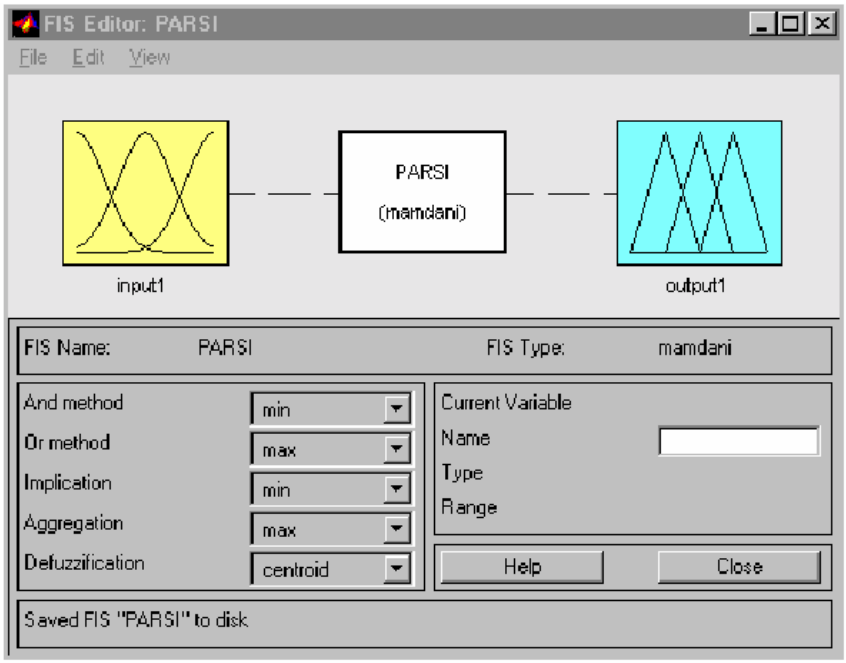
Formulate fuzzy deduction system ‘tip’. Use Matlab Fuzzy Logic Toolbox.

2.5. Sugeno-style Fuzzy Inference

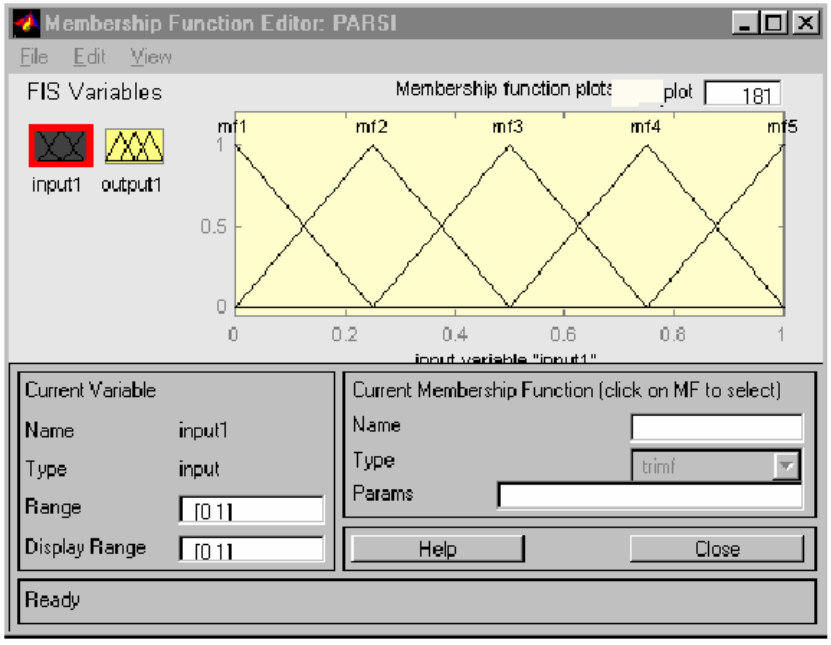
The result of Sugeno reasoning is an exact number.
Consider **input-output data** given in the table.

x	y
-1.0	1.0
-0.5	0.25
0	0
0.5	0.25
1.0	1.0

The data is from the function $y = x^2$. The data can be represented using Mamdani reasoning with e.g. triangular membership functions at the input and singletons at the output positioned at y_i.



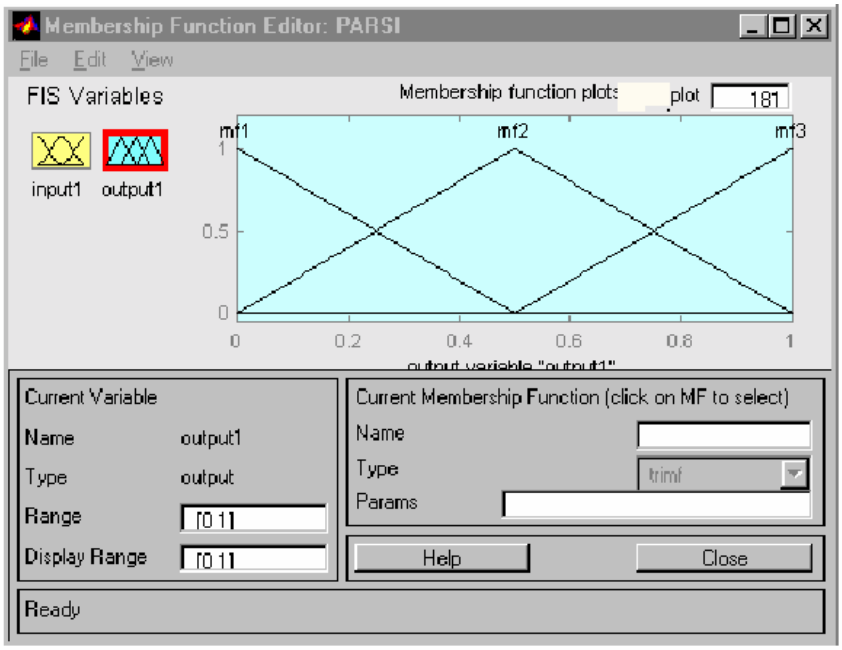
Default display of FIS Editor with Mamdani reasoning



The number of x_i points is five. Choose as many triangular membership functions for input variable

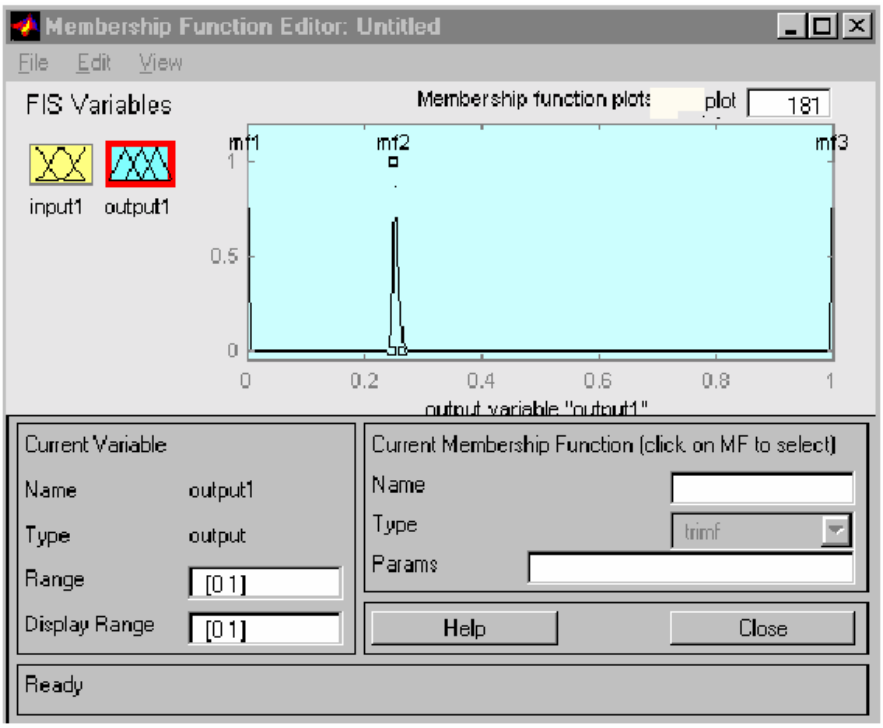
For each data point generate a triangular membership function, the maximum of which occurs exactly at a given data point, say *mf3* has maximum at $x = 0$, which is one of the data points. This is seen in the above figure.

Mamdani reasoning does not support singletons, so let's first choose triangular membership functions at the output.



The number of y_i points three. Choose as many triangular membership functions for output variable

Then make the base of the triangles narrow, so that they resemble singletons.

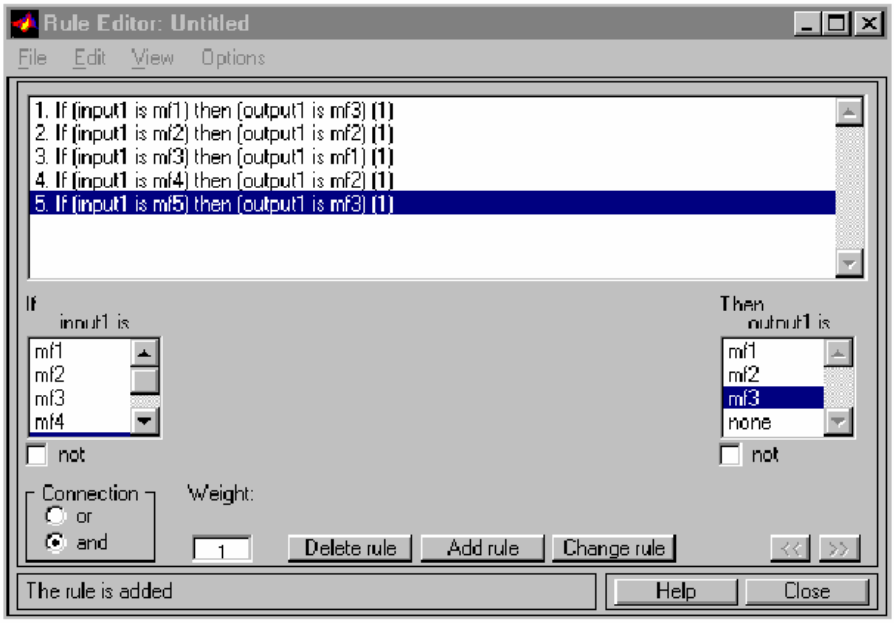


Make the basis of output triangular membership functions as narrow as possible to make them resemble singletons

Observe again the position of singletons. They have nonzero value exactly at the output data point. Only three membership functions are needed at the output. Why? Next set the rules to be

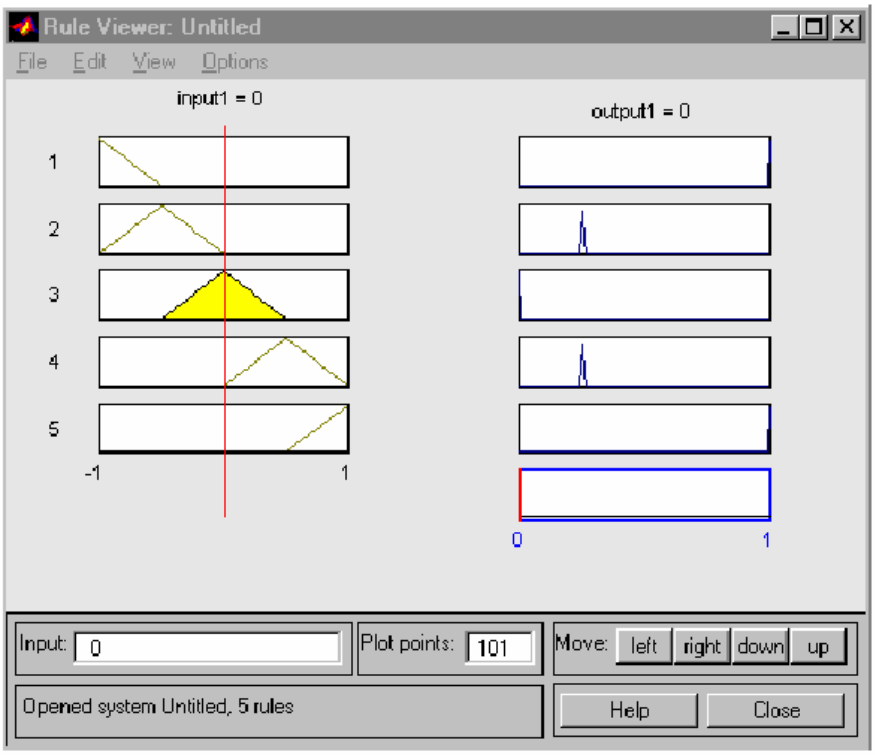
If x is mf_i then y is omf_i .

Complete rule base for the example is shown below.



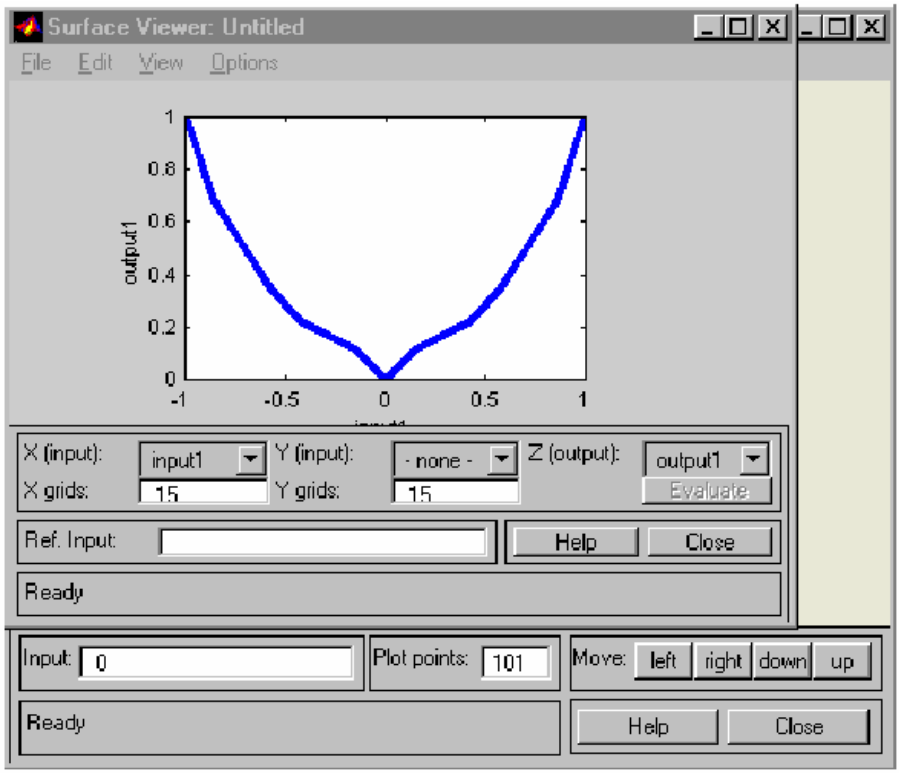
The rule base corresponding to the input-output data shown in the Table

We can view the rules by *View rules*



The complete rule base of the example

The result of our data fitting can be shown below



The resulting fuzzy system approximating the given data

The fit is **exact** at the given data points. Overall shape is that of parabola, but to have a better fit, more data points would be needed.

2.5.1. Exercise:

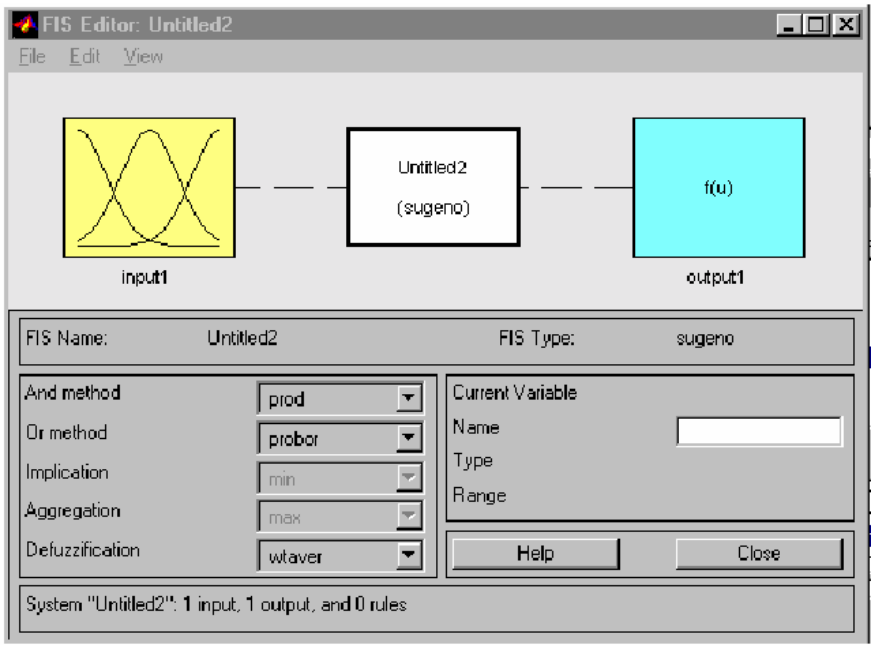
Keep the range the same. Add four more data points. Repeat the procedure and compare the results.

The same procedure as above can be produced with **Sugeno** reasoning.
In Sugeno reasoning the consequence, the output side, is deterministic:

If x is X_i then $y = y_i$.

In Mamdani reasoning the determinism is produced with singletons.

Let us repeat the above example with Sugeno reasoning.
In the FIS editor choose ***New Sugeno FIS***.

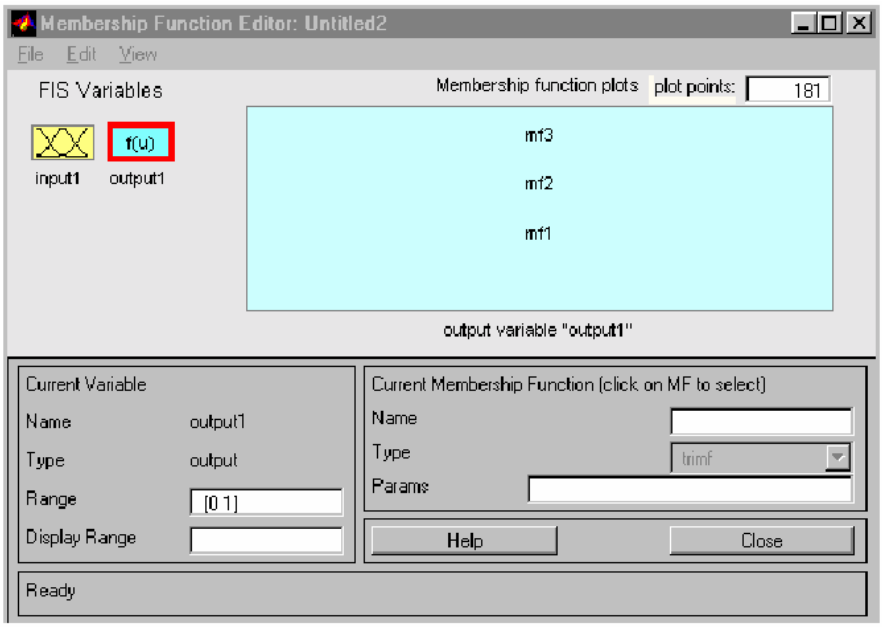


FIS Editor for Sugeno reasoning

Then activate the input and name it *x.* Similarly activate output and name it *y*. Next *Edit membership functions.* Let the input range be [-1 1]. Click *Ready* so that your choice is recorded. Determine the input membership functions as before by *Add MF's* (5 triangular). Next repeat the same for output. The range is [0 1], which is default value. Then *Add MF's*. At this point the following appears.

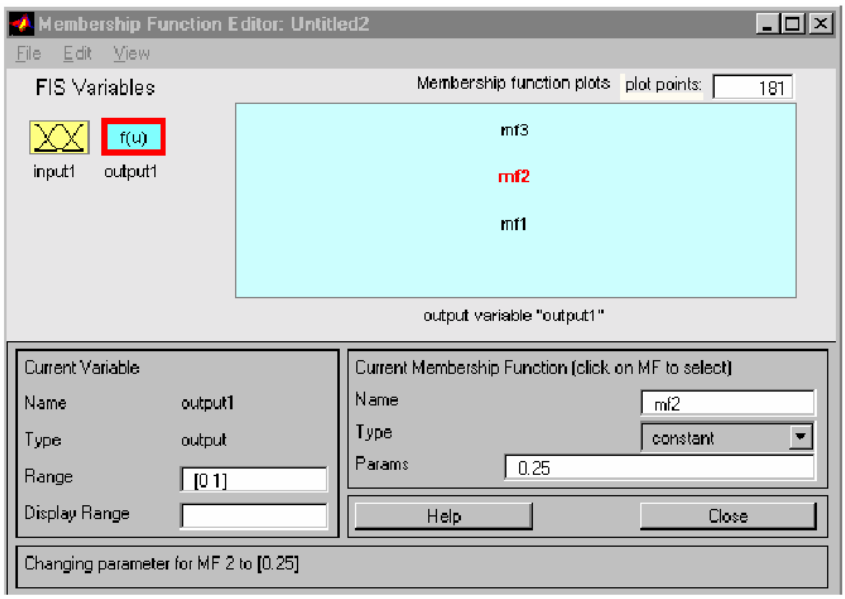
Choose three, constant output membership functions

MF type *constant* corresponds to Mamdani singletons. There are three different values of output, so choose 3 as *Number of MFs*. Clicking OK results in



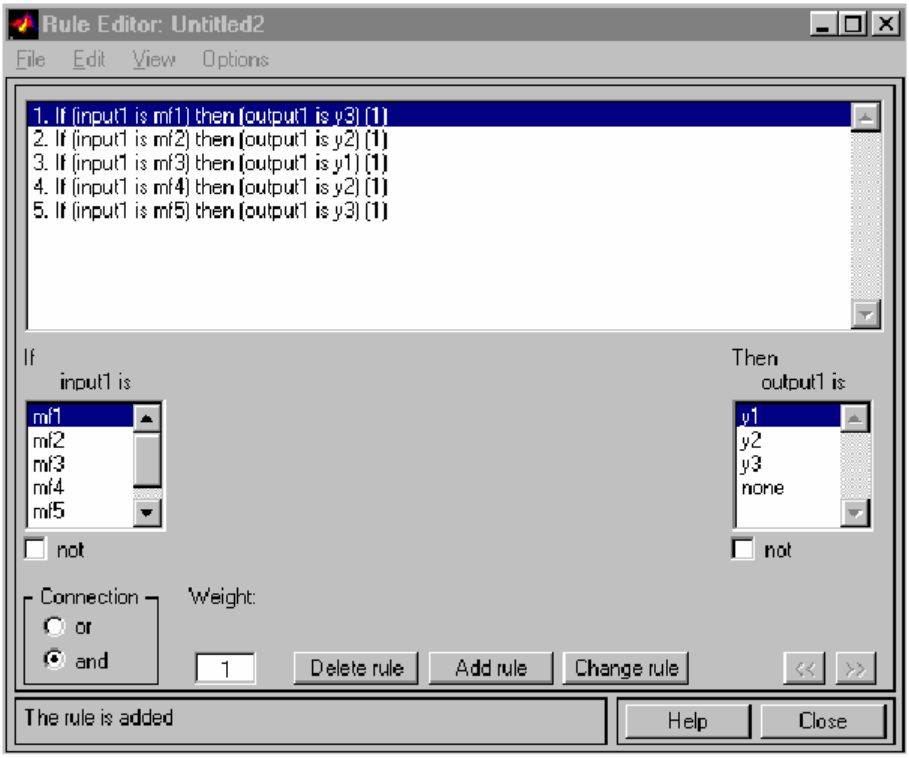
Membership Function Editor in Sugeno type of fuzzy system

The values of the constant membership functions are determined by activating them by clicking with the mouse. Now you change the name and value of the constant.



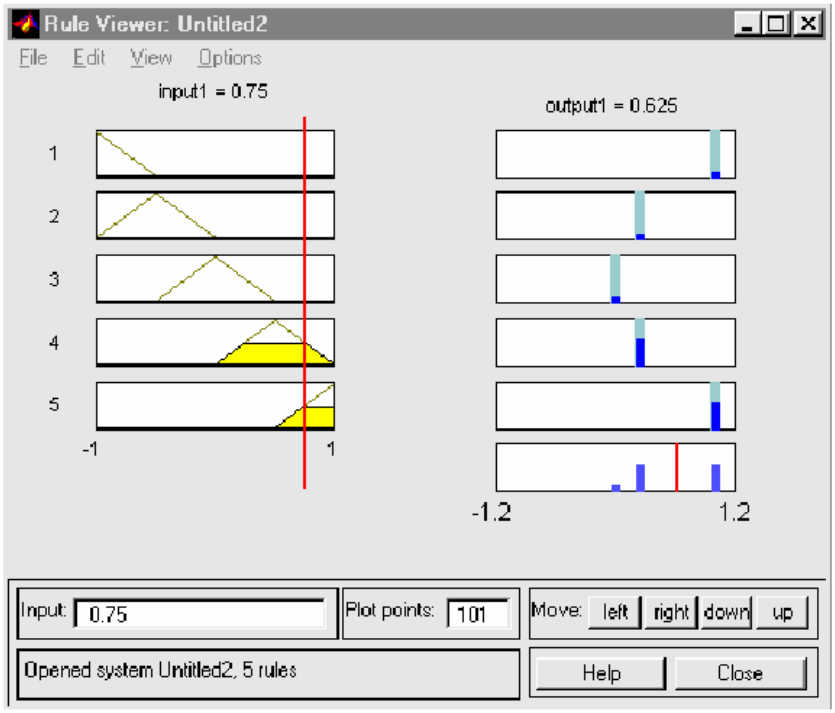
Activate membership function *mf2*

The final task is to form the rulebase. Choose *Edit rules* and write them in as before.



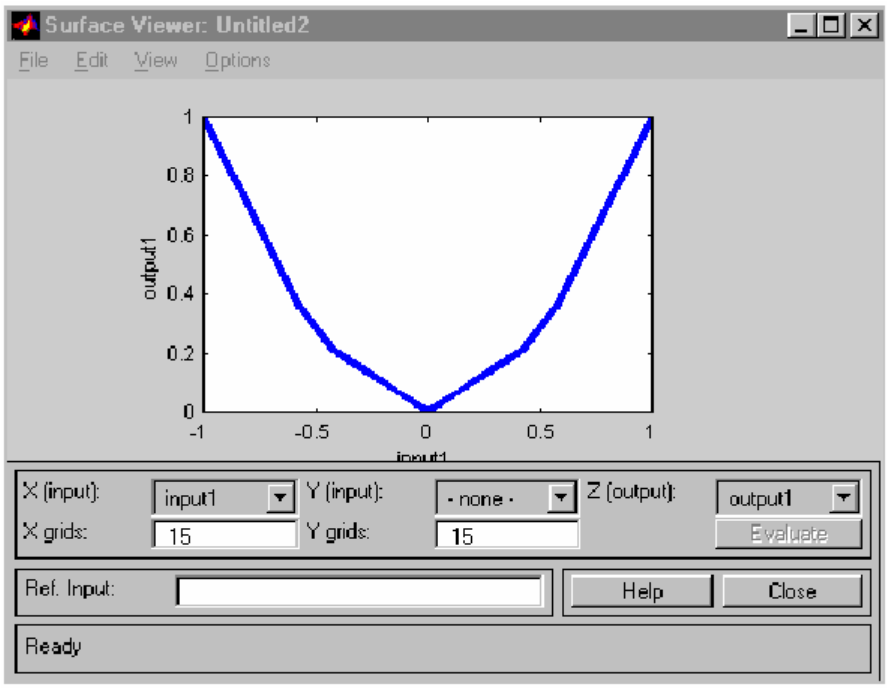
The rule base for the example in Sugeno type of fuzzy system

The Sugeno FIS system is now complete. Let us view the result. First the overall rules



The complete rule base of the example in the case of Sugeno type of fuzzy system

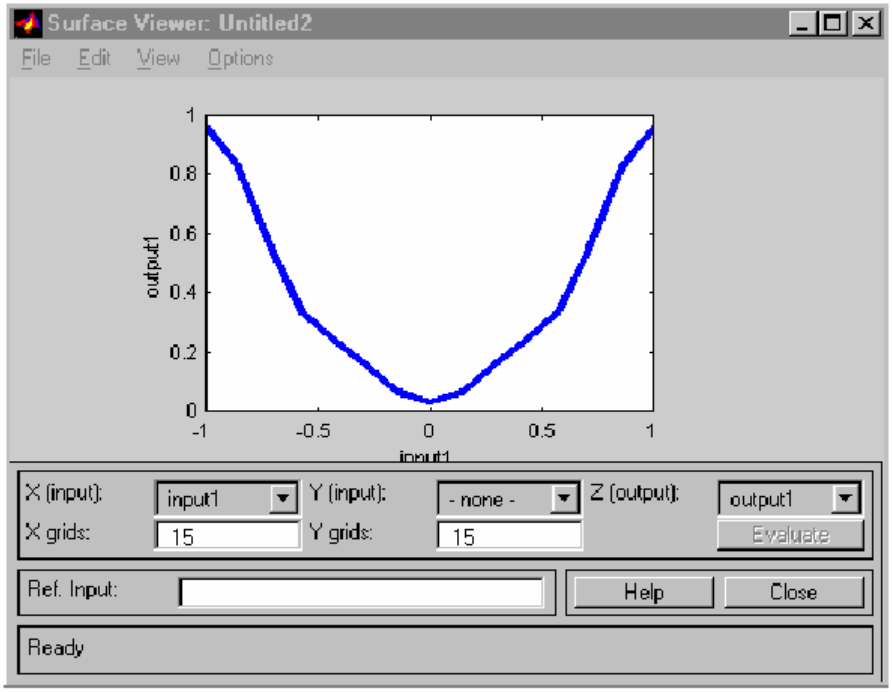
Then the Surface View.



The resulting fit of Sugeno type of fuzzy system

The result is the roughly the same as we obtained before.

Trying *gaussmf* membership functions results in the following approximating system.



Fuzzy Sugeno system with gaussian type of membership functions

More generally the above procedure would be the following. The input-output data is given by

x	y
x_0	y_0
x_1	y_1
. . .	. . .
x_n	y_n

The rules would be of the form

If x is X_i then $y = y_i$.

Then Sugeno type of reasoning with weighted average leads to

$$y(x) = \frac{\sum_{i=1}^{m} \mu_{X_i}(x) y_i}{\sum_{i=1}^{m} \mu_{X_i}(x)}$$

y_i = discretization points of membership functions

m = number of rules

REMARK: Weighted average requires that the rule base is complete and input fuzzy membership functions cover the input space. Otherwise there is a danger to divide with zero.

Sugeno reasoning allows us to use also functions of input x, not only constants, on the right hand side. The rules would look like

If x is X_i then $y = f_i(x)$.

Function f_i can be a different nonlinear mapping for each rule.

x may be a vector and more complicated rule structures can also appear.

Simplest examples of functions fi are straight lines. Let $i = 2$. Then

$$y = p_1 x + r_1$$

$$y = p_2 x + r_2$$

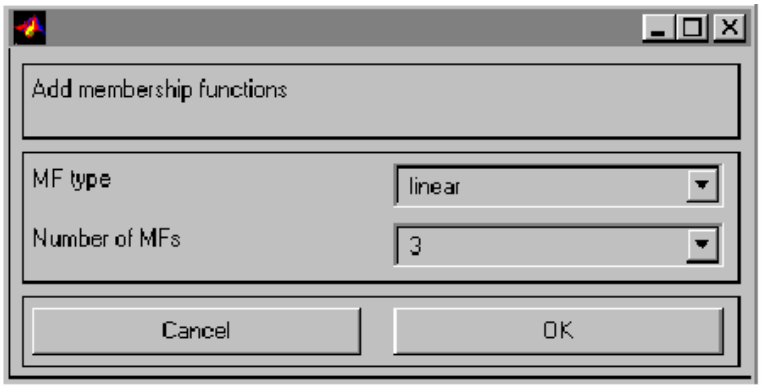
This is supported by *Fuzzy Toolbox.* More general functions can also be used, but these have to be set up by yourself.

2.5.2. Example:

Consider again the example above. Set up a Sugeno system with five membership functions as before. Use three straight lines at the output: One with a negative slope, one constant, and one with a positive slope. Start with very simple ones

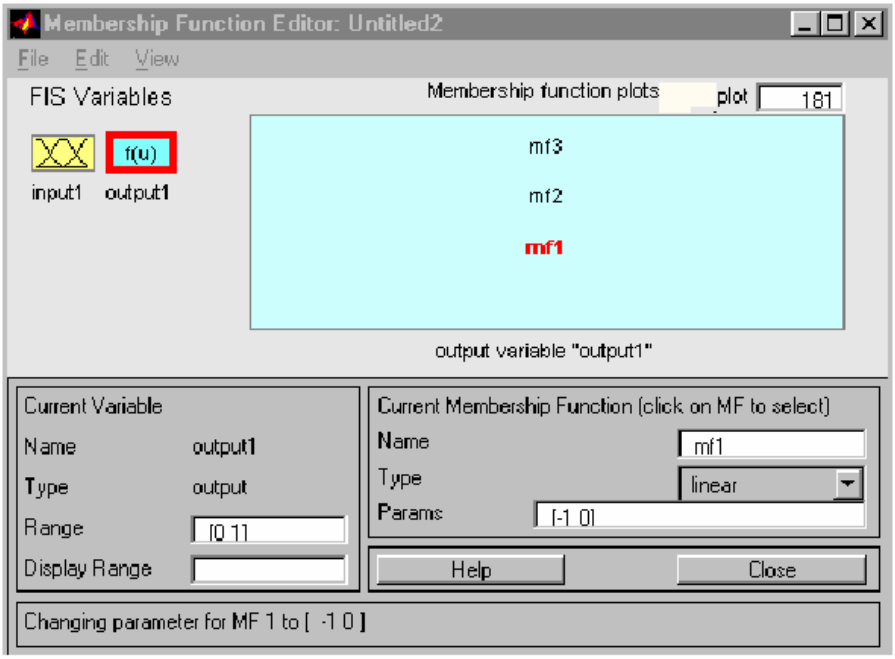
$$1: y = -x$$
$$2: y = 0$$
$$3: y = x$$

Define them by *Add MF's*. Choose *linear* type.

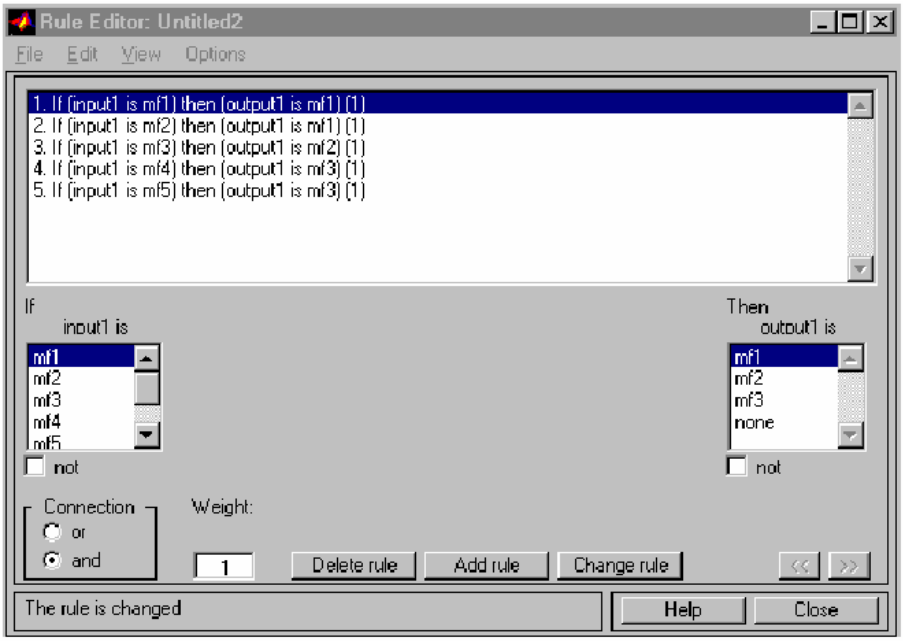


Choose linear form of membership functions instead of constant

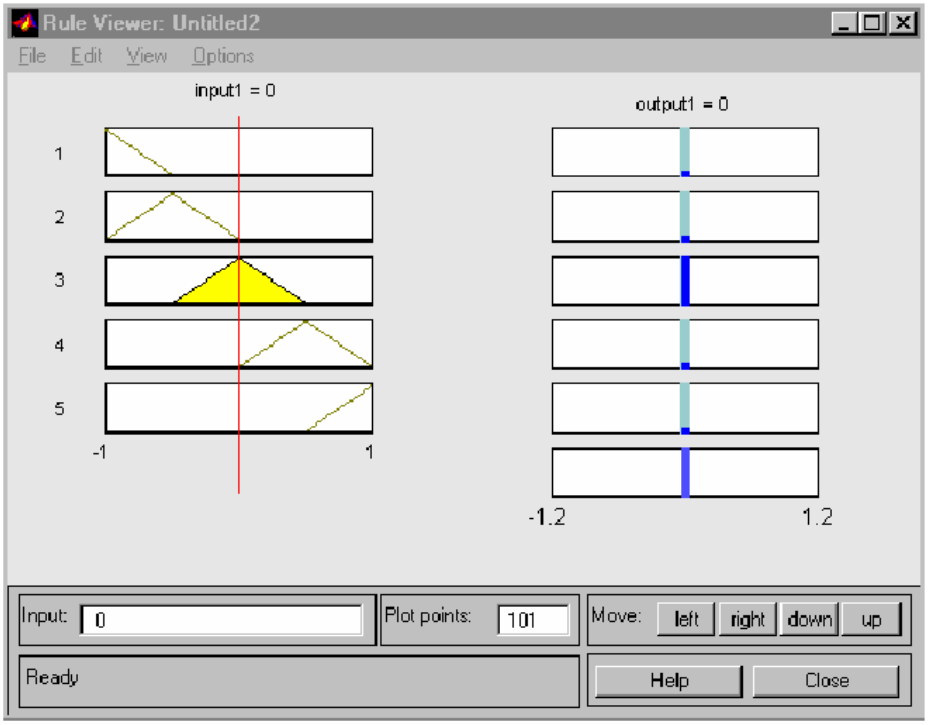
The two parameters for each straight line can be chosen in *Params* box.
The slope is first and then the constant.



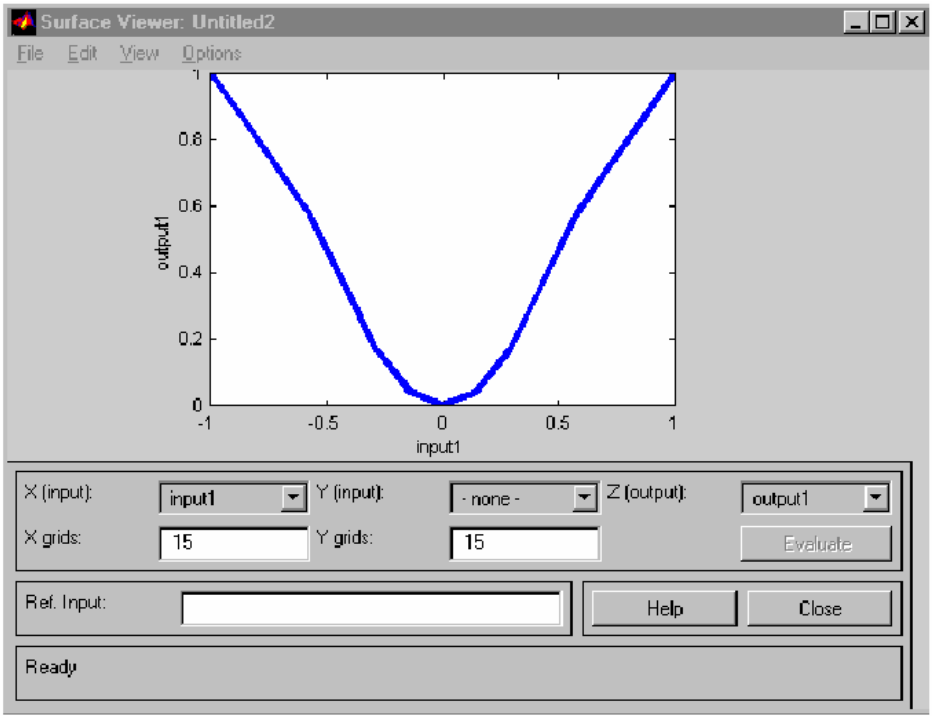
The rule base could be e.g.



The overall view of the rules is shown below.



The Surface View becomes:



The result is smoother than before, but does not agree as well at point $x = 0.5$, $y = 0.25$. This would require further fine-tuning of parameters.

Suppose the rules are of the form

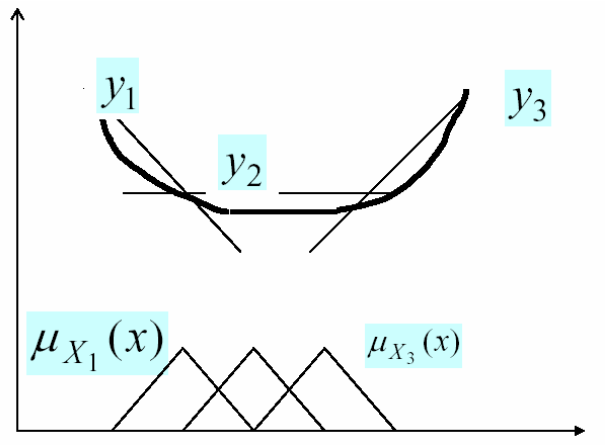
If x is X_i then $y = f_i(x)$

Combining results of all the rules leads to a weighted average

$$y(x) = \frac{\sum_{i=1}^{m} \mu_{X_i}(x) f_i(x)}{\sum_{i=1}^{m} \mu_{X_i}(x)}$$

where $m =$ number of rules.

The interpretation is that for a given value of x, the membership functions smooth (interpolate) the output function f_i. This is illustrated below for the case of straight lines y_1, y_2 and y_3.



A function defined by Sugeno model. The straight lines y_t, i = *1,2,3,* represent the local models. The bold line is the final result.

The rules can be more general like

If x_1 is A_1^i and … and x_n is A_n^i

then $y^i = f^i(x_1,...,x_n)$

where the consequences of the fuzzy rules are functions of the input vector **x** = [$x_1,\ldots,x_n$].A general linear function of Sugeno type at the output is

$$y^j = c_0^j + \sum_{k=1}^{n} c_k^j x_k ,$$

where c_k^j are real-valued parameters and specific to each rule. Since each rule has a crisp output, the aggregate result is obtained by weighted average.

$$y(\mathbf{x}) = \frac{\sum_{i=1}^{n} w_i(\mathbf{x})\, y^i(\mathbf{x})}{\sum_{i=1}^{n} w_i(\mathbf{x})}$$

where **x** = [$x_1,\ldots,x_n$] and w_i depends on the applied *T*-norm for logical *and*. If product is chosen and the number of rule is *m*, then

$$w_i = \prod_{u=1}^{b} \mu_{A_i}(x), i = 1,..., n$$

REMARK: Sugeno systems using constants or linear functions in the consequence are clearly parameterized maps. Therefore it is possible to use optimization techniques to find best parameters to fit data instead of trying to do it heuristically. Depending on the choice of T-norm and defuzzification, the expression above may change, but will always have similar structure.

2.6. Laboratory Work № 2 (LAB. 2)

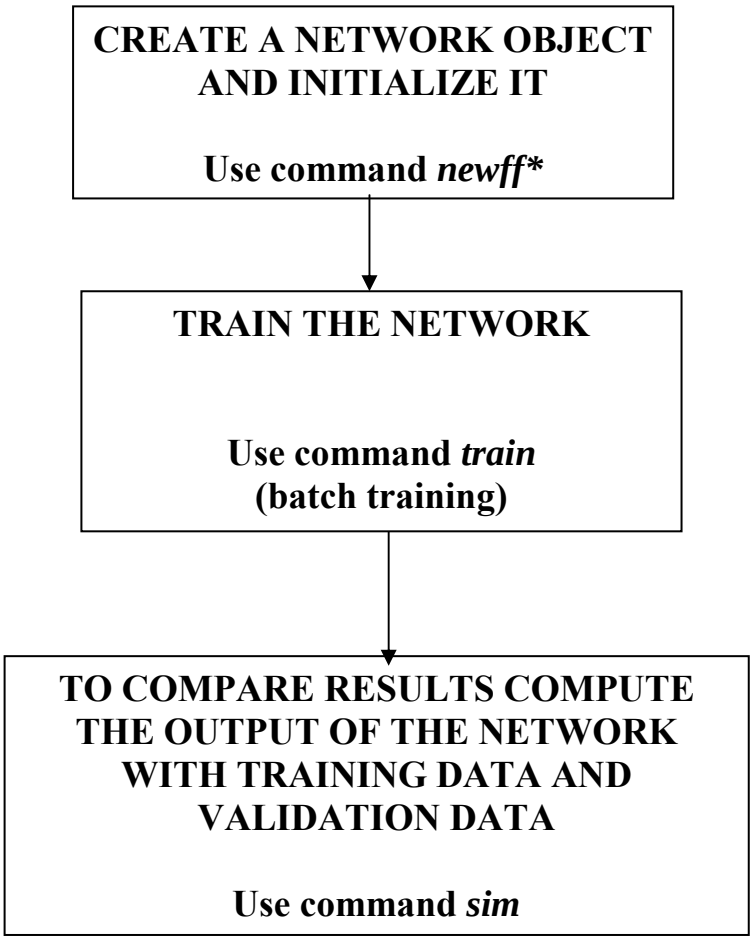
In the Sauna, the temperature should be comfortable. There is three different mode in electric bath-house stove. The bath-house stove can work in full power, half power and it can be switched off. When it works half power, it maintains the same temperature. When working in full power the temperature is increasing and when the electric bath-house stove is switched off, the temperature is decreasing. Temperature is observed.

Formulate fuzzy controller for electric bath-house stove. Use Matlab Fuzzy Toolbox and Sugeno-style fuzzy inference. Use to inputs: temperature and temperature change.

Chapter 3

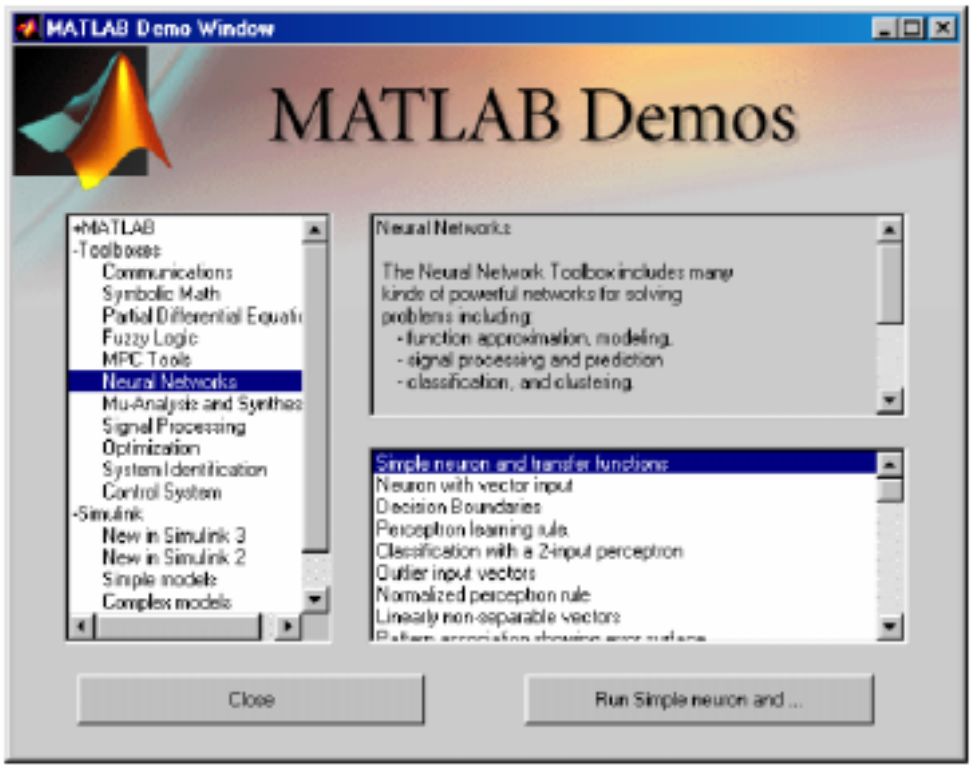
Neural Network Basics

BASIC FLOW DIAGRAM



*The command *newff* both defines the network (type of architecture, size and type of training algorithm to be used). It also automatically initializes the network. The last two letters in the command *newff* indicate the type of neural network in question: feedforward network. For radial basis function networks *newrb* and for Kohonen's Self-Organizing Map (SOM) *newsom* are used.

Before starting with the solved exercises, it is a good idea to study MATLAB Neural Network Toolbox demos. Type *demo* on MATLAB Command side and the MATLAB Demos window opens. Choose Neural Networks under Toolboxes and study the different windows.



3.1. EXAMPLE :

Consider *humps* function in MATLAB. It is given by

$$y = 1./((x-.3).\hat{}2+.01)+1./((x-.9).\hat{}2+.04)-6;$$

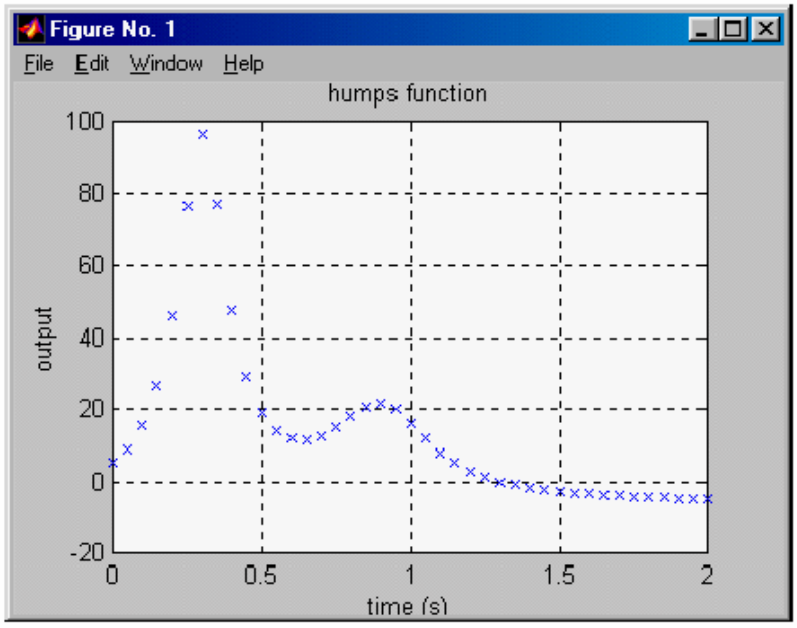
but in MATLAB can be called by *humps.* Here we like to see if it is possible to find a neural network to fit the data generated by humps-function between [0,2].

a) Fit a multilayer perceptron network on the data. Try different network sizes and different teaching algorithms.
b) Repeat the exercise with radial basis function networks.

SOLUTION: To obtain the data use the following commands

x = 0:.05:2; y=humps(x);
P=x; T=y;

Plot the data
plot(P,T,'x')
grid; xlabel('time (s)'); ylabel('output'); title('humps function')



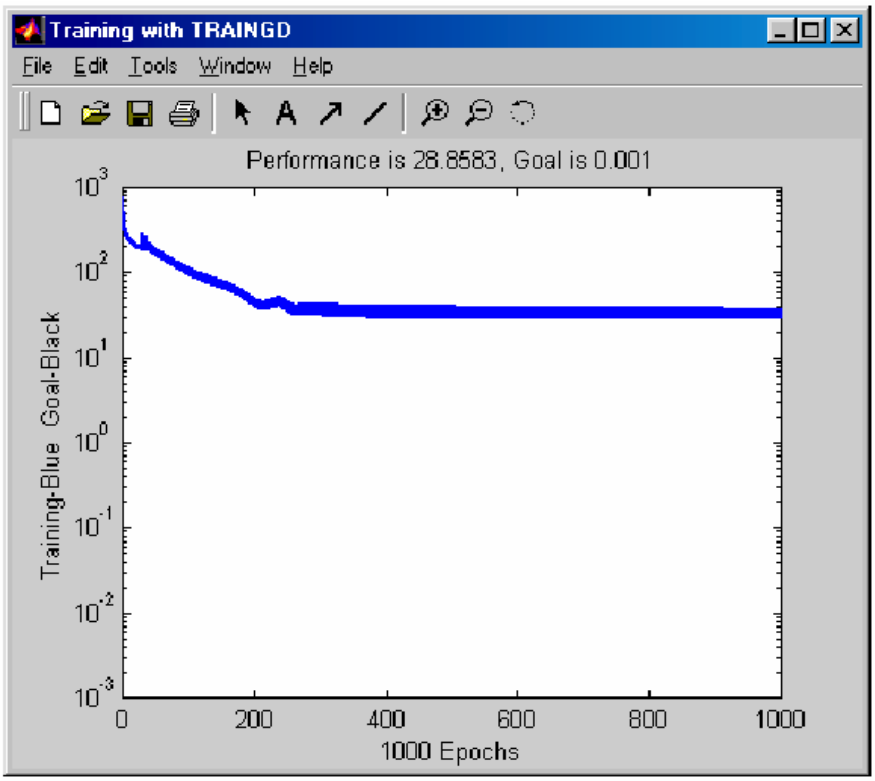
The teaching algorithms for **multilayer perceptron networks** have the following structure:
a. Initialize the neural network parameters, weights and biases, either providing them yourself or using initializing routines. At the same time define the structure of the network.
b. Define the parameters associated with the algorithm like error goal, maximum number of epochs (iterations), etc.
c. Call the teaching algorithm.

```
% DESIGN THE NETWORK
% ==================
%First try a simple one – feedforward (multilayer perceptron) network
net=newff([0 2], [5,1], {'tansig','purelin'},'traingd');
% Here newff defines feedforward network architecture.
% The first argument [0 2] defines the range of the input and initializes the network parameters.
% The second argument the structure of the network. There are two layers.
% 5 is the number of the nodes in the first hidden layer,
% 1 is the number of nodes in the output layer,
% Next the activation functions in the layers are defined.
% In the first hidden layer there are 5 tansig functions.
% In the output layer there is 1 linear function.
% 'learngd' defines the basic learning scheme – gradient method
% Define learning parameters
net.trainParam.show = 50; % The result is shown at every 50th iteration (epoch)
net.trainParam.lr = 0.05; % Learning rate used in some gradient schemes
net.trainParam.epochs =1000; % Max number of iterations
net.trainParam.goal = 1e-3; % Error tolerance; stopping criterion
%Train network
net1 = train(net, P, T); % Iterates gradient type of loop
% Resulting network is strored in net1
TRAINGD, Epoch 0/1000, MSE 765.048/0.001, Gradient 69.9945/1e-010
....
TRAINGD, Epoch 1000/1000, MSE 28.8037/0.001, Gradient 33.0933/1e-010
TRAINGD, Maximum epoch reached, performance goal was not met.
```

The goal is still far away after 1000 iterations (epochs).

REMARK 1: If you cannot observe exactly the same numbers or the same performance, this is not surprising. The reason is that *newff* uses random number generator in creating the initial values for the network weights. Therefore the initial network will be different even when exactly the same commands are used.

Convergence is shown below.

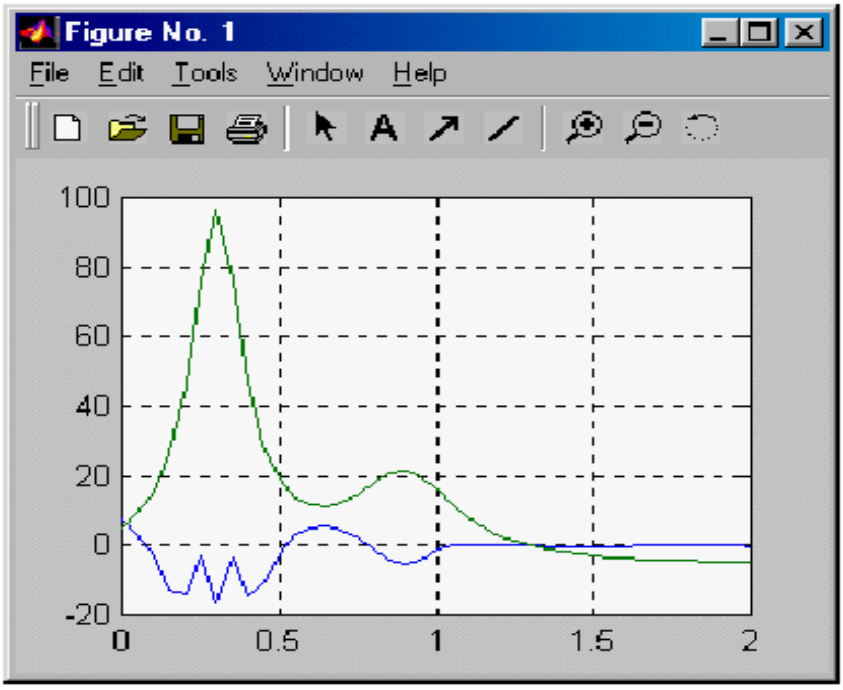


It is also clear that even if more iterations will be performed, no improvement is in store. Let us still check how the neural network approximation looks like.

% Simulate how good a result is achieved: Input is the same input vector P.
% Output is the output of the neural network, which should be compared with output data

a= sim(net1,P);

% Plot result and compare
plot(P,a-T, P,T); grid;



The fit is quite bad, especially in the beginning. What is there to do? Two things are apparent. With all neural network problems we face the question of determining the reasonable, if not optimum, size of the network. Let us make the size of the network bigger. This brings in also more network parameters, so we have to keep in mind that there are more data points than network parameters. The other thing, which could be done, is to improve the training algorithm performance or even change the algorithm.
We'll return to this question shortly.

Increase the size of the network: Use 20 nodes in the first hidden layer.

net=newff([0 2], [20,1], {'tansig','purelin'},'traingd');

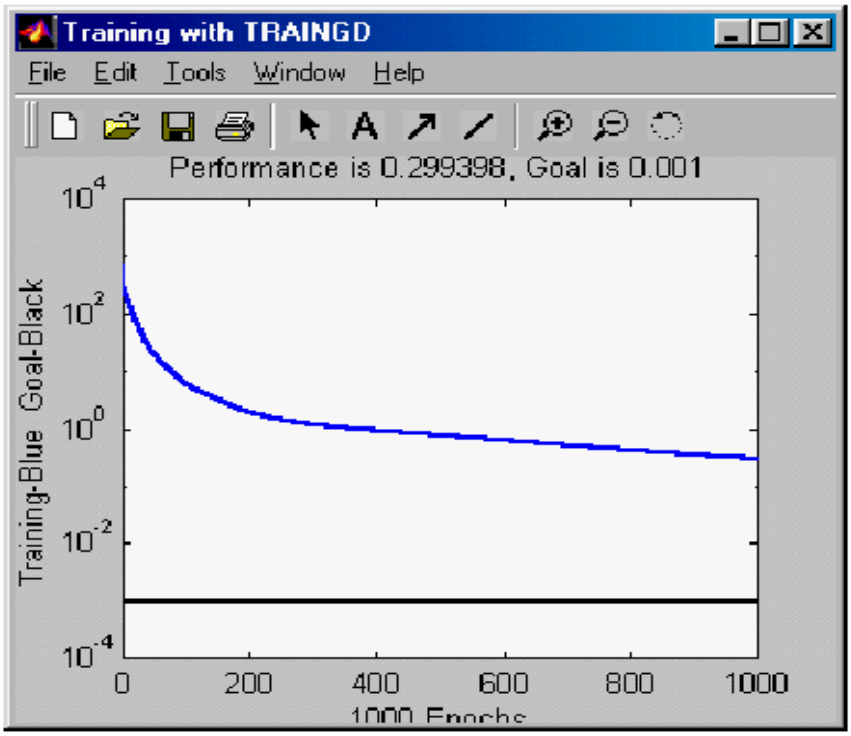
Otherwise apply the same algorithm parameters and start the training process.

net.trainParam.show = 50; % The result is shown at every 50th iteration (epoch)
net.trainParam.lr = 0.05; % Learning rate used in some gradient schemes
net.trainParam.epochs =1000; % Max number of iterations
net.trainParam.goal = 1e-3; % Error tolerance; stopping criterion

%Train network
net1 = train(net, P, T); % Iterates gradient type of loop

TRAINGD, Epoch 1000/1000, MSE 0.299398/0.001, Gradient 0.0927619/1e-010
TRAINGD, Maximum epoch reached, performance goal was not met.

The error goal of 0.001 is not reached now either, but the situation has improved significantly.



From the convergence curve we can deduce that there would still be a chance to improve the network parameters by increasing the number of iterations (epochs). Since the backpropagation (gradient) algorithm is known to be slow, we will try next a more efficient training algorithm.

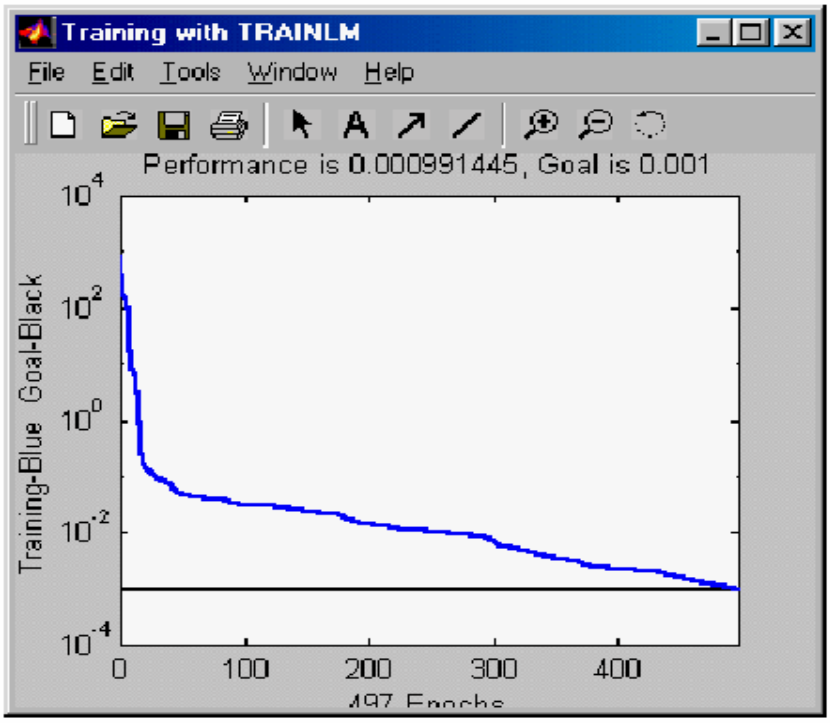
Try **Levenberg-Marquardt** – *trainlm.* Use also smaller size of network – 10 nodes in the first hidden layer.
net=newff([0 2], [10,1], {'tansig','purelin'},'trainlm');

%Define parameters

net.trainParam.show = 50;
net.trainParam.lr = 0.05;
net.trainParam.epochs =1000;
net.trainParam.goal = 1e-3;
%Train network

net1 = train(net, P, T);
TRAINLM, Epoch 0/1000, MSE 830.784/0.001, Gradient 1978.34/1e-010
....
TRAINLM, Epoch 497/1000, MSE 0.000991445/0.001, Gradient 1.44764/1e-010
TRAINLM, Performance goal met.

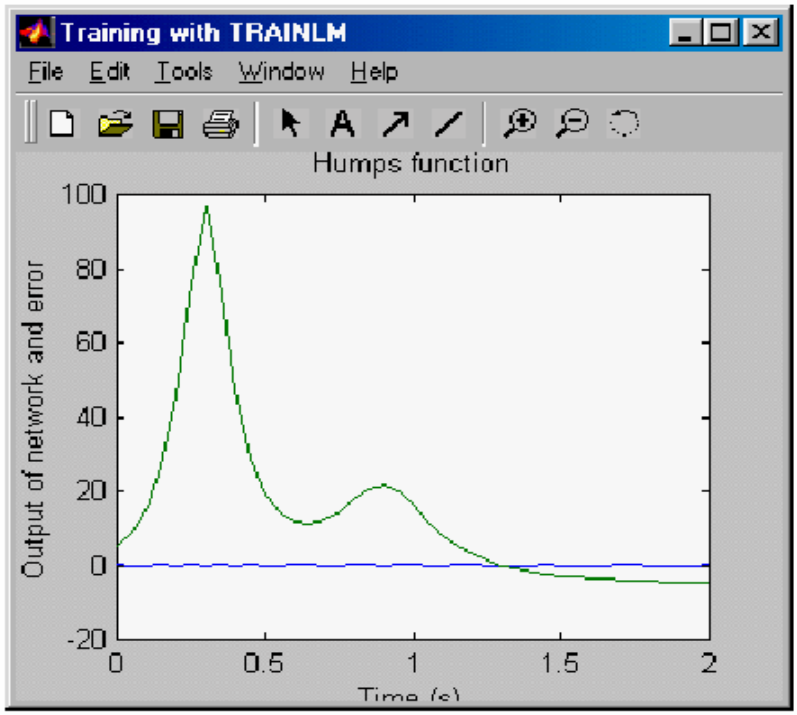
The convergence is shown in the Figure



Performance is now according to the tolerance specification.

%Simulate result
a= sim(net1,P);

%Plot the result and the error
plot(P,a-T,P,T)
xlabel('Time (s)'); ylabel('Output of network and error'); title('Humps function')



It is clear that L-M algorithm is significantly faster and preferable method to back-propagation. Note that depending on the initialization the algorithm converges slower or faster.

There is also a question about the fit: should all the dips and abnormalities be taken into account or are they more result of poor, noisy data.

When the function is fairly flat, then multilayer perception network seems to have problems.

Try simulating with independent data.
x1=0:0.01:2; P1=x1;y1=humps(x1); T1=y1;
a1= sim(net1,P1);
plot(P1,a-a1,P1,T1,P,T)

If in between the training data points are used, the error remains small and we cannot see very much difference with the figure above.

Such data is called test data. Another observation could be that in the case of a fairly flat area, neural networks have more difficulty than with more varying data.

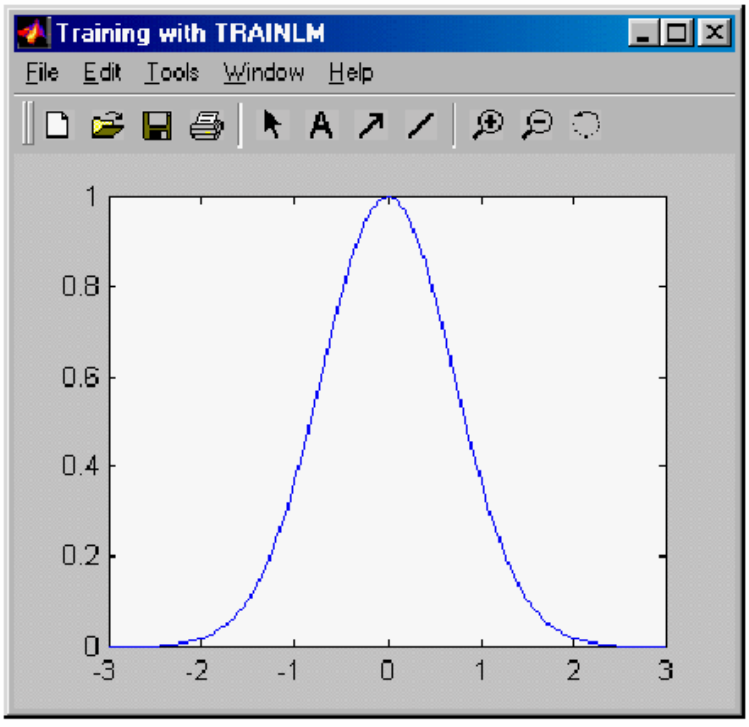
3.2. Radial Basic Function Networks

Here we would like to find a function, which fits the 41 data points using a radial basis network.

A radial basis network is a network with two layers. It consists of a hidden layer of radial basis neurons and an output layer of linear neurons.

Here is a typical shape of a radial basis transfer function used by the hidden layer:

p = -3:.1:3;
a = radbas(p);
plot(p,a)



The weights and biases of each neuron in the hidden layer define the position and width of a radial basis function.

Each linear output neuron forms a weighted sum of these radial basis functions. With the correct weight and bias values for each layer, and enough hidden neurons, a radial basis network can fit any function with any desired accuracy.

We can use the function *newrb* to quickly create a radial basis network, which approximates the function at these data points.

From MATLAB help command we have the following description of the algorithm.
Initially the RADBAS layer has no neurons. The following steps are repeated until the network's mean squared error falls below GOAL.
1) The network is simulated
2) The input vector with the greatest error is found
3) A RADBAS neuron is added with weights equal to that vector.
4) The PURELIN layer weights are redesigned to minimize error.

Generate data as before

```
x = 0:.05:2; y=humps(x);
P=x; T=y;
```

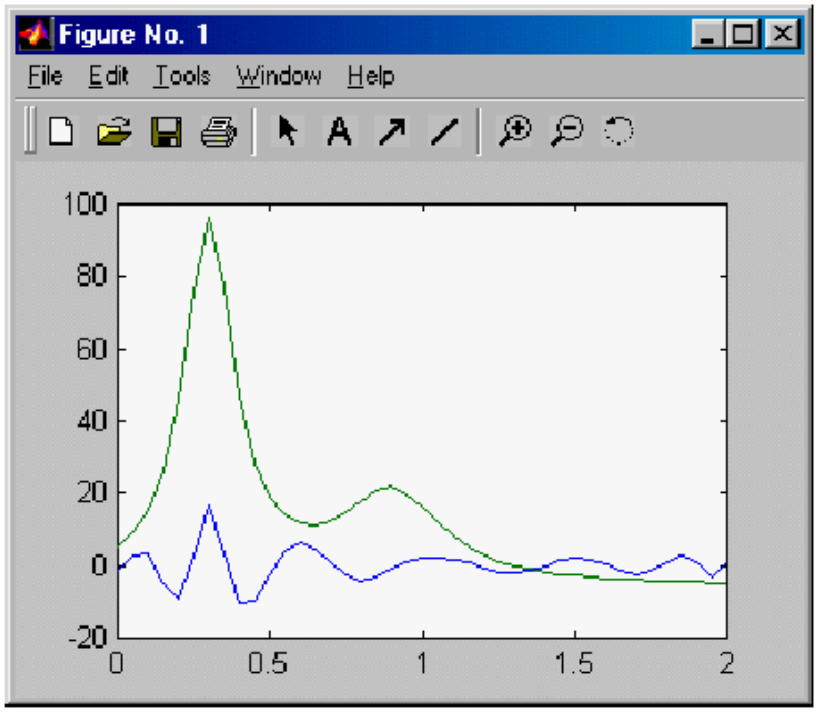
The simplest form of *newrb* command is

```
net1 = newrb(P,T);
```

For humps the network training leads to singularity and therefore difficulties in training.

Simulate and plot the result

```
a= sim(net1,P);
plot(P,T-a,P,T)
```



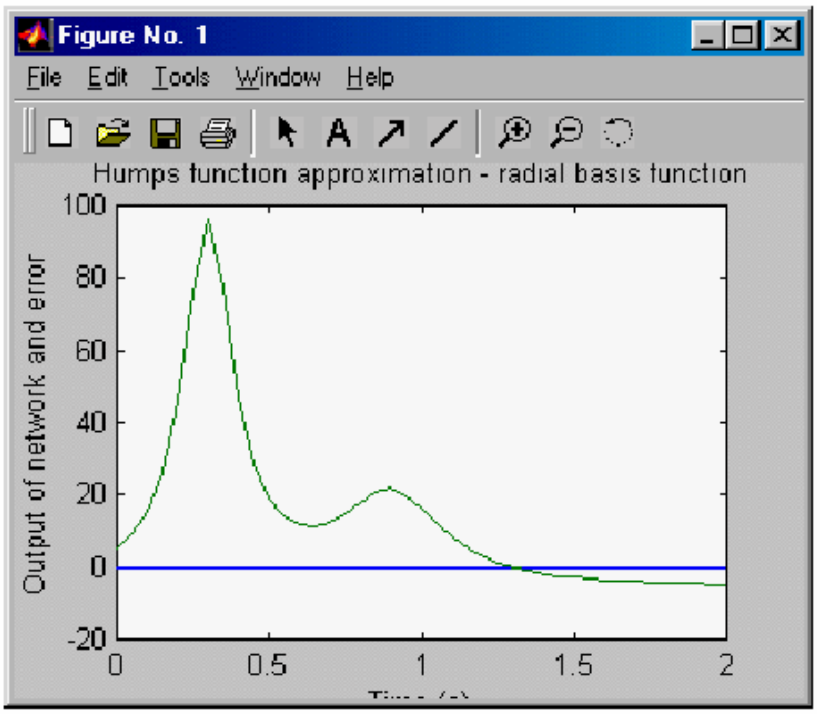
The plot shows that the network approximates *humps* but the error is quite large. The problem is that the default values of the two parameters of the network are not very good. Default values are *goal* -mean squared error goal = 0.0, *spread* - spread of radial basis functions = 1.0.

In our example choose *goal* = 0.02 and *spread* = 0.1.

goal=0.02; spread= 0.1;
```
net1 = newrb(P,T,goal,spread);
```
Simulate and plot the result

a= sim(net1,P);
plot(P,T-a,P,T)

xlabel('Time (s)'); ylabel('Output of network and error');
title('Humps function approximation - radial basis function')



This choice will lead to a very different end result as seen in the figure.

QUESTION: What is the significance of small value of spread. What about large?

The problem in the first case was too large a spread (default = 1.0), which will lead to too sparse a solution. The learning algorithm requires matrix inversion and therefore the problem with singularity.
By better choice of spread parameter result is quite good.

Test also the other algorithms, which are related to radial base function or similar

networks NEWRBE,
NEWGRNN, NEWPNN.

3.3. EXAMPLE :

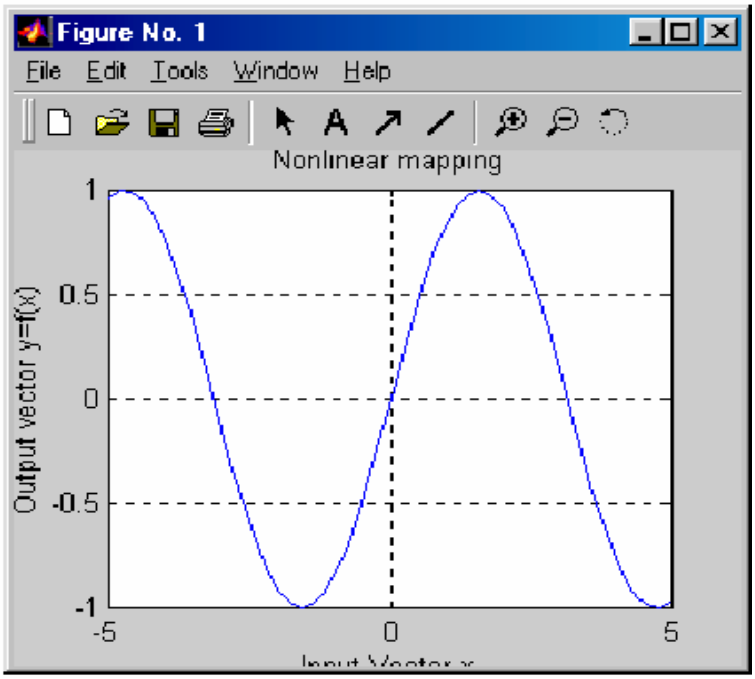
Simulate the hybrid system

$$\ddot{x} = -\dot{x} - f(x)$$

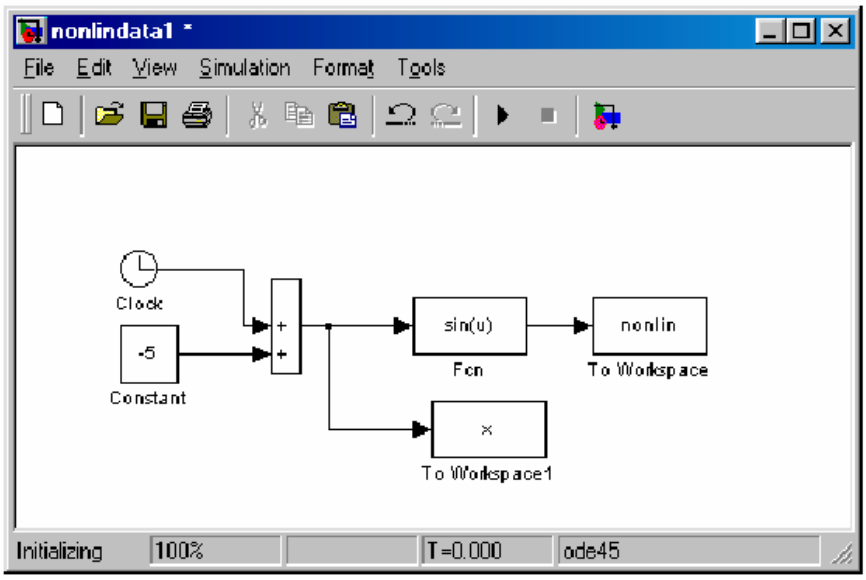
for different initial conditions. Function $y = f(x)$ is has been measured and you must first fit a neural network on the data. Use both backpropagation and radial basis function networks. The data is generated using $y=f(x)=\sin(x)$.

-5.0000 0.9589
-4.6000 0.9937
-4.2000 0.8716
-3.8000 0.6119
-3.4000 0.2555
-3.0000 -0.1411

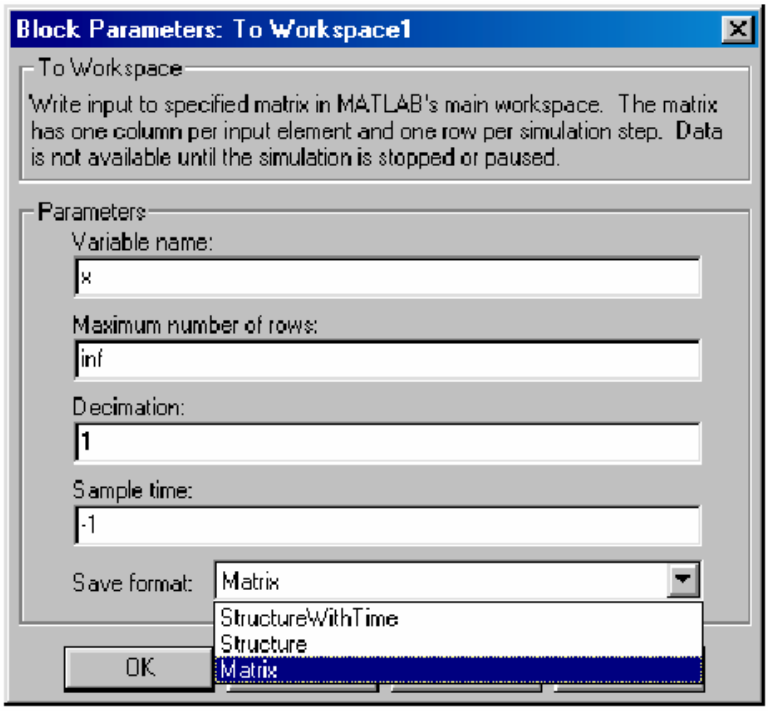
-2.6000 -0.5155
-2.2000 -0.8085
-1.8000 -0.9738
-1.4000 -0.9854
-1.0000 -0.8415
-0.6000 -0.5646
-0.2000 -0.1987
0.2000 0.1987
0.6000 0.5646
1.0000 0.8415
1.4000 0.9854
1.8000 0.9738
2.2000 0.8085
2.6000 0.5155
3.0000 0.1411
3.4000 -0.2555
3.8000 -0.6119
4.2000 -0.8716
4.6000 -0.9937
5.0000 -0.9589



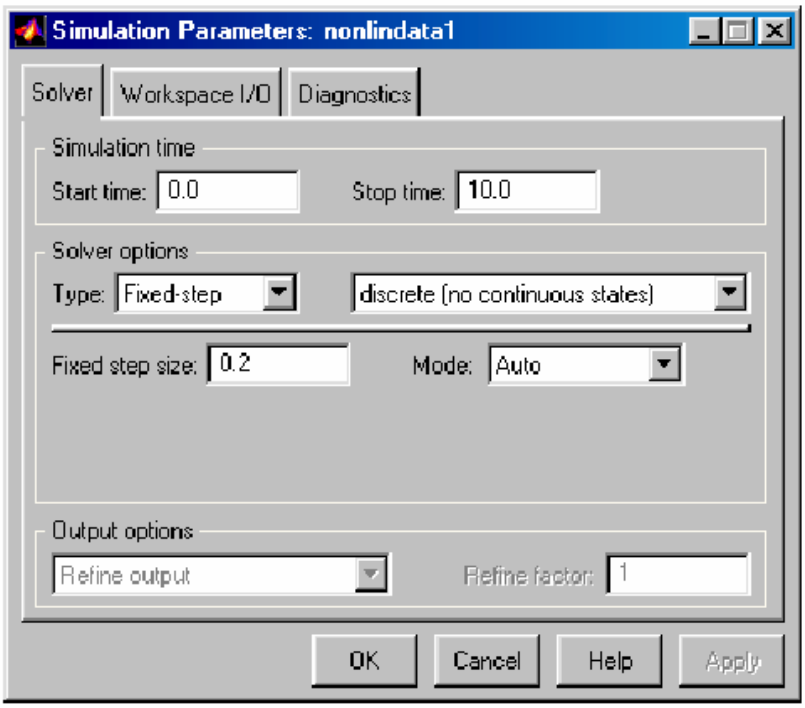
Instead of typing the data generate it with the following SIMULINK model shown below.



When you use workspace blocks, choose *Matrix* Save format. Open *To workspace* block and choose *Matrix* in the Menu and click OK. In this way the data is available in Command side.



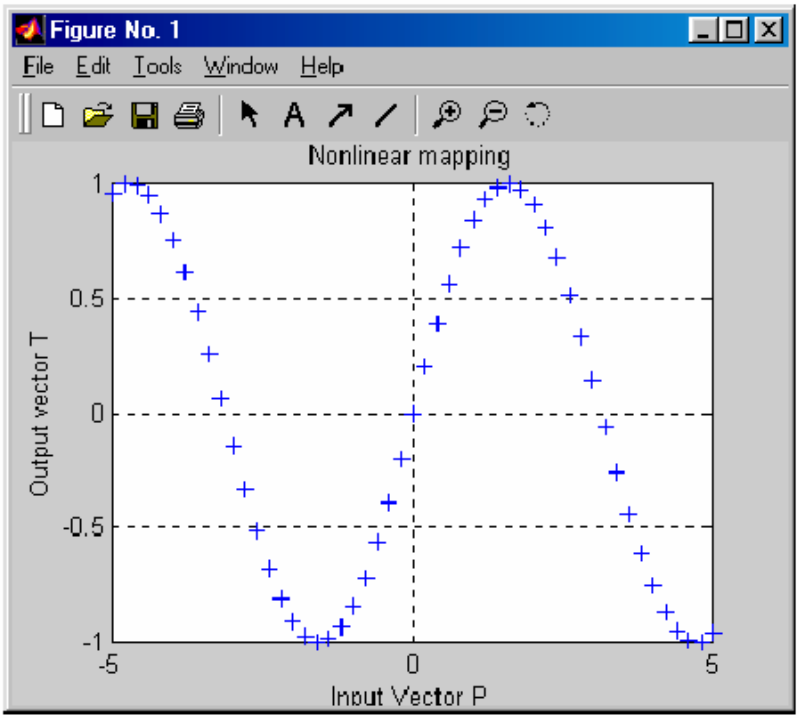
The simulation parameters are chosen from simulation menu given below, fixed-step method, step size = 0.2. Observe also the start and stop times. SIMULINK can be used to generate handily data. This could also be done on the command side. Think of how to do it.



SOLUTION:

Define the input and output data vectors for the neural networks.

```
P=x';T=nonlin';
plot(P,T,'+')
title('Nonlinear mapping');
xlabel('Input Vector P');
ylabel('Output vector T');
grid;
gi=input('Strike any key ...');
```

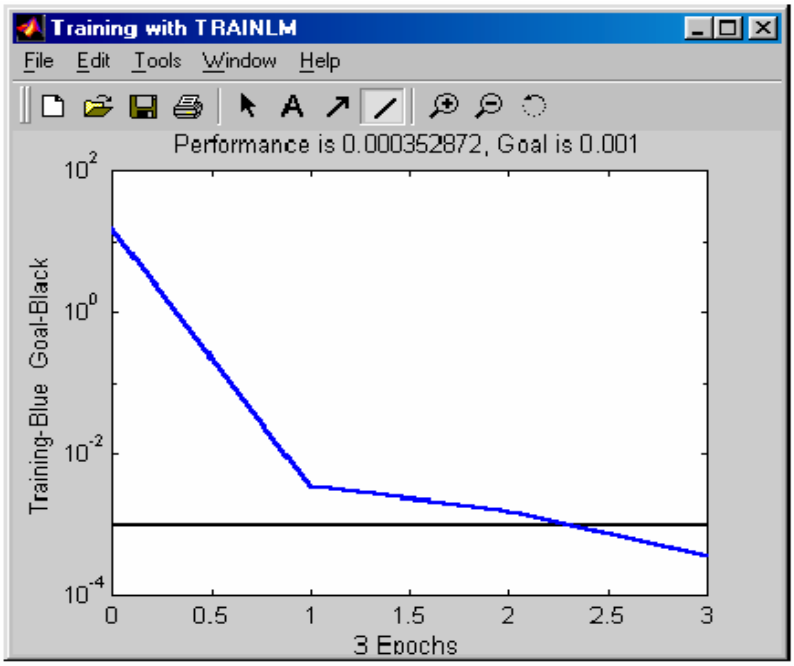


% LEVENBERG-MARQUARDT:
net=newff([-6 6], [20,1], {'tansig','purelin'},'trainlm');

%Define parameters
net.trainParam.show = 50;
net.trainParam.lr = 0.05;
net.trainParam.epochs = 500;
net.trainParam.goal = 1e-3;

%Train network
net1 = train(net, P, T);
TRAINLM, Epoch 0/500, MSE 15.6185/0.001, Gradient 628.19/1e-010
TRAINLM, Epoch 3/500, MSE 0.000352872/0.001, Gradient 0.0423767/1e-010
TRAINLM, Performance goal met.

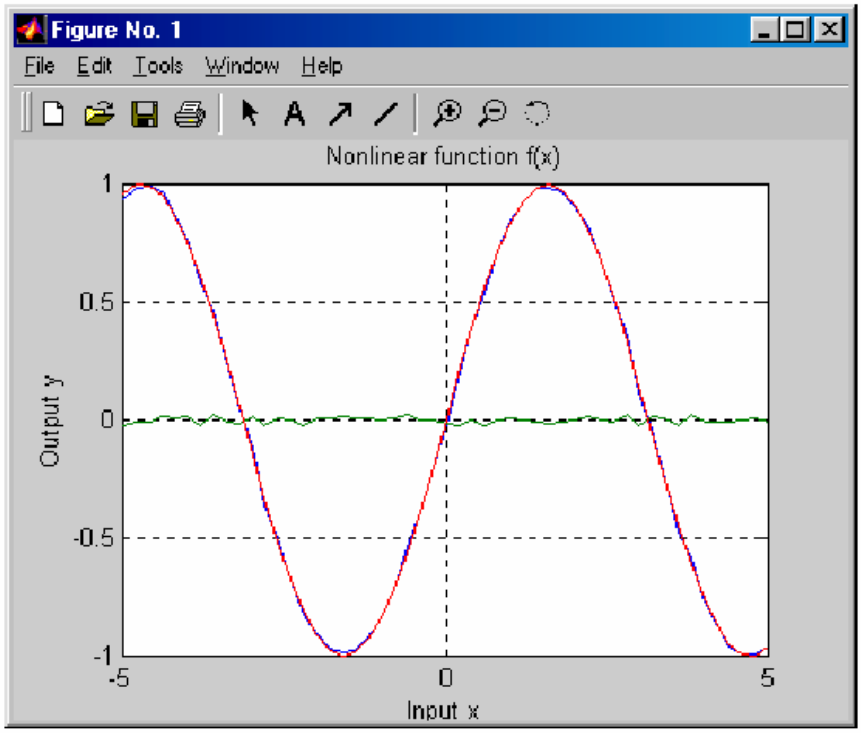
The figure below shows convergence. The error goal is reached.



The result of the approximation by multilayer perceptron network is shown below together with the error.

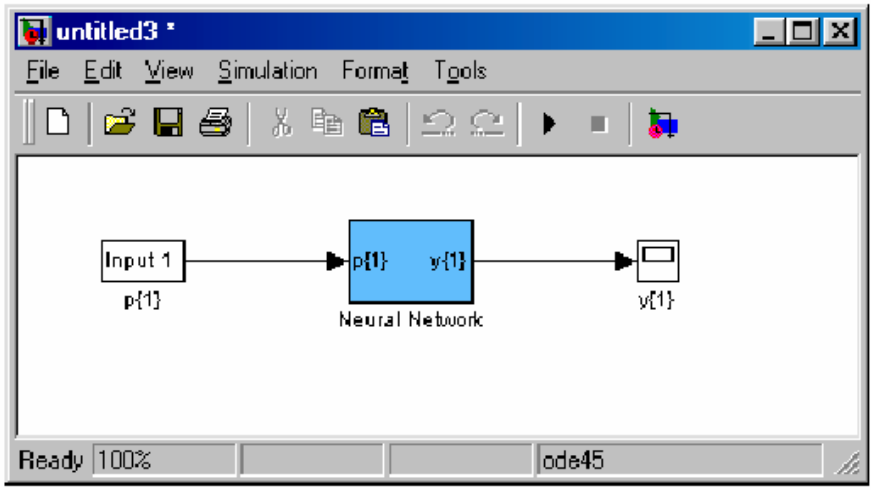
a=sim(net1,P); plot(P,a,P,a-T,P,T)

xlabel('Input x');ylabel('Output y');title('Nonlinear function f(x)')

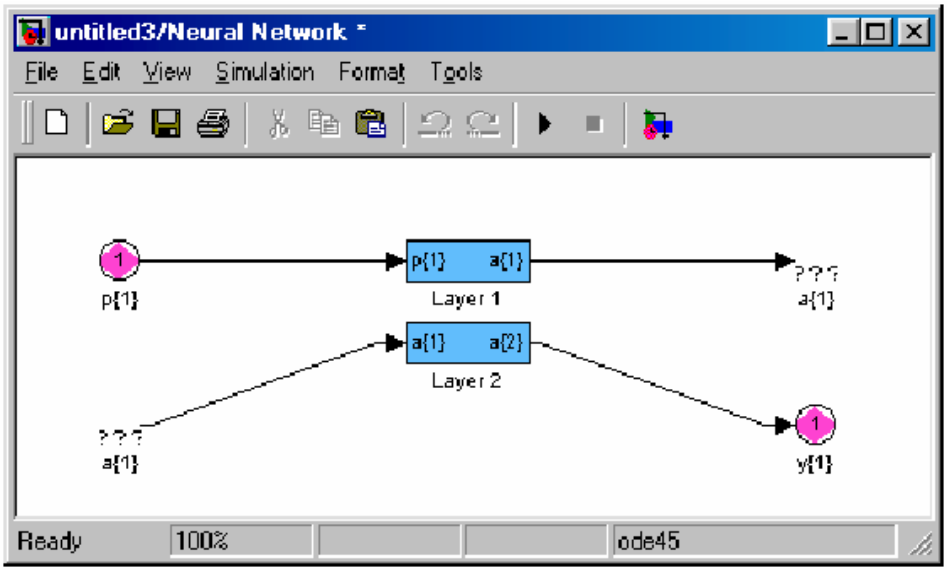


There is still some error, but let us proceed. Now we are ready to tackle the problem of solving the hybrid problem in SIMULINK. In order to move from the command side to SIMULINK use command *gensim*. This will transfer the information about the neural network to SIMULINK and at the same time it automatically generates a SIMULINK file with the neural network block. The second argument is used to define sampling time. For continuous sampling the value is –1.

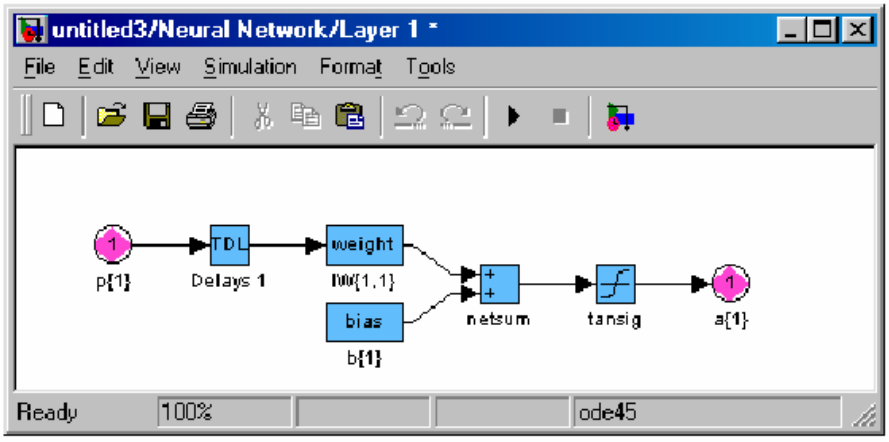
gensim(net1,-1)



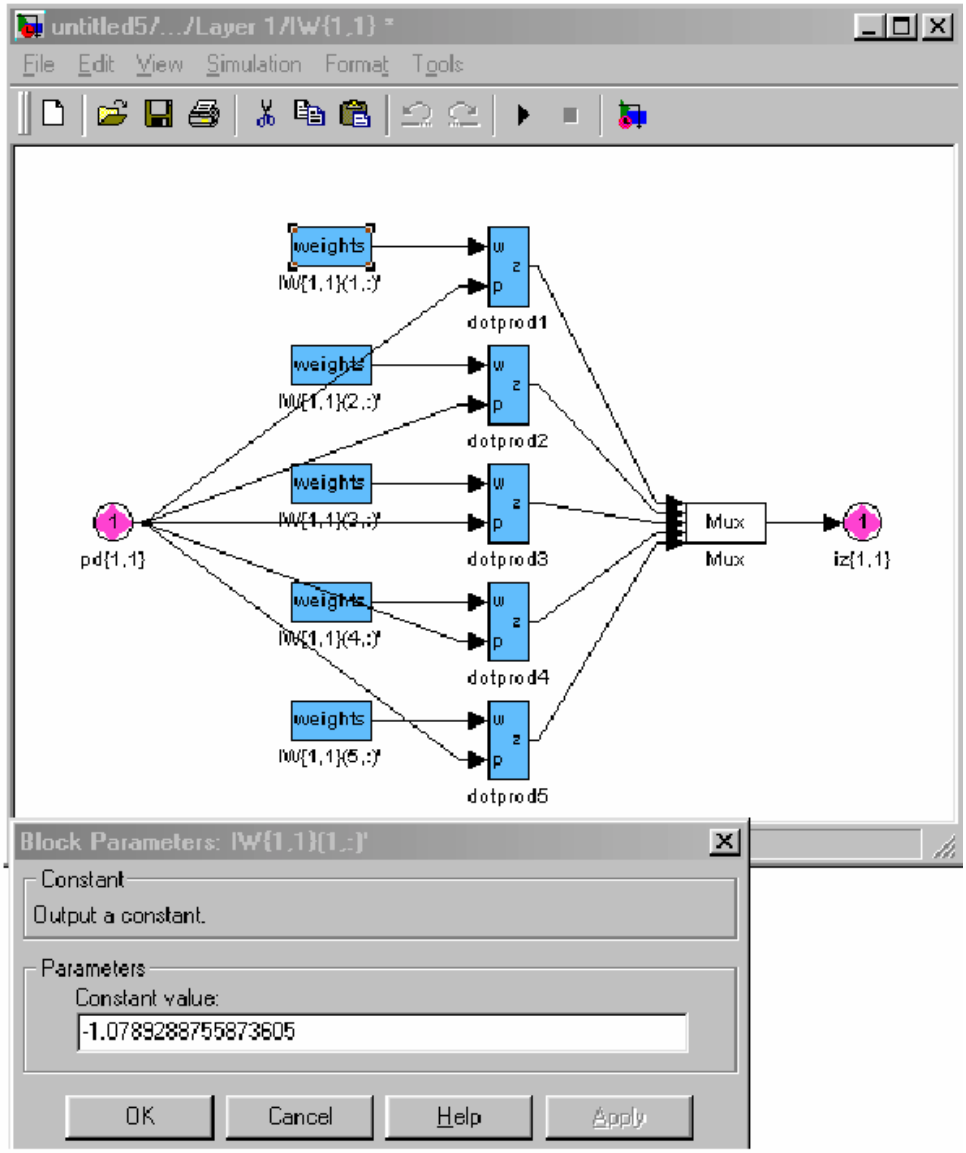
If you open the Neural Network block, you can see more details.



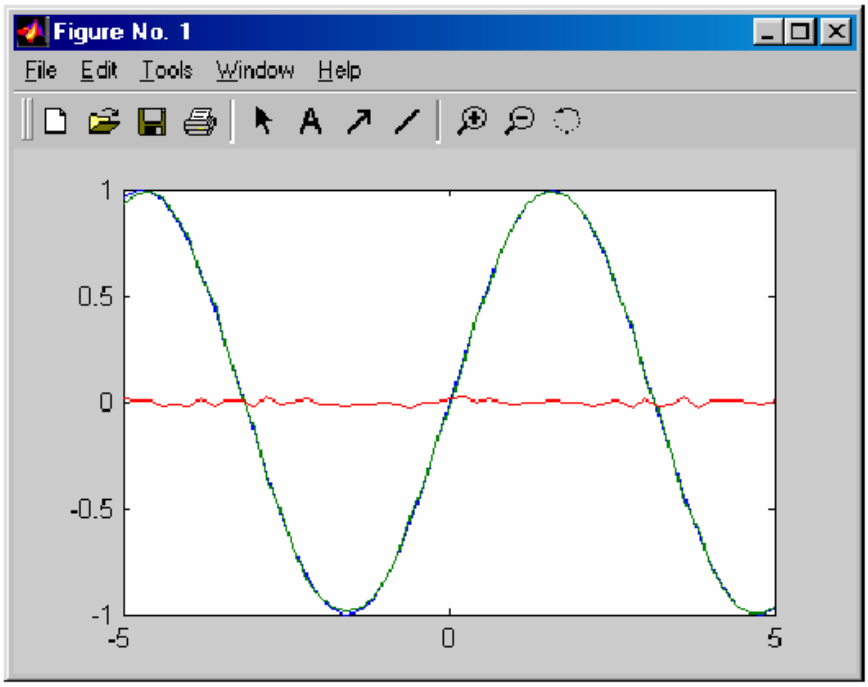
Open Layer 1. You'll see the usual block diagram representation of Neural Network Toolbox. In our examples, Delays block is unity map, i.e., no delays are in use.



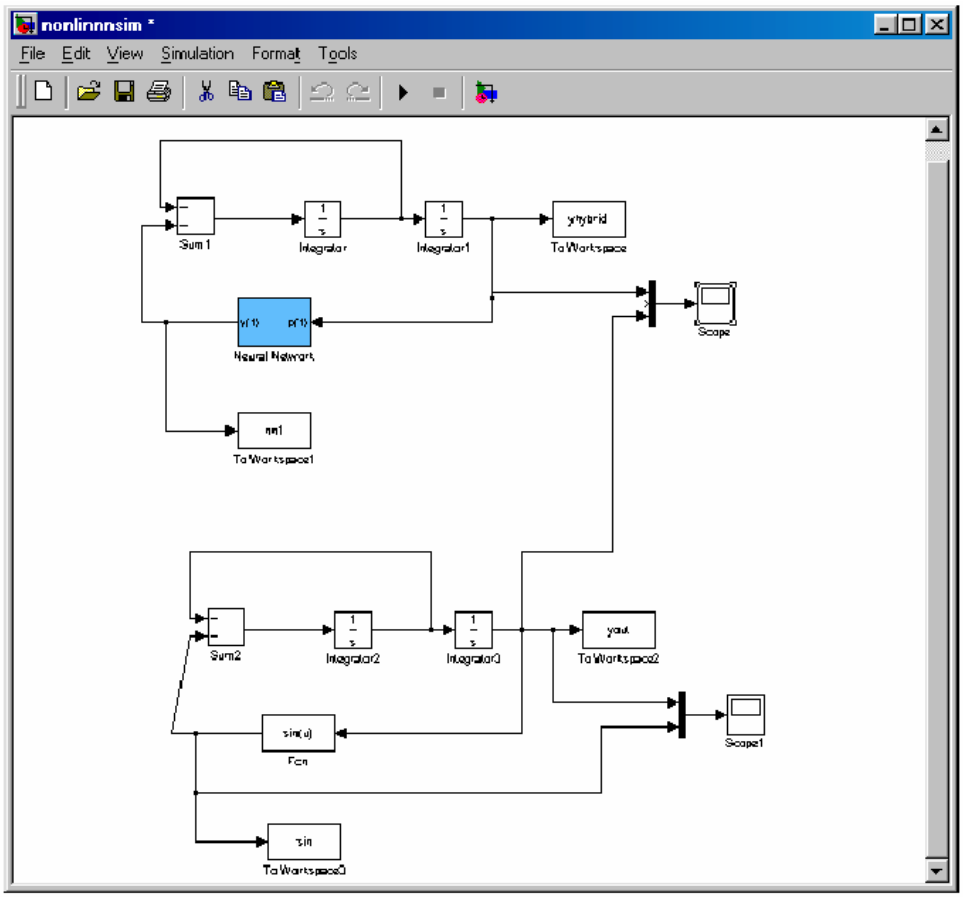
Open also the weight block. The figure that you see is shown below. The number of nodes has been reduced to 5, so that the figure fits on the page. In the example we have 20.



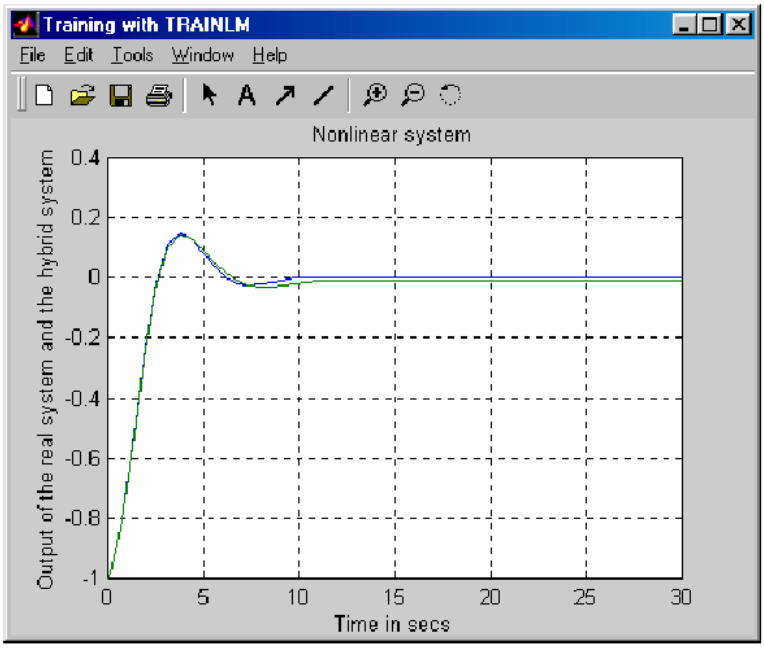
To convince ourselves that the block generated with *gensim* really approximates the given data, we'll feed in values of x in the range [-5,5]. Use the following SIMULINK configuration.



Now simulate the system.

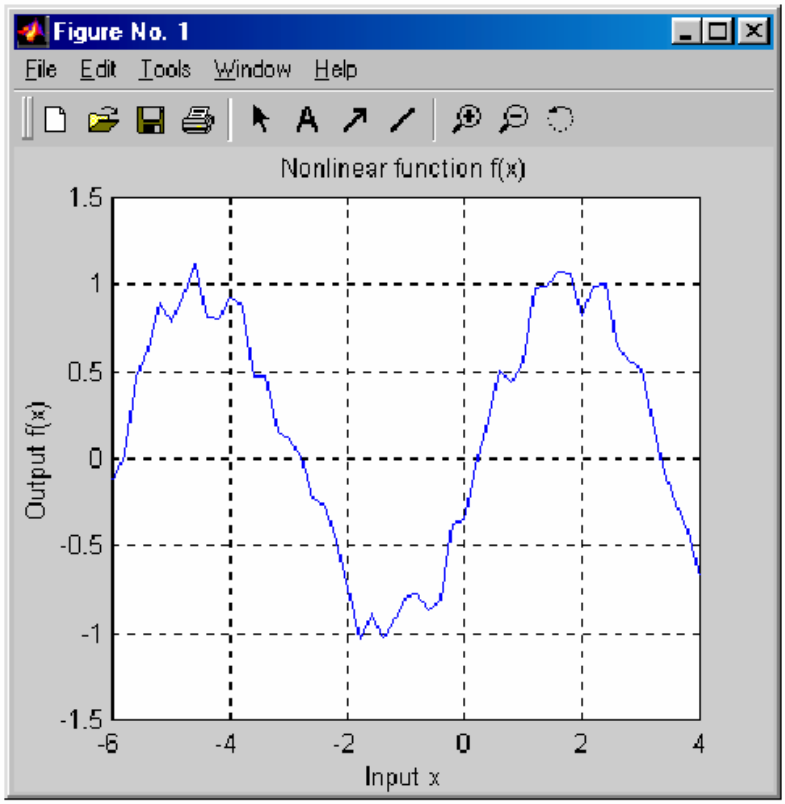


The result is plotted below.
plot(tout,yout,tout,yhybrid)
title('Nonlinear system'); xlabel('Time in secs');
ylabel('Output of the real system and the hybrid system'); grid;

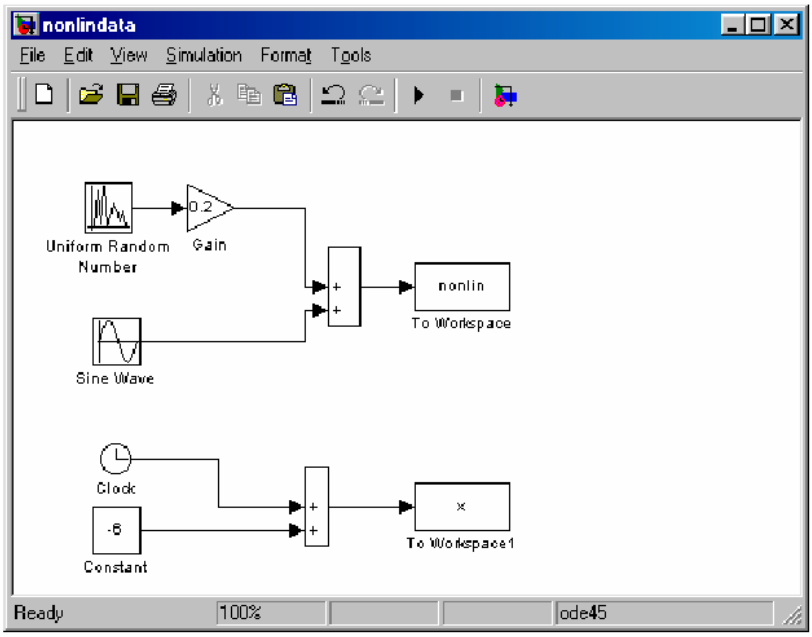


Careful study shows that some error remains. Further improvement can be obtained either by adding the network size or e.g. tightening the error tolerance bound.

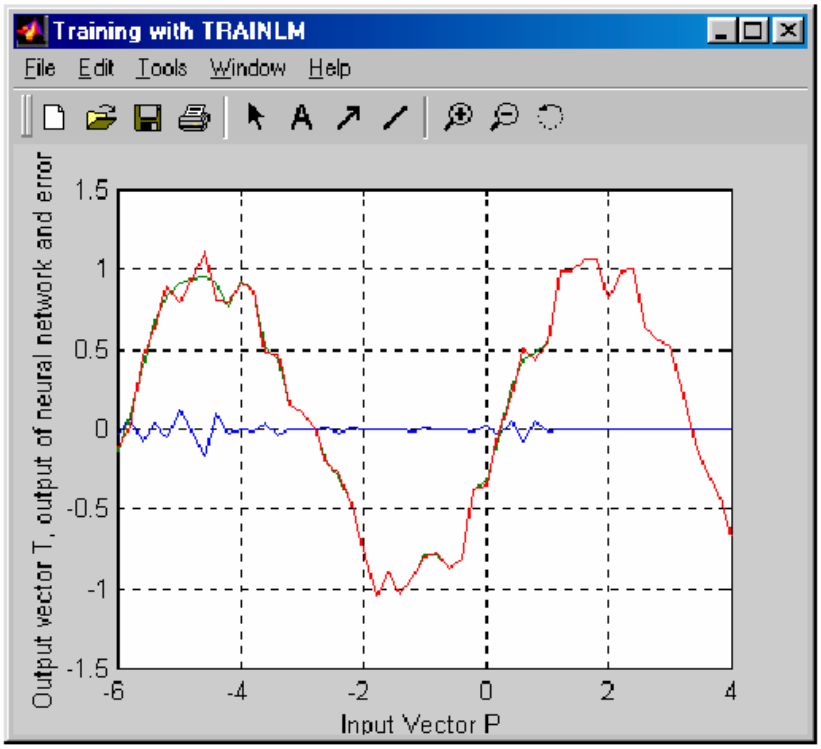
If the data is noise corrupted the procedure is the same, except that it is recommended that data preprocessing is performed. The data is shown below.



The following SIMULINK model shown below is configured to simulate noise-corrupted data.



The result of the approximation by multilayer perceptron network is shown below together with the error.

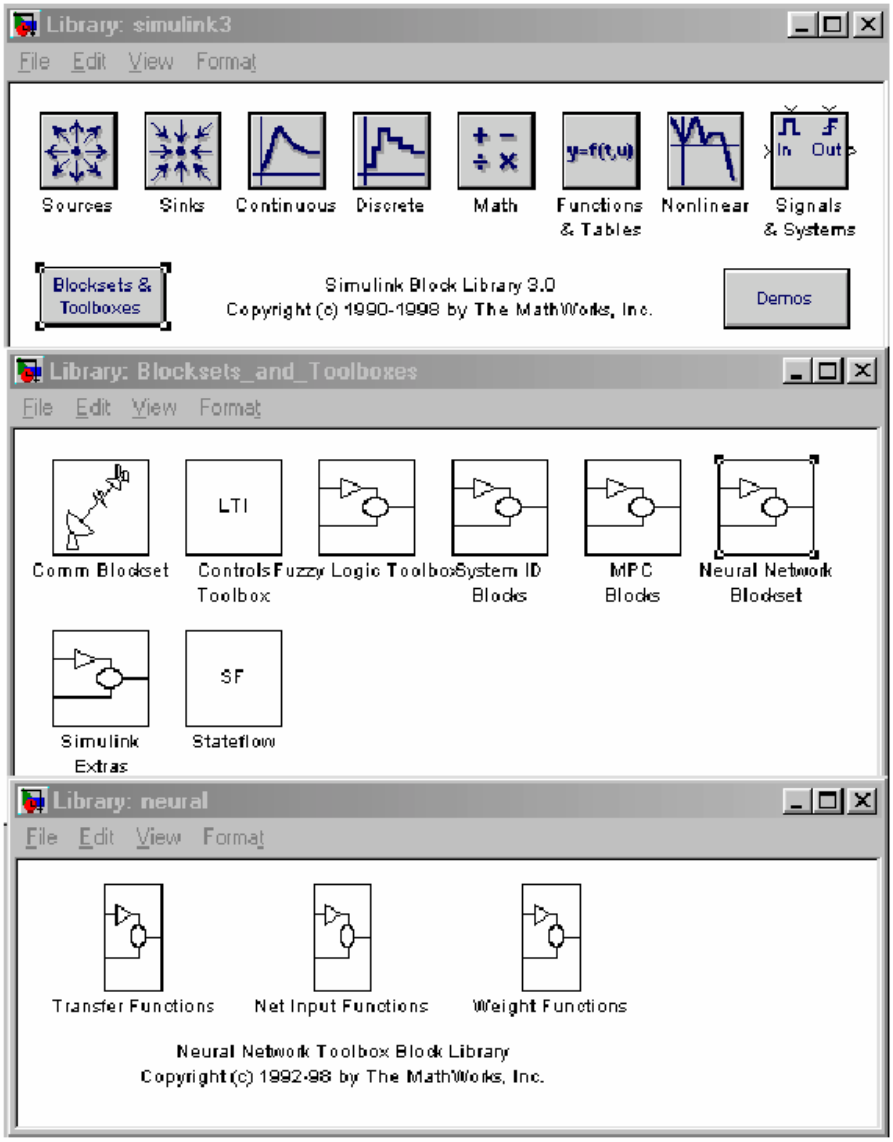


As can be seen in the figure, there is still error, but since the given data is noisy, we are reasonably happy with the fit. Preprocessing of data, using e.g. filtering, should be carried out before neural network fitting.

Now we are ready to tackle the problem of solving the hybrid problem in SIMULINK. Again use command *gensim.*

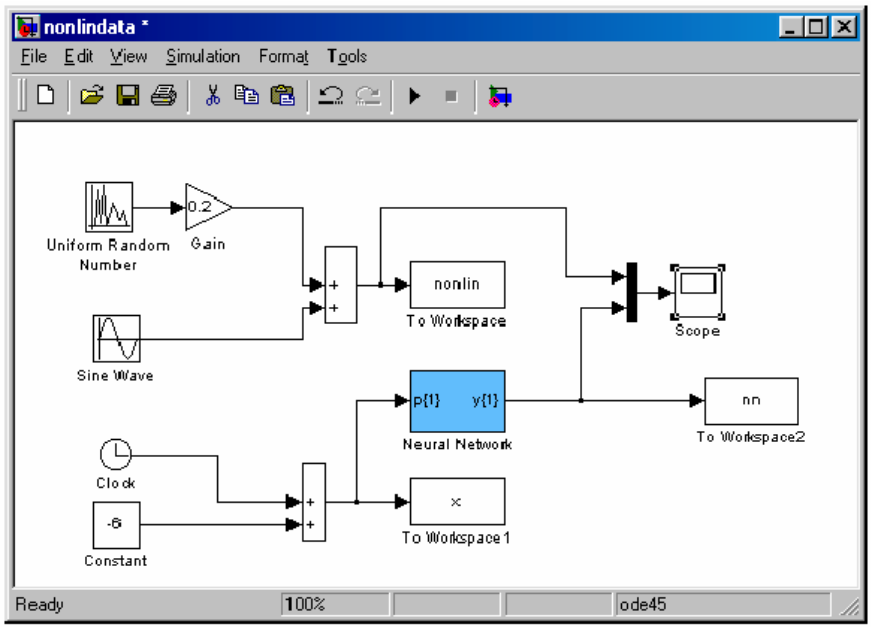
gensim(net1,-1)

REMARK: The other way to set up your own neural network from the blocks is to look under Blocksets & Toolboxes.

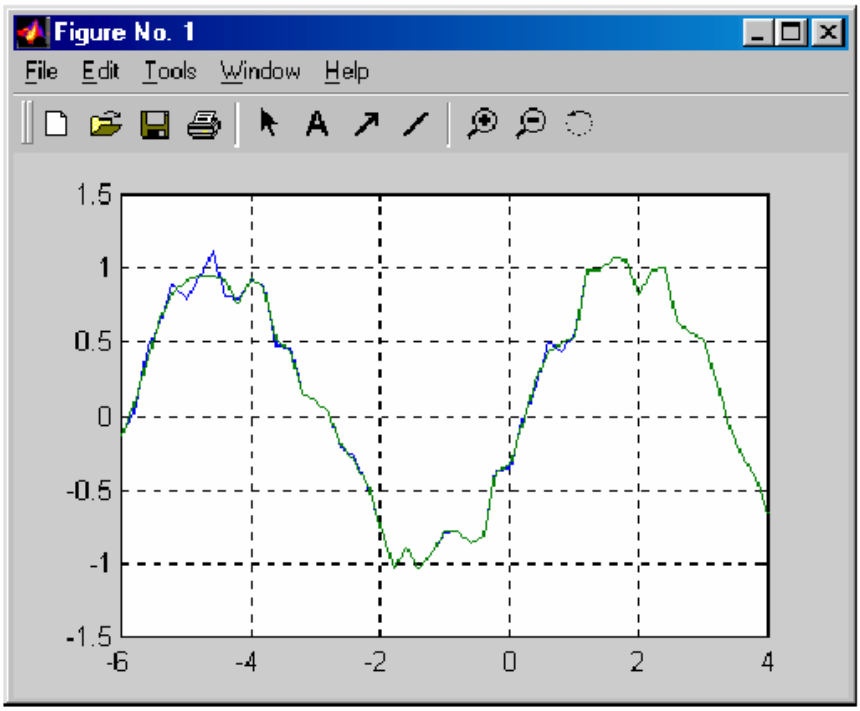


The basic blocks can be found in the three block libraries: Transfer Functions, Net Input Functions and Weight Functions. Parameters for the blocks have to be generated in the Command window. Then the SIMULINK configuration is performed. This seems to be quite tedious and the added freedom does not seem to be worth the trouble. Perhaps, in the coming versions of the Toolbox, a more user-friendly and flexible GUI is provided. Currently, the use of *gensim* command is recommended.

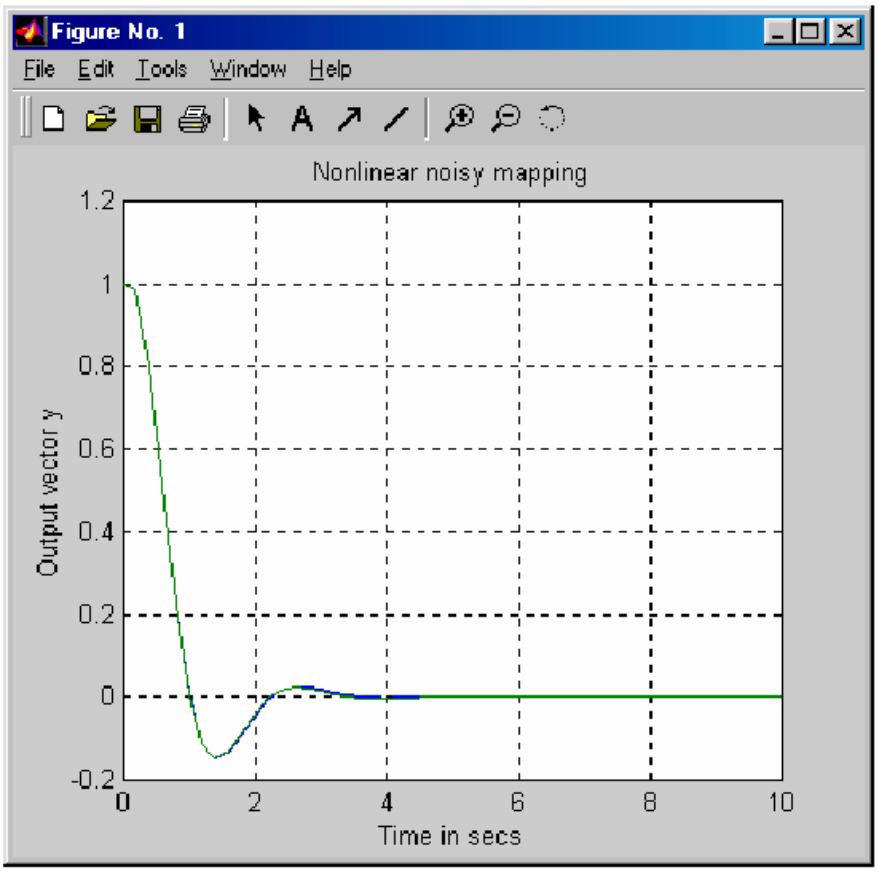
To convince ourselves that the block generated with *gensim* really approximates the given data, we'll feed in values of x in the range [-6,6]. Use the following SIMULINK configuration for comparison.



The result is quite satisfactory.



Now simulate the system.



Simulation result looks quite reasonable.

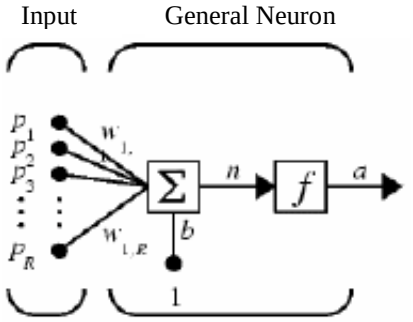
3.4. Backpropagation Network

The following text gives a short introduction to backpropagation neural networks. Same information is available also in Matlab.

Backpropagation was created by generalizing the Widrow-Hoff learning rule to multiple-layer networks and nonlinear differentiable transfer functions. Input vectors and the corresponding target vectors are used to train a network until it can approximate a function, associate input vectors with specific output vectors, or classify input vectors in an appropriate way as defined by you. Networks with biases, a sigmoid layer, and a linear output layer are capable of approximating any function with a finite number of discontinuities.

Properly trained backpropagation networks tend to give reasonable answers when presented with inputs that they have never seen. Typically, a new input leads to an output similar to the correct output for input vectors used in training that are similar to the new input being presented. This generalization property makes it possible to train a network on a representative set of input/target pairs and get good results without training the network on all possible input/output pairs.

The multilayer feedforward network is most commonly used with the backpropagation algorithm.
An elementary neuron with R inputs is shown below. Each input is weighted with an appropriate w. The sum of the weighted inputs and the bias forms the input to the transfer function *f*. Neurons may use any differentiable transfer function f to generate their output:

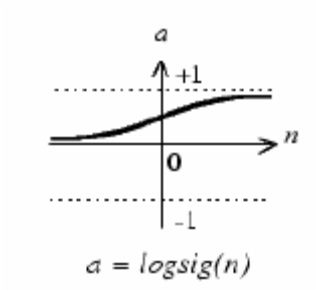


$a = f(W_p = b)$

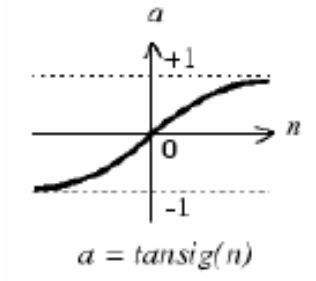
Where …

R-Number of elements in input vector

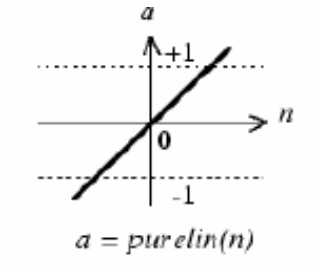
Multilayer networks often use the log-sigmoid transfer function *logsig*:



The function `logsig` generates outputs between 0 and 1 as the neuron's net input goes from negative to positive infinity. Alternatively, multilayer networks may use the tan-sigmoid transfer function `tansig`:

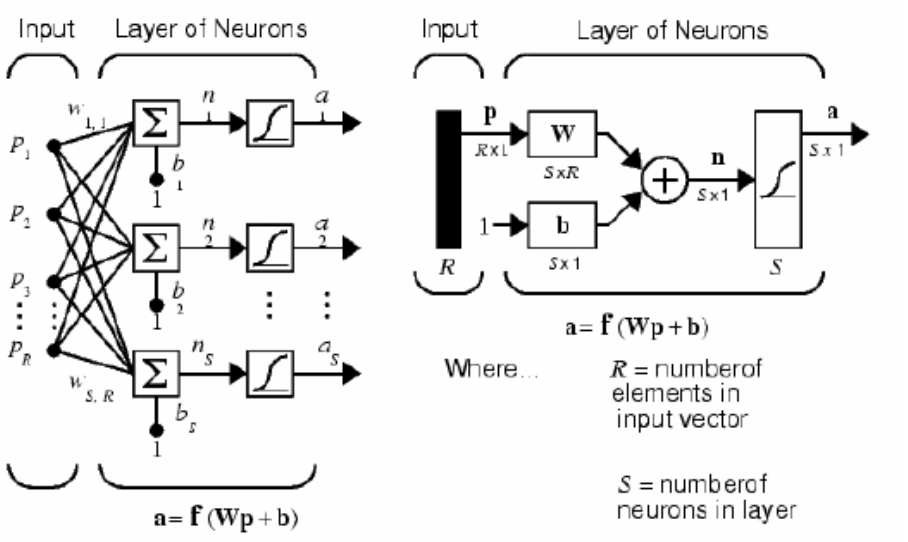


Occasionally, the linear transfer function *purelin* is used in backpropagation networks:

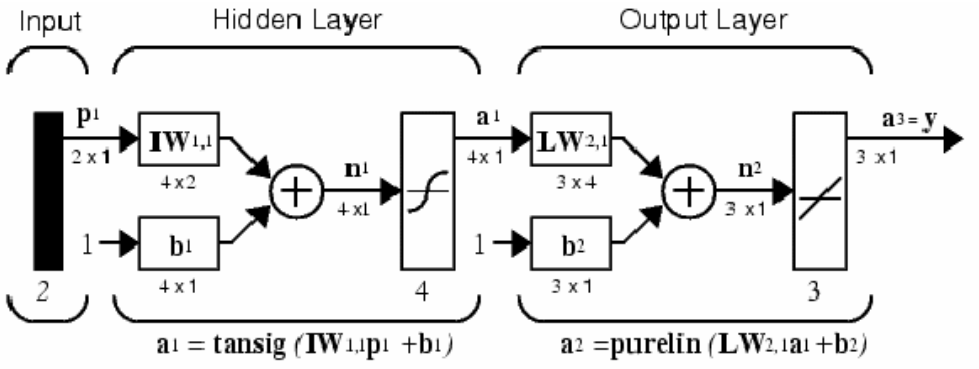


If the last layer of a multilayer network has sigmoid neurons, then the outputs of the network are limited to a small range. If linear output neurons are used the network outputs can take on any value.

A single-layer network of S *logsig* neurons having R inputs is shown below in full detail on the left and with a layer diagram on the right:



Feedforward networks often have one or more hidden layers of sigmoid neurons followed by an output layer of linear neurons. Multiple layers of neurons with nonlinear transfer functions allow the network to learn nonlinear and linear relationships between input and output vectors. The linear output layer lets the network produce values outside the range -1 to +1.



3.4.1. Backpropagation Algorithm

There are many variations of the backpropagation algorithm, several of which we discuss in this chapter. The simplest implementation of backpropagation learning updates the network weights and biases in the direction in which the performance function decreases most rapidly - the negative of the gradient. One iteration of this algorithm can be written,

$$X_{k+1} = X_k - \alpha_k g_k$$

,where X_k is a vector of current weights and biases, g_k is the current gradient, and α_k is the learning rate.

There are generally four steps in the training process:
1. Assemble the training data
2. Create the network object
3. Train the network

4. Simulate the network response to new inputs

3.4.2. Assembling the data

The following code creates a training set of inputs p and targets t for this example

```
p = [-1 -1 2 2;0 5 0 5];
t = [-1 -1 1 1];
```

When the training data is created, you should save a part of it for checking/validation purposes. Before training, it is often useful to scale the inputs and targets so that they always fall within a specified range. The function *premnmx* can be used to scale inputs and targets so that they fall in the range [-1,1].

Another approach for scaling network inputs and targets is to normalize the mean and standard deviation of the training set. This procedure is implemented in the function *prestd*. It normalizes the inputs and targets so that they will have zero mean and unity standard deviation.

In some situations, the dimension of the input vector is large, but the components of the vectors are highly correlated (redundant). It is useful in this situation to reduce the dimension of the input vectors. An effective procedure for performing this operation is principal component analysis.

3.4.3. Creating the network:

The first step in training a feedforward network is to create the network object. The function *newff* creates a feedforward network. It requires four inputs and returns the network object. The first input is an R by 2 matrix of minimum and maximum values for each of the R elements of the input vector. The second input is an array containing the sizes of each layer. The third input is a cell array containing the names of the transfer functions to be used in each layer. The final input contains the name of the training function to be used.

```
net=newff (minmax(p),[3,1],{'tansig','purelin'},'traingd');
```

(See: *help newff* for more information)

Before training a feedforward network, the weights and biases must be initialized. The newff command will automatically initialize the weights, but you may want to reinitialize them. This can be done with the command *init*.

3.4.4. Training:

Once the network weights and biases have been initialized, the network is ready for training. The network can be trained for function approximation (nonlinear regression), pattern association, or pattern classification. The training process requires a set of examples of proper network behavior - network inputs p and target outputs t. During training the weights and biases of the network are iteratively adjusted to minimize the network performance function *net.performFcn*. The default performance function for feedforward networks is mean square error mse - the average squared error between the network outputs a and the target outputs t. (Check *Matlab* help for more details on training)

Before starting the training procedure, you might want to change some of the training parameters. Otherwise default parameters is used:

```
net.trainParam.show = 50;
net.trainParam.lr = 0.05;
net.trainParam.epochs = 300;
net.trainParam.goal = 1e-5;
```

Command *train* trains the network using the training function (selected, when network was created) and default or manually set training parameters (as shown above):

```
[net,tr]=train(net,p,t);
```

Now the trained network can be simulated to obtain its response to the inputs in the training set.

```
a = sim(net,p)
a =
   -1.0010  -0.9989  1.0018  0.9985
```

After the trained network estimates the training data enough well, you should test it against the checking data. Also function *postreg* (post-training analysis) can be used in studying the results.

3.4.5. About training algorithms:

The batch steepest descent training function is *traingd*. The weights and biases are updated in the direction of the negative gradient of the performance function.

Another batch algorithm for feedforward networks that often provides faster convergence is *traingdm*, steepest descent with momentum. Momentum allows a network to respond not only to the local gradient, but also to recent trends in the error surface. Acting like a low-pass filter, momentum allows the network to ignore small features in the error surface. Without momentum a network may get stuck in a shallow local minimum. With momentum a network can slide through such a minimum.

The previous methods are often too slow for practical problems. Faster, high performance algorithms can converge from ten to one hundred times faster than the algorithms discussed previously.

These faster algorithms fall into two main categories. The first category uses heuristic techniques, which were developed from an analysis of the performance of the standard steepest descent algorithm. In Matlab there are variable learning rate backpropagation, *traingda*; and resilient backpropagation *trainrp* that represent this group.

The second category of fast algorithms uses standard numerical optimization techniques. There are three types of numerical optimization techniques for neural network training: conjugate gradient (*traincgf, traincgp, traincgb, trainscg*), quasi-Newton (*trainbfg, trainoss*), and Levenberg-Marquardt (*trainlm*).

3.4.6. Example:

(*Matlab example*) Data pre- and post-processing:

We want to design an instrument that can determine serum cholesterol levels from measurements of spectral content of a blood sample. We have a total of 264 patients for which we have measurements of 21 wavelengths of the spectrum. For the same patients we also have measurements of hdl, ldl, and vldl cholesterol levels, based on serum separation. The first step is to load the data into the workspace and perform a principal component analysis.

```
>> load choles_all
>> [pn,meanp,stdp,tn,meant,stdt] = prestd(p,t);
>> [ptrans,transMat] = prepca(pn,0.001);
```

Here we have conservatively retained those principal components which account for 99.9% of the variation in the data set. Let's check the size of the transformed data.

```
[R,Q] = size(ptrans)
R =
     4
Q =
   264
```

There was apparently significant redundancy in the data set, since the principal component analysis has reduced the size of the input vectors from 21 to 4.

The next step is to divide the data up into training, validation and test subsets. We will take one fourth of the data for the validation set, one fourth for the test set and one half for the training set. We pick the sets as equally spaced points throughout the original data.

```
Iitst = 2:4:Q;
iival = 4:4:Q;
iitr = [1:4:Q 3:4:Q];
val.P = ptrans(:,iival); val.T = tn(:,iival);
test.P = ptrans(:,iitst); test.T = tn(:,iitst);
ptr = ptrans(:,iitr); ttr = tn(:,iitr);
```

We are now ready to create a network and train it. For this example, we will try a two-layer network, with tansigmoid transfer function in the hidden layer and a linear transfer function in the output layer. This is a useful structure for function approximation (or regression) problems. As an initial guess, we use five neurons in the hidden layer. The network should have three output neurons since there are three targets. We will use the Levenberg-Marquardt algorithm for training.

```
Net = newff(minmax(ptr),[5 3],{'tansig' 'purelin'},'trainlm');
[net,tr]=train(net,ptr,ttr,[],[],val,test);
```

The training stopped after 15 iterations because the validation error increased. It is a useful diagnostic tool to plot the training, validation and test errors to check the progress of training. We can do that with the following commands:

```
plot(tr.epoch,tr.perf,tr.epoch,tr.vperf,tr.epoch,tr.tperf)

legend('Training','Validation','Test',-1);
```

```
ylabel('Squared Error'); xlabel('Epoch')
```
The result here is reasonable, since the test set error and the validation set error have similar characteristics, and it doesn't appear that any significant overfitting has occurred.

The next step is to perform some analysis of the network response. We will put the entire data set through the network (training, validation and test) and will perform a linear regression between the network outputs and the corresponding targets. First we need to unnormalize the network outputs.

```
An = sim(net,ptrans);
a = poststd(an,meant,stdt);
for i=1:3
figure(i)
[m(i),b(i),r(i)] = postreg(a(i,☺,t(i,☺);
end
```

In this case, we have three outputs, so we perform three regressions.

The first two outputs seem to track the targets reasonably well (this is a difficult problem), and the R-values are almost 0.9. The third output (vldl levels) is not well modeled. We probably need to work more on that problem. We might go on to try other network architectures (more hidden layer neurons), or to try Bayesian regularization instead of early stopping for our training technique. Of course there is also the possibility that vldl levels cannot be accurately computed based on the given spectral components.

3.4.7. Exercise: Run demo *nnd11gn*

```
>> nnd11gn
```

How increasing of hidden layer neurons affects to function approximation? Are there any side-effects if number of hidden layer neurons is high?

3.5. Laboratory Work № 3 (LAB. 3)

Construct a neural net, which approximates function $y(x) = 2x^3 + 1$. Use in training

a) Gradient method (`traingd`)
b) Fletcher-Reeves -method (`traincgf`)
c) kvasi-Newton -method (`trainbfg`)
d) Levenberg-Marquadt -method (`trainlm`)

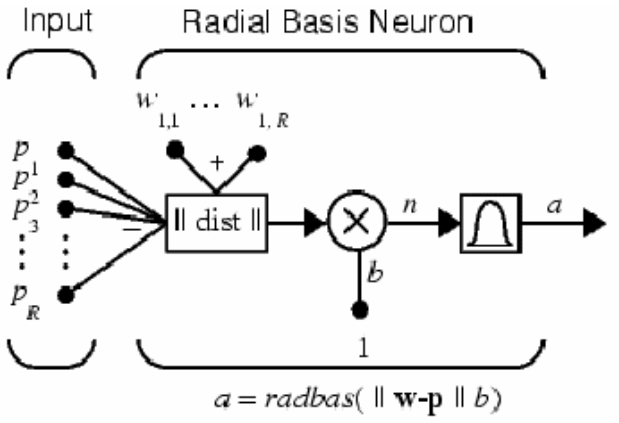
Compare convergence, convergence time and accuracy when you have different number of epochs, different accuracy and different number of training data points. You can also try to change the number of neurons in input layer.

3.6. Radial Basic Networks:

Radial basis networks may require more neurons than standard feed-forward backpropagation networks, but often they can be designed in a fraction of the time it takes to train standard feedforward networks. They work best when many training vectors are available.

3.6.1. Neuron Model:

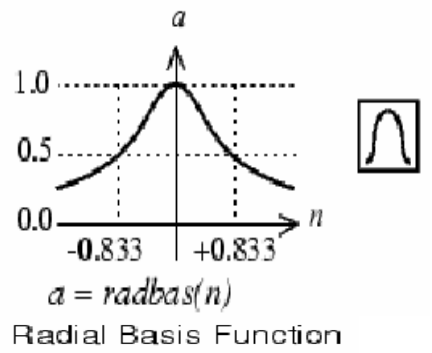
Here is a radial basis network with R inputs:



Notice that the expression for the net input of a radbas neuron is different from that of neurons in previous chapters. Here the net input to the radbas transfer function is the vector distance between its weight vector **w** and the input vector **p**, multiplied by the bias *b*. (The $\|dist\|$ box in this figure accepts the input vector p and the single row input weight matrix, and produces the dot product of the two.)

The transfer function for a radial basis neuron is:

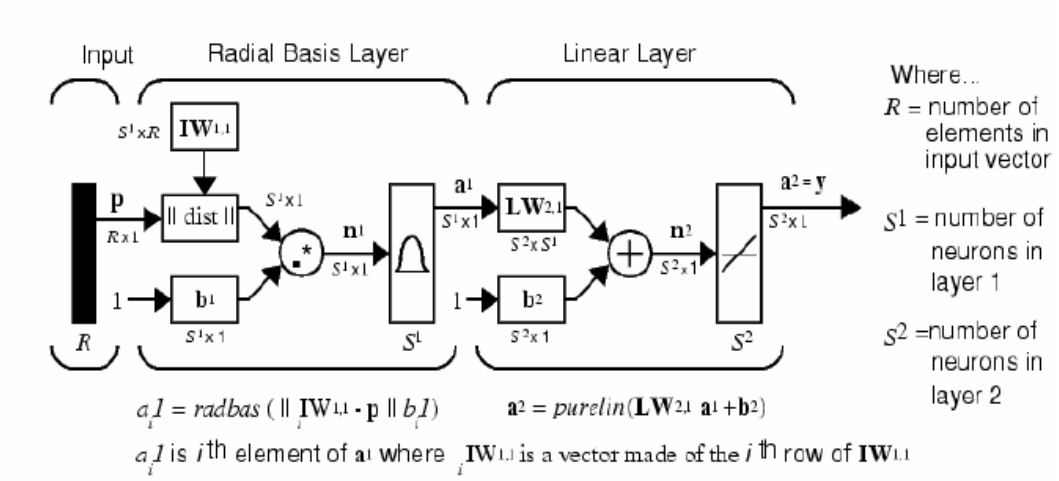
$$radbas(n) = e^{-n^2}$$



The radial basis function has a maximum of 1 when its input is 0. As the distance between w and p decreases, the output increases. Thus, a radial basis neuron acts as a detector that produces 1 whenever the input p is identical to its weight vector p. The bias b allows the sensitivity of the radbas neuron to be adjusted. For example, if a neuron had a bias of 0.1 it would output 0.5 for any input vector p at vector distance of 8.326 (0.8326/b) from its weight vector w.

3.6.2. Network Architecture

Radial basis networks consist of two layers: a hidden radial basis layer of S1 neurons, and an output linear layer of S2 neurons.



The $\|dist\|$ box in this figure accepts the input vector p and the input weight matrix *IW1,1*, and produces a vector having *S1* elements. The elements are the distances between the input vector and vectors *iIW1,1* formed from the rows of the input weight matrix.

The bias vector *b1* and the output of $\|dist\|$ are combined with the MATLAB operation .* , which does element-by-element multiplication.

We can understand how this network behaves by following an input vector *p* through the network to the output *a2*. If we present an input vector to such a network, each neuron in the radial basis layer will output a value according to how close the input vector is to each neuron's weight vector.

Thus, radial basis neurons with weight vectors quite different from the input vector *p* have outputs near zero. These small outputs have only a negligible effect on the linear output neurons.

In contrast, a radial basis neuron with a weight vector close to the input vector *p* produces a value near 1. If a neuron has an output of 1 its output weights in the second layer pass their values to the linear neurons in the second layer.

In fact, if only one radial basis neuron had an output of 1, and all others had outputs of 0's (or very close to 0), the output of the linear layer would be the active neuron's output weights. This would, however, be an extreme case. Typically several neurons are always firing, to varying degrees.

Now let us look in detail at how the first layer operates. Each neuron's weighted input is the distance between the input vector and its weight vector, calculated with dist. Each neuron's net input is the element-by-element product of its weighted input with its bias, calculated with netprod. Each neurons' output is its net input passed through *radbas*. If a neuron's weight vector is equal to the input vector (transposed), its weighted input is 0, its net input is 0, and its output is 1. If a neuron's weight vector is a distance of spread from the input vector, its weighted input is spread, its net input is sqrt(-log(.5)) (or 0.8326), therefore its output is 0.5.

3.6.3. Design procedure:

Radial basis networks can be designed with the function newrbe and newrb . newrbe function can produce a network with zero error on training vectors. It is called in the following way:

```
net = newrbe(P,T,SPREAD)
```

The function `newrbe` takes matrices of input vectors P and target vectors T, and a spread constant SPREAD for the radial basis layer, and returns a network with weights and biases such that the outputs are exactly T when the inputs are P. This function `newrbe` creates as many *radbas* neurons as there are input vectors in P, and sets the first-layer weights to P'. Thus, we have a layer of *radbas* neurons in which each neuron acts as a detector for a different input vector. If there are Q input vectors, then there will be Q. The drawback to newrbe is that it produces a network with as many hidden neurons as there are inputvectors. For this reason, newrbe does not return an acceptable solution when many input vectors are needed to properly define a network, as is typically the case.

The function `newrb` iteratively creates a radial basis network one neuron at a time. It is thus more efficient design method than `newrbe`. Neurons are added to the network until the sum-squared error falls beneath an error goal or a maximum number of neurons has been reached. The call for this function is:

```
net = newrb(P,T,GOAL,SPREAD)
```

The function `newrb` takes matrices of input and target vectors, P and T, and design parameters GOAL and, SPREAD, and returns the desired network.

The design method of `newrb` is similar to that of `newrbe`. The difference is that `newrb` creates neurons one at a time. At each iteration the input vector that results in lowering the network error the most, is used to create a *radbas* neuron. The error of the new network is checked, and if low enough `newrb` is finished. Otherwise the next neuron is added. This procedure is repeated until the error goal is met, or the maximum number of neurons is reached.

As with `newrbe`, it is important that the spread parameter be large enough that the *radbas* neurons respond to overlapping regions of the input space, but not so large that all the neurons respond in essentially the same manner.

Why not always use a radial basis network instead of a standard feed-forward network? Radial basis vnetworks, even when designed efficiently with `newrbe`, tend to have many times more neurons than a comparable feed-forward network with *tansig* or *logsig* neurons in the hidden layer.

This is because sigmoid neurons can have outputs over a large region of the input space, while *radbas* neurons only respond to relatively small regions of the input space. The result is that the larger the input space (in terms of number of inputs, and the ranges those inputs vary over) the more *radbas* neurons required.

On the other hand, designing a radial basis network often takes much less time than training a sigmoid/linear network, and can sometimes result in fewer neurons being used.

3.7. Laboratory Work № 4 (LAB. 4)

1) Formulate Radial Base –network, which approximates function

$$y = f(x) = 2\sin(5x-1) + \cos(3x-1),$$

when $x \in [-1,1]$. Give values 0.01, 1 and 100 to transfer function width parameter. How does the approximation change?

2) Simulate the behavior of system

$$\ddot{x} = -0.6\dot{x} + f(x),$$

where $f(x)$ is same than in problem 1, with initial values

$$x(0) = -0.2$$

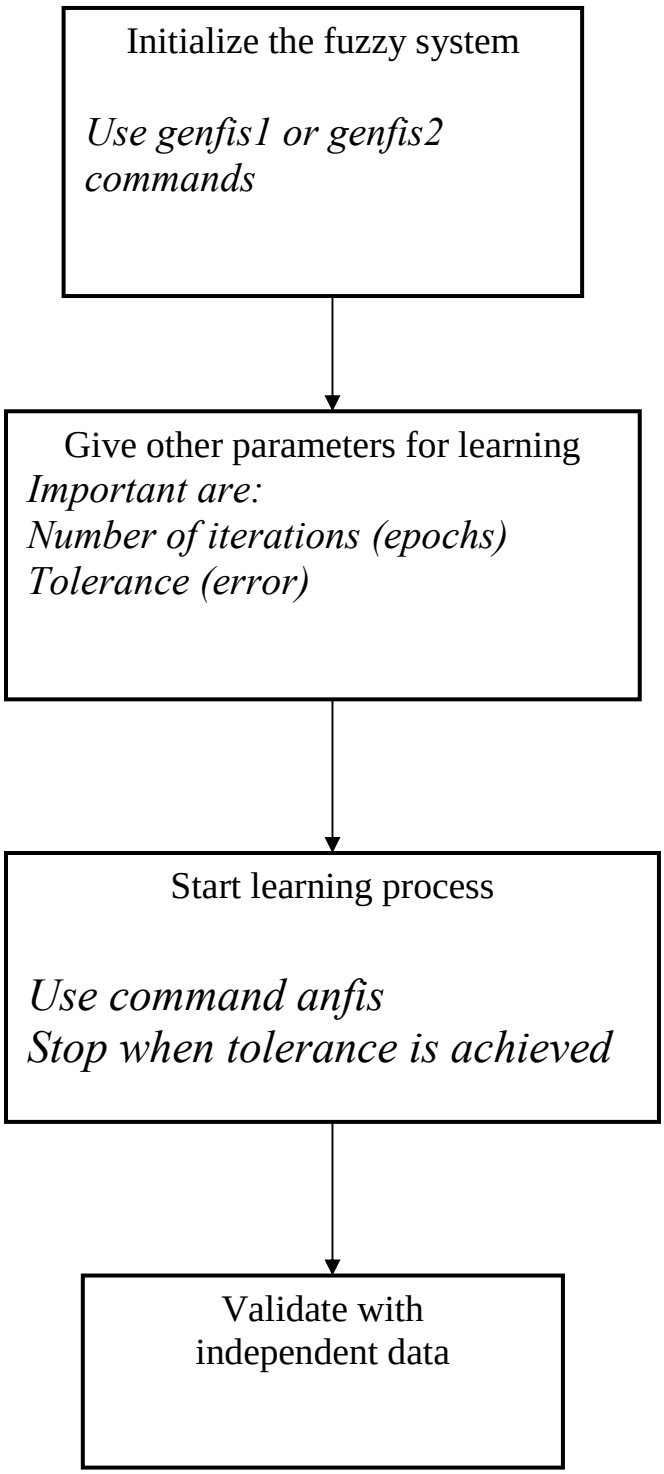
$$\dot{x}(0) = 0.5$$

After that, use the neural network approximation of *f* formulated in problem 1, and compare the behavior of system with exact function and system with neural network approximation. You can generate neural network block in SIMULINK by command *gensim*.

Chapter 4

ANFIS (Adaptive Neuro-Fuzzy Inference System)

Basic flow diagram of computations in ANFIS



ANFIS (Adaptive Neuro-Fuzzy Inference System) –method is used as a teaching method for Sugeno-type fuzzy systems. System parameters are identified by the aid if ANFIS. Usually the number and type of fuzzy system membership functions are defined by user when applying ANFIS. ANFIS –method is a hybrid method, which consists two parts: gradient method is applied to calculation of input membership function parameters, and least square method is applied to calculation of output function parameters.
The restrictions of Matlab ANFIS-method:

- only Sugeno-type decision method is available
- there can be only one output
- defuzzification method is weighted mean value

In Fuzzy Control Toolbox a useful command called ***anfis*** exists. This provides an optimization scheme to find the parameters in the fuzzy system that best fit the data. It is explained in the Toolbox manual that since most (not all) optimization algorithms require computation of the gradient, this is done with a neural network. Then, in principle, any of the optimization schemes, say those in the MATLAB Optimization Toolbox, can be used.

4.1. GENFIS1 and ANFIS Commands

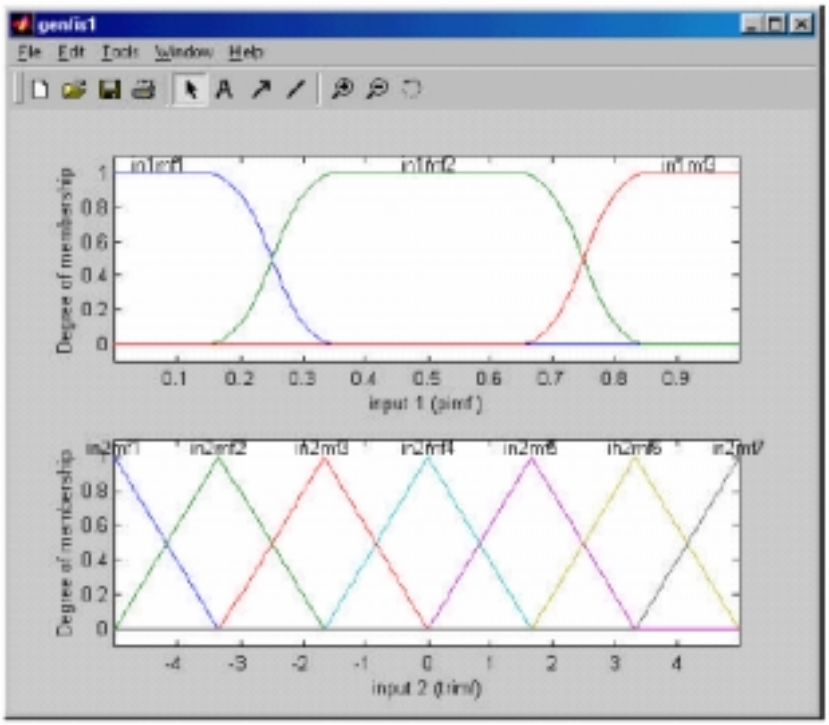
It is not clear at the beginning, what the initial fuzzy system should be, that is, the type and number of membership functions, command ***genfis1*** may be used. This command will go over the data in a crude way and find a good starting system.

> ***GENFIS1*** *Generate FIS matrix using generic method.*
>
> ***GENFIS1(DATA, MF_N, IN_MF_TYPE)*** *generates a fismatrix from training data DATA, using grid partition style. MF_N is a vector specifying the number of membership functions on all inputs. IN_MF_TYPE is a string array where each row specifies the MF type of an input variable.*
>
> *If MF_N is a number and/or IN_MF_TYPE is a single string, they will be used for all inputs. Note that MF_N and IN_MF_TYPE are pass directly to GENPARAM for generating input MF parameters.*
>
> *Default number of membership function MF_N is 2; default input membership function type is 'gbellmf'.*
>
> *For example:*

```
% Generate random data
    NumData = 1000;
    data = [rand(NumData,1) 10*rand(NumData,1)-5 rand(NumData,1)];

% Specify number and type of membership functions
    NumMf = [3 7];
    MfType = str2mat('pimf', 'trimf');
```

```
% Use GENFIS1 to generate initial membership functions
    FisMatrix = genfis1(data, NumMf, MfType);

% Plot the membership functions
    figure('name', 'genfis1', 'numbertitle', 'off');
    NumInput = size(data, 2) - 1;
    for i = 1:NumInput;
        subplot(NumInput, 1, i);
        plotmf(FisMatrix, 'input', i);
        xlabel(['input ' num2str(i) ' (' MfType(i, :) ')']);
    end

    See also GENPARAM, ANFIS.
```

SUMMARY: Use **genfis1** to generate **initial FIS system.** Use **anfis** to generate the best FIS system. **Anfis** uses the result from **genfis1** to start optimization.



REMARK: **Anfis** command has limitations. It can only be used for Sugeno systems and further:

- Constant and linear output membership functions only
- Single output derived by weighted average defuzzification

In MATLAB *help anfis* the following is said

When a specific Sugeno fuzzy inference system is used for fitting, we can call ANFIS with from 2 to 4 input arguments and it returns from 1 to 3 output arguments

- Unity weight for each rule

4.2. Exersise :

1) Make fuzzy approximation of function $y = 0.6\sin(\pi x) + 0.3\sin(3\pi x) + 0.1\sin(5\pi x)$. Use ANFIS-method. Compare fuzzy approximation with teaching and checking data.

2) Make fuzzy approximation of function $y = f(x) = -2x - x^2$, when $x \in$ [-10, 10]. Use ANFIS method. Compare fuzzy approximation and original function.

3) Simulate the behavior of system

$$\ddot{x} = -0.6\dot{x} + f(x)$$

$f(x)$ is same than in problem 2 above. Start by values

$$x(0) = -0.2$$

$$\dot{x}(0) = 0.5$$

Replace $f(x)$ by fuzzy approximation defined in problem 2 and compare the results. What happens, if the number of membership functions in fuzzy system is increased in approximation? Test for example 5, 15 and 60 membership functions. Is there any risk in increasing the number of membership functions?

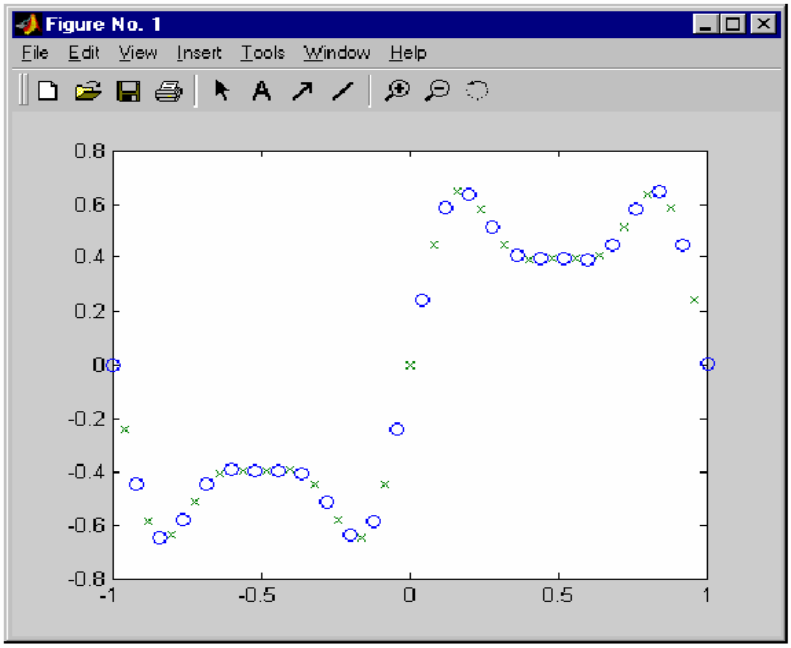
SOLUTIONS

1) Let's generate training and checking data first. Generate 51 input-output pairs between $x \in$ [-1, 1]:

```
>> numPts=51;
>> x=linspace(-1,1,numPts)';
>> y=0.6*sin(pi*x)+0.3*sin(3*pi*x)+0.1*sin(5*pi*x);
>> data=[x y];
>> trndata=data(1:2:numPts,:);
>> chkdata=data(2:2:numPts,:);
```

Plot training and checking data with symbols o and x respectively:

```
>> plot(trndata(:,1),trndata(:,2),'o',chkdata(:,1),chkdata(:,2),'x')
```



Next we configure fuzzy system for training. First we set the number and type of input membership functions:

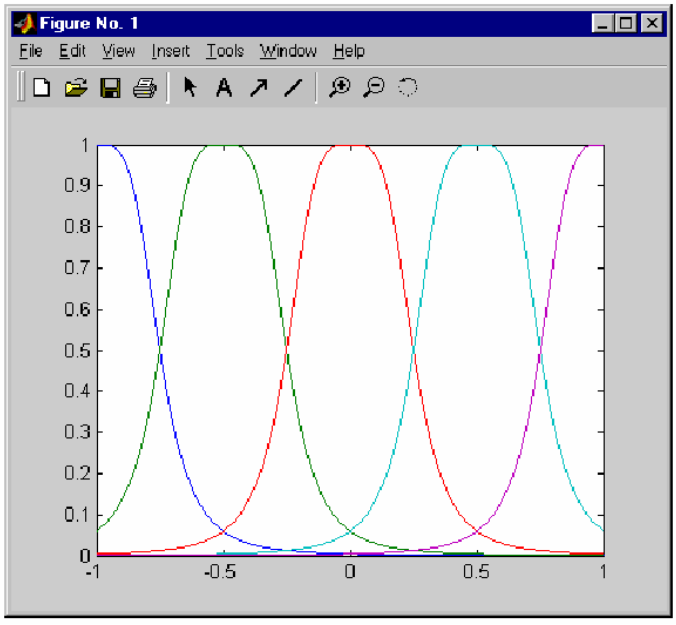
>> numMFs=5;
>> mfType='gbellmf';

Then FIS-matrix fismat is generated by command genfish1:

>> fismat=genfis1(trndata,numMFs,mfType);

It is possible to plot the membership functions by commands `plot` and `plotmf`:

```
>> [x,mf]=plotmf(fismat,'input',
>> plot(x,mf)
```



Then we do the ANFIS-training by 40 rounds (the meaning of NaN is that default settings are used concerning the issue, what information is shown in command window during the training).

```
>> numEpochs=40;
>>
[fismat1,trnErr,ss,fismat2,chkErr]=anfis(trndata,fismat,numEpoch
s,
NaN,chkdata);
```

When we use checking data (chkdata) as a option in anfis-command, Matlab use it to prevent overfitting during the time of training. Because of this, we have to use (another) checking data, what is not used in anfis-command, if we want totally independent checking data.

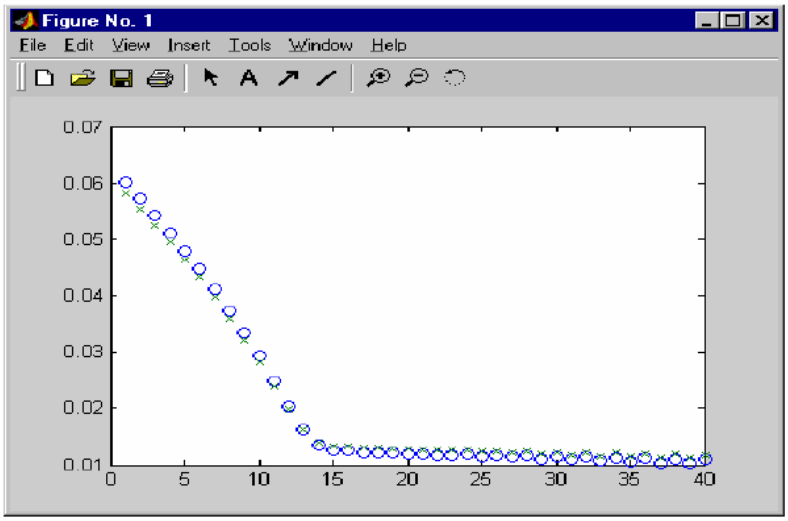
Check the training result:

```
>> trnOut=evalfis(trndata(:,1),fismat1);
>> trnRMSE=norm(trnOut-trndata(:,2))/sqrt(length(trnOut))

trnRMSE =

       0.0104
>> chkOut=evalfis(chkdata(:,1),fismat2);
>> chkRMSE=norm(chkOut-chkdata(:,2))/sqrt(length(chkOut))

chkRMSE =

      0.0112
```

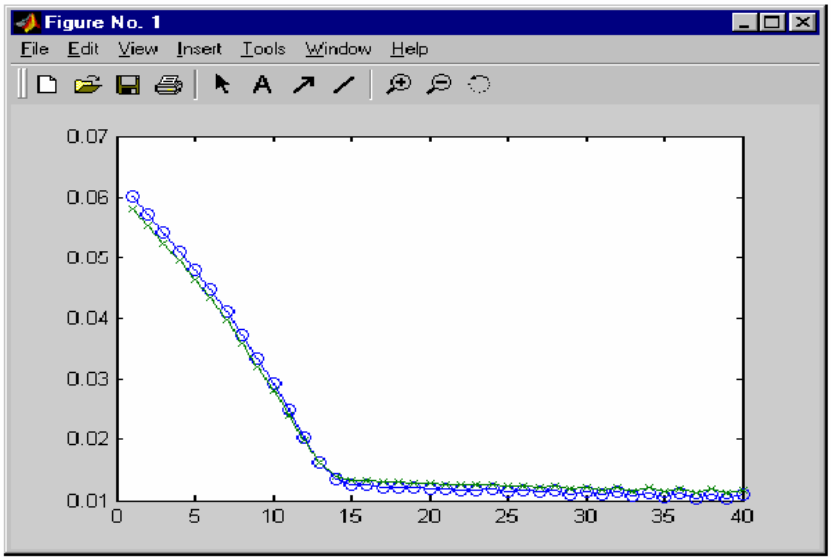
(evalfis calculates the output of fuzzy system, RMSE=root mean squared error)

Draw the error curves:

```
>> epoch=1:numEpochs;
>> plot(epoch,trnErr,'o',epoch,chkErr,'x')
>> hold on;
```

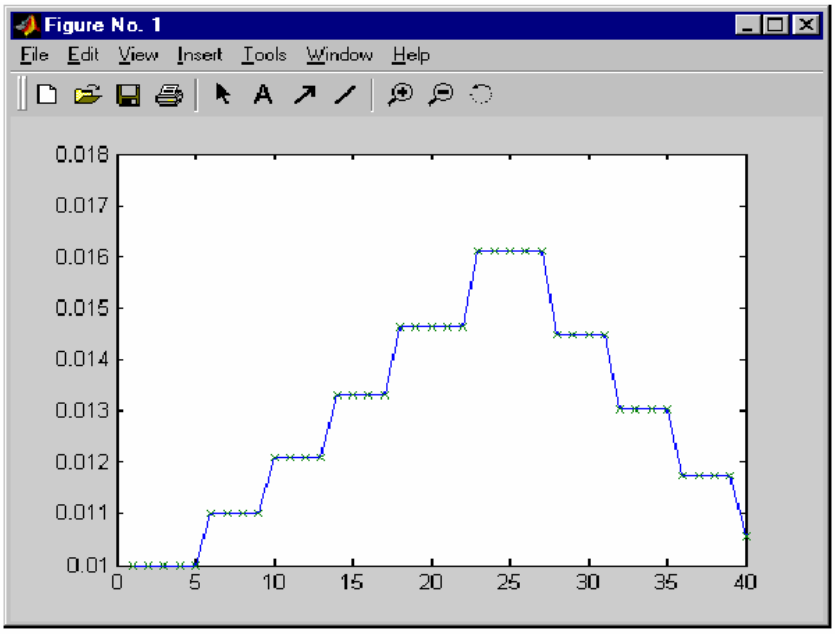


```
>> hold on;
>> plot(epoch,[trnErr chkErr])
>> hold off;
```



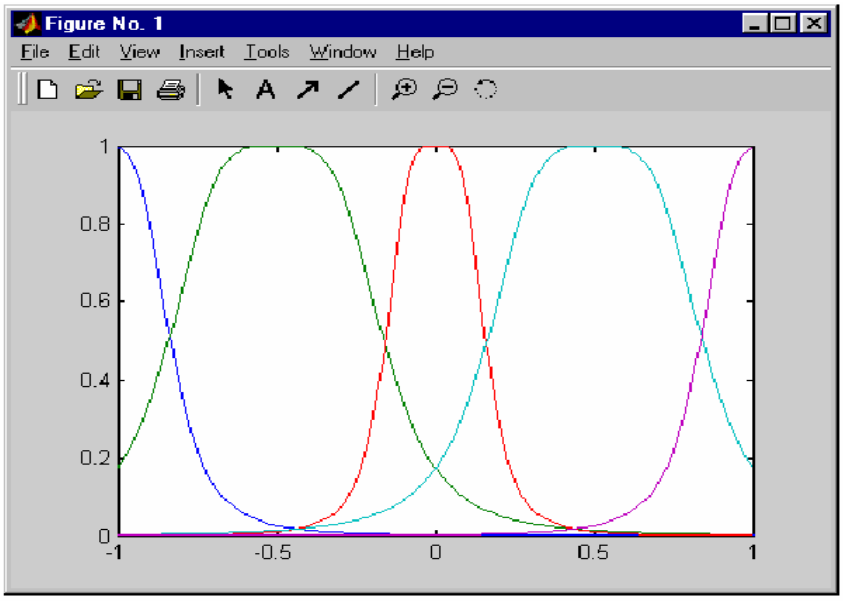
Step size during the training:

```
>> plot(epoch,ss,'-',epoch,ss,'x')
```



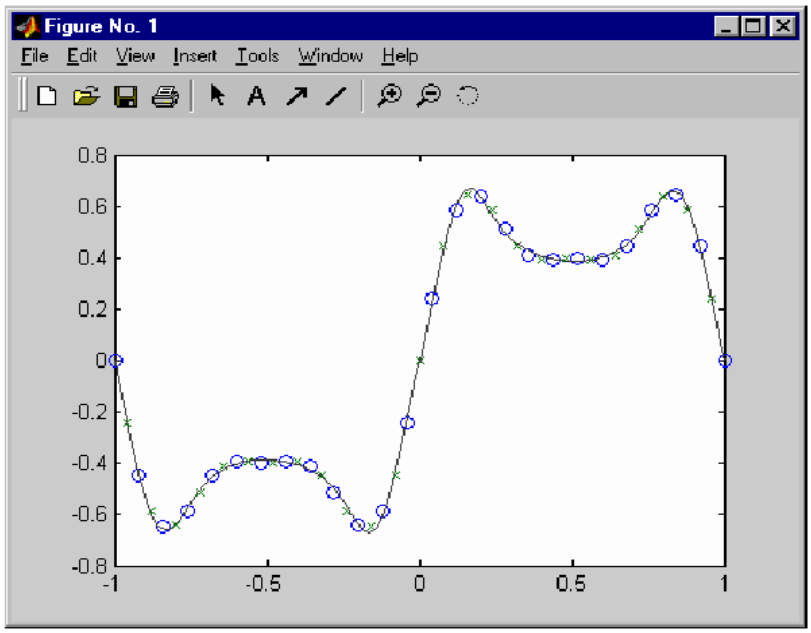
Membership functions after training:

```
>> [x,mf]=plotmf(fismat1,'input',1);
>> plot(x,mf)
```



Finally we can compare fuzzy system output to the training and checking data:

```
>> anfis_y=evalfis(x(:,1),fismat1);
>>
plot(trndata(:,1),trndata(:,2),'o',chkdata(:,1),chkdata(:,2),'x'
,x,anfis_y,'-')
```



In addition to command lines, it is also possible to open the system in ANFIS-editor and modify it there. The editor is started by command

```
>> anfisedit
```

2) Generate 51 input-output pairs between x ∈ [-10, 10], and choose training and checking datas:

```
>> numPts=51;
>> x=linspace(-10,10,numPts)';
```

```
>> y=-2*x-x.^2;
>> data=[x y];
>> trndata=data(1:2:numPts,:);
>> chkdata=data(2:2:numPts,:);
```

Set the number and type of membership functions:

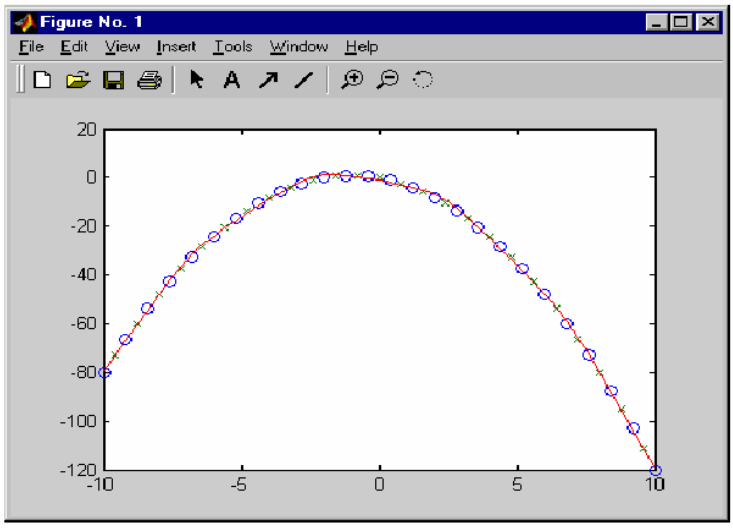
```
>> numMFs=5;
>> mfType='gbellmf';
```

Generate the FIS-matrix and execute the ANFIS-training by 40 rounds:

```
>> fismat=(genfis1(trndata,numMFs,mfType))
>> numEpochs=40;
>>
[fismat1,trnErr,ss,fismat2,chkErr]=anfis(trndata,fismat,numEpoch
s,NaN,chkdata);
```

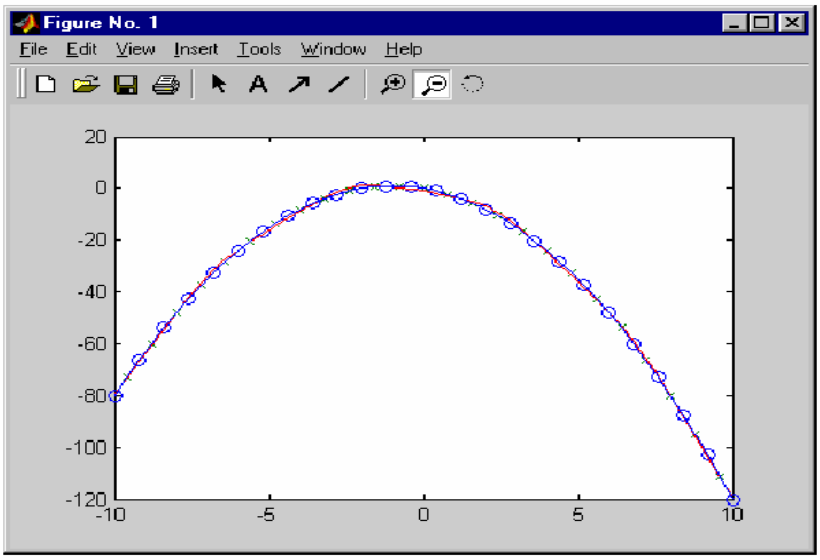
Compare training and checking data to the fuzzy approximation:

```
>> anfis_y=evalfis(x(:,1),fismat1);
>>
plot(trndata(:,1),trndata(:,2),'o',chkdata(:,1),chkdata(:,2),'x'
,x,anfis_y,'-')
```

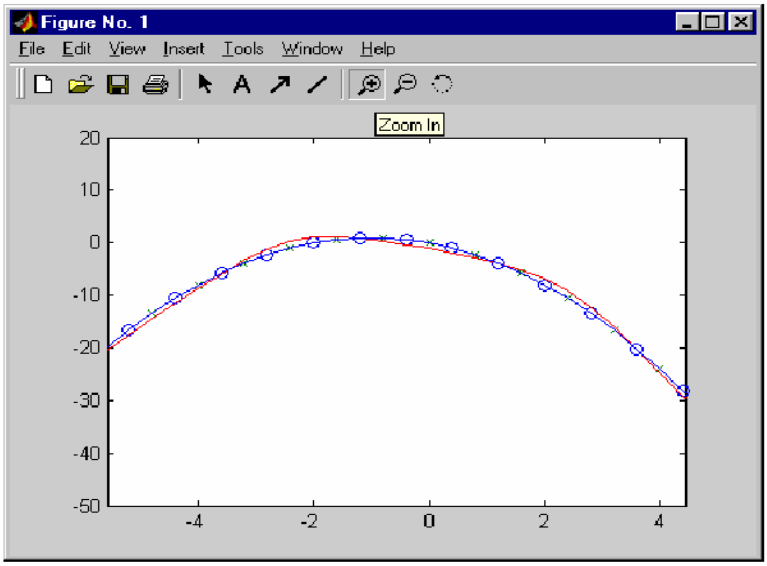


Draw also original function to the same picture so it is possible to compare function and fuzzy approximation:

```
>> hold on;
>> plot(x,y)
```

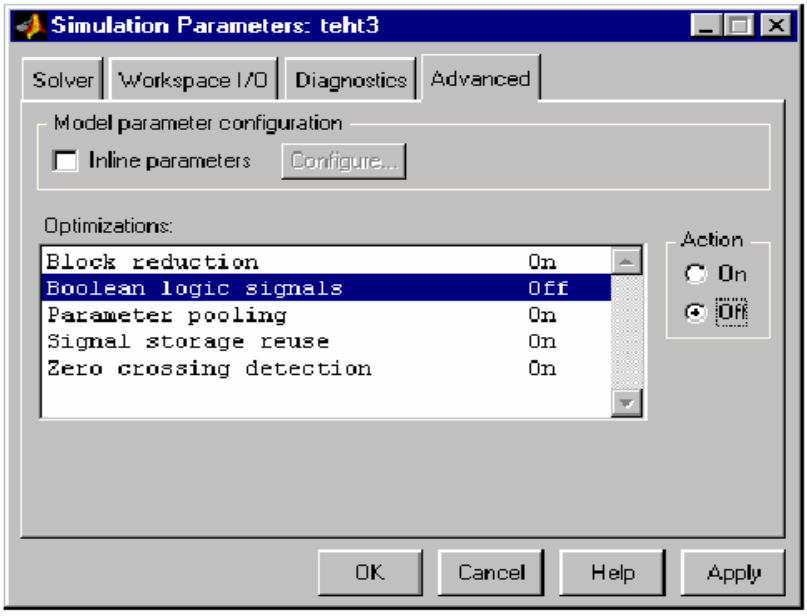


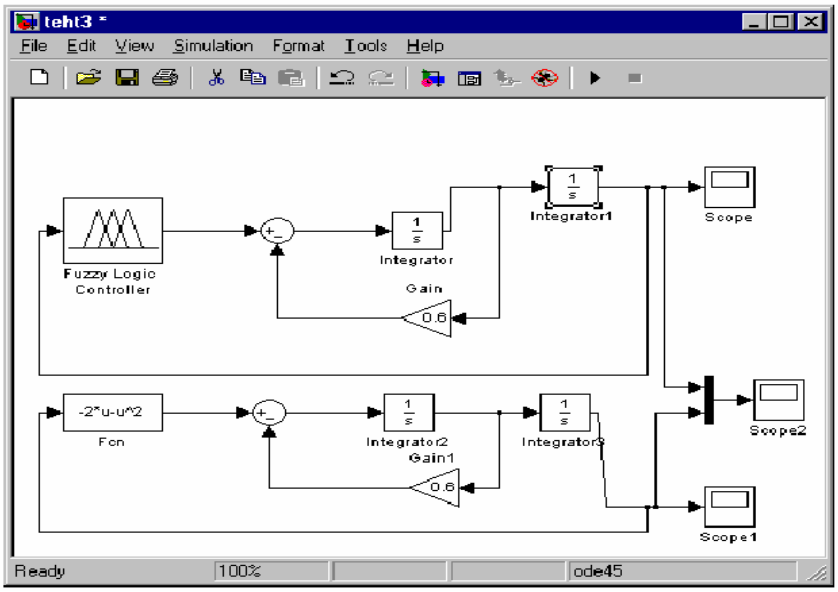
It is easier to see the differences in smaller scale picture:



3) Build SIMULINK-models of the system $\ddot{x} = -0.6x + f(x)$, where $f(x) = -2x - x^2$

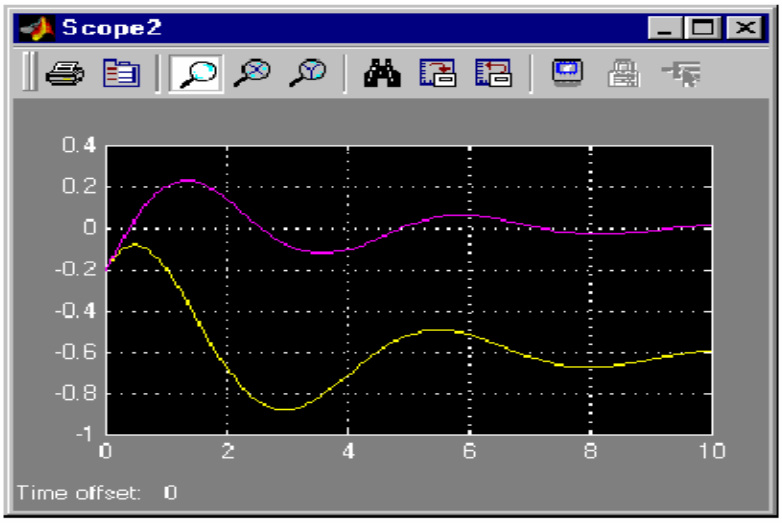
Note that you have to choose Boolean logic off in SIMULINK -> Simulation -> Simulation Parameters. Otherwise the fuzzy block doesn't work.





Remember to set the values $x(0) = -0.2$ and $\dot{x}(0) = 0.5$. Set the fuzzy system `fismat1` defined in problem 2 in Fuzzy Logic Controller. After that, it is possible to compare the results in Scope2 (or take the results in Matlab workspace).

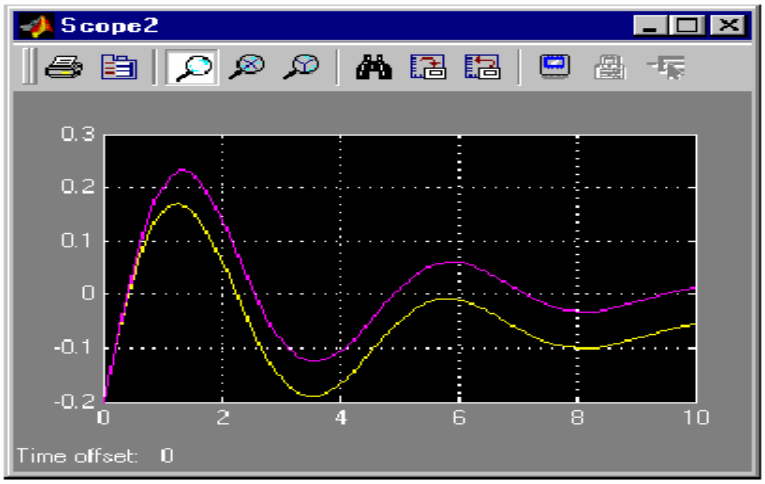
Result with 5 membership functions (yellow: fuzzy, pink: function):



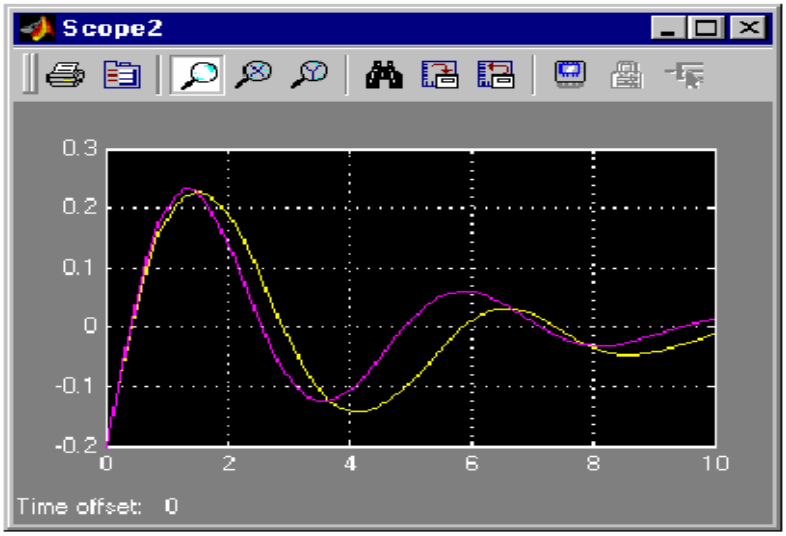
Increase the number of membership functions from 5 to 15. When changing the number, you have to do also new ANFIS-training:

```
>> numMFs=15;
>> mftype='gbellmf';
>> fismat=(genfis1(trndata,numMFs,mfType))

>> numEpochs=40;
>>
[fismat1,trnErr,ss,fismat2,chkErr]=anfis(trndata,fismat,numEpoch
s,NaN,chkdata);
```

After that the difference between systems with exact function and fuzzy approximation is smaller:



When we increase the number of membership functions from 15 to 60 the result might be better…



…but there is a danger that if too many membership functions is used, the system became overfitted! It means that the approximation is fitted too exactly to the training data and because of that, it doesn't give as good results by any other data anymore. To avoid overfitting, there should always be different datapoints for training, and for checking and validation.