NEURAL NETWORKS:

Basics using *MATLAB*

Neural Network Toolbox

By

Heikki N. Koivo

©

2008

Neural networks consist of a large class of different architectures. In many cases, the issue is approximating a static nonlinear, mapping $f(\mathbf{x})$ with a neural network $f_{NN}(\mathbf{x})$, where $\mathbf{x} \in \mathbb{R}^K$.

The most useful neural networks in function approximation are Multilayer Layer Perceptron (MLP) and Radial Basis Function (RBF) networks. Here we concentrate on MLP networks.

A MLP consists of an input layer, several hidden layers, and an output layer. Node *i*, also called a neuron, in a MLP network is shown in Fig. 1. It includes a summer and a nonlinear activation function *g*.

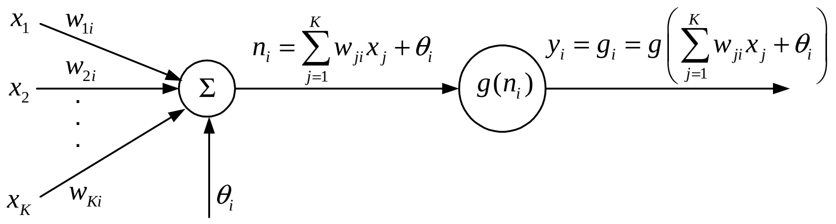

x_1
w_{1i}
x_2
w_{2i}
x_K
w_{Ki}
Σ
θ_i
$n_i = \sum_{j=1}^{K} w_{ji}x_j + \theta_i$
$g(n_i)$
$y_i = g_i = g\left(\sum_{j=1}^{K} w_{ji}x_j + \theta_i\right)$

Fig. 1. Single node in a MLP network.

The inputs x_k, $k = 1,\ldots,K$ to the neuron are multiplied by weights w_{ki} and summed up together with the constant bias term θ_i. The resulting n_i is the input to the activation function *g*. The activation function was originally chosen to be a relay function, but for mathematical convenience a hyberbolic tangent (*tanh*) or a sigmoid function are most commonly used. Hyberbolic tangent is defined as

$$\tanh(x) = \frac{1-e^{-x}}{1+e^{-x}} \tag{1}$$

The output of node *i* becomes

$$y_i = g_i = g\left(\sum_{j=1}^{K} w_{ji}x_j + \theta_i\right) \tag{2}$$

Connecting several nodes in parallel and series, a MLP network is formed. A typical network is shown in Fig. 2.

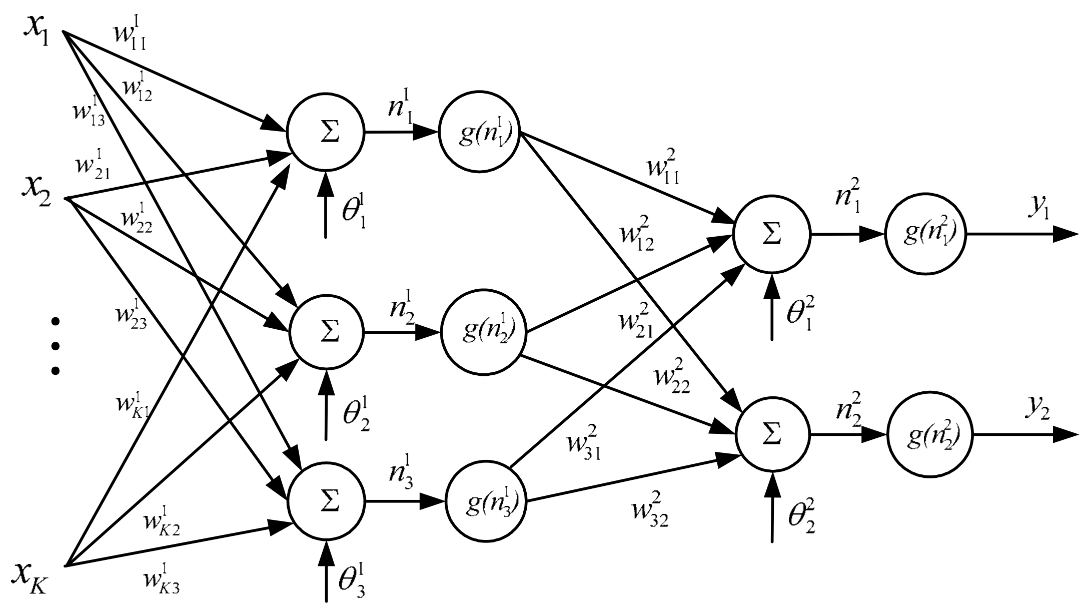


Fig. 2. A multilayer perceptron network with one hidden layer. Here the same activation function g is used in both layers. The superscript of n, θ, or w refers to the layer, first or second.

The output $y_i,\ i = 1,2,$ of the MLP network becomes

$$y_i = g\left(\sum_{j=1}^{3} w^2_{ji} g(n^1_j) + \theta^2_j\right) = g\left(\sum_{j=1}^{3} w^2_{ji} g\left(\sum_{k=1}^{K} w^1_{kj} x_k + \theta^1_j\right) + \theta^2_j\right) \tag{3}$$

From (3) we can conclude that a MLP network is a nonlinear parameterized map from input space $\mathbf{x} \in \mathbf{R}^K$ to output space $\mathbf{y} \in \mathbf{R}^m$ (here m = 3). The parameters are the weights w^k_{ji} and the biases θ^k_j. Activation functions g are usually assumed to be the same in each layer and known in advance. In the figure the same activation function g is used in all layers.

Given input-output data $(x_i, y_i),\ i = 1,\ldots,N,$ finding the best MLP network is formulated as a data fitting problem. The parameters to be determined are $\left(w^k_{ji}, \theta^k_j\right)$.

The procedure goes as follows. First the designer has to fix the structure of the MLP network architecture: the number of hidden layers and neurons (nodes) in each layer. The activation functions for each layer are also chosen at this stage, that is, they are assumed to be known. The unknown parameters to be estimated are the weights and biases, $\left(w^k_{ji}, \theta^k_j\right)$.

Many algorithms exist for determining the network parameters. In neural network literature the algorithms are called *learning* or *teaching* algorithms, in system identification they belong to *parameter estimation* algorithms. The most well-known are back-propagation and Levenberg-Marquardt algorithms. Back-propagation is a gradient based algorithm, which has many variants. Levenberg-Marquardt is usually more efficient, but needs more computer memory. Here we will concentrate only on using the algorithms.

Summarizing the procedure of teaching algorithms for multilayer perceptron networks:

a. The structure of the network is first defined. In the network, activation functions are chosen and the network parameters, weights and biases, are initialized.
b. The parameters associated with the training algorithm like error goal, maximum number of epochs (iterations), etc, are defined.
c. The training algorithm is called.
d. After the neural network has been determined, the result is first tested by simulating the output of the neural network with the measured input data. This is compared with the measured outputs. Final validation must be carried out with independent data.

The MATLAB commands used in the procedure are *newff, train* and *sim.*

The MATLAB command *newff* generates a MLPN neural network, which is called *net.*

$$\boxed{net = newff(\underbrace{PR}_{\text{min,max values}},[\underbrace{S1\ S2...SNl}_{\text{size of the } i\text{th layer}}],\{\underbrace{TF1\ TF2...TFNl}_{\text{activation function of } i\text{th layer}}\},\underbrace{BTF}_{\text{training algorithm}})} \tag{4}$$

The inputs in (4) are

R	= Number of elements in input vector
xR	= *R*x2 matrix of min and max values for *R* input elements,
Si	= Number of neurons (size) in the *i*th layer, *i = 1,…,Nl*
Nl	= Number of layers
TFi	= Activation (or transfer function) of the *i*th layer, default = '*tansig*',
BTF	= Network training function, default = '*trainlm*'

In Fig. 2 *R = K, S1=3, S2 = 2, Nl* = 2 and *TFi =g.*

The default algorithm of command *newff* is Levenberg-Marquardt, *trainlm.* Default parameter values for the algorithms are assumed and are hidden from the user. They need not be adjusted in the first trials. Initial values of the parameters are automatically generated by the command. Observe that their generation is random and therefore the answer might be different if the algorithm is repeated.

After initializing the network, the network training is originated using *train* command. The resulting MLP network is called *net*1.

$$net1 = train(\underbrace{net}_{\substack{initial\\ MLP}}, \underbrace{x}_{\substack{measured\\ input\ vector}}, \underbrace{y}_{\substack{measured\\ output\ vector}}) \tag{5}$$

The arguments are: *net*, the initial MLP network generated by *newff*, *x,* measured input vector of dimension *K* and *y* measured output vector of dimension *m*.

To test how well the resulting MLP *net*1 approximates the data, *sim* command is applied. The measured output is *y*. The output of the MLP network is simulated with *sim* command and called *ytest.*

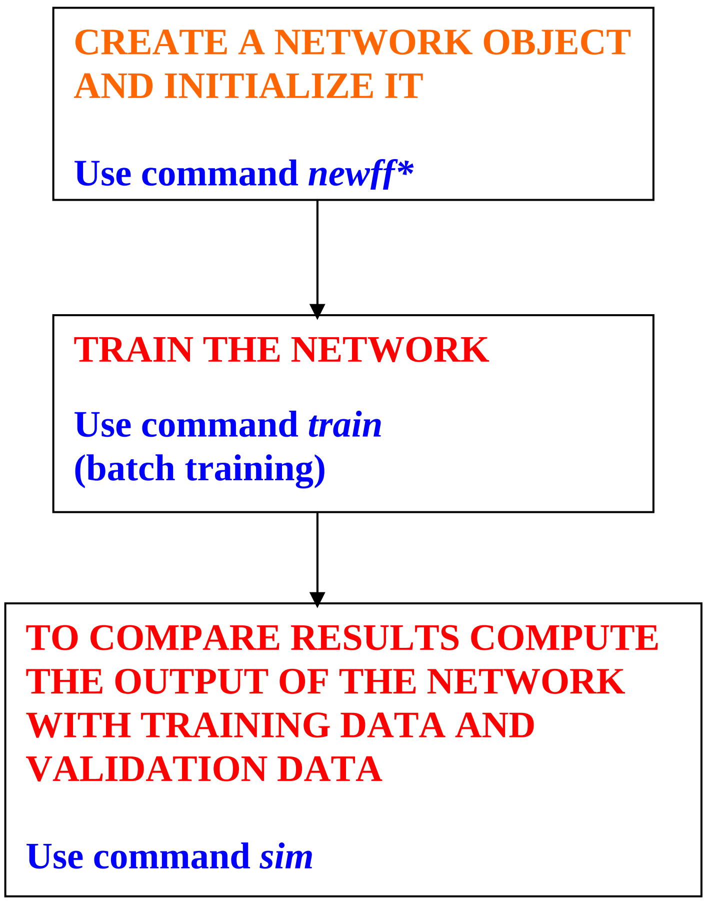
$$ytest = sim(\underbrace{net1}_{\substack{final\\ MLP}}, \underbrace{x}_{\substack{\hat{i}nput\\ vector}}) \tag{6}$$

The measured output *y* can now be compared with the output of the MLP network *ytest* to see how good the result is by computing the error difference *e = y – ytest* at each measured point. The final validation must be done with independent data.

In the following a number of examples are covered, where MATLAB Neural Network Toolbox is used to learn the parameters in the network, when input-output data is available.

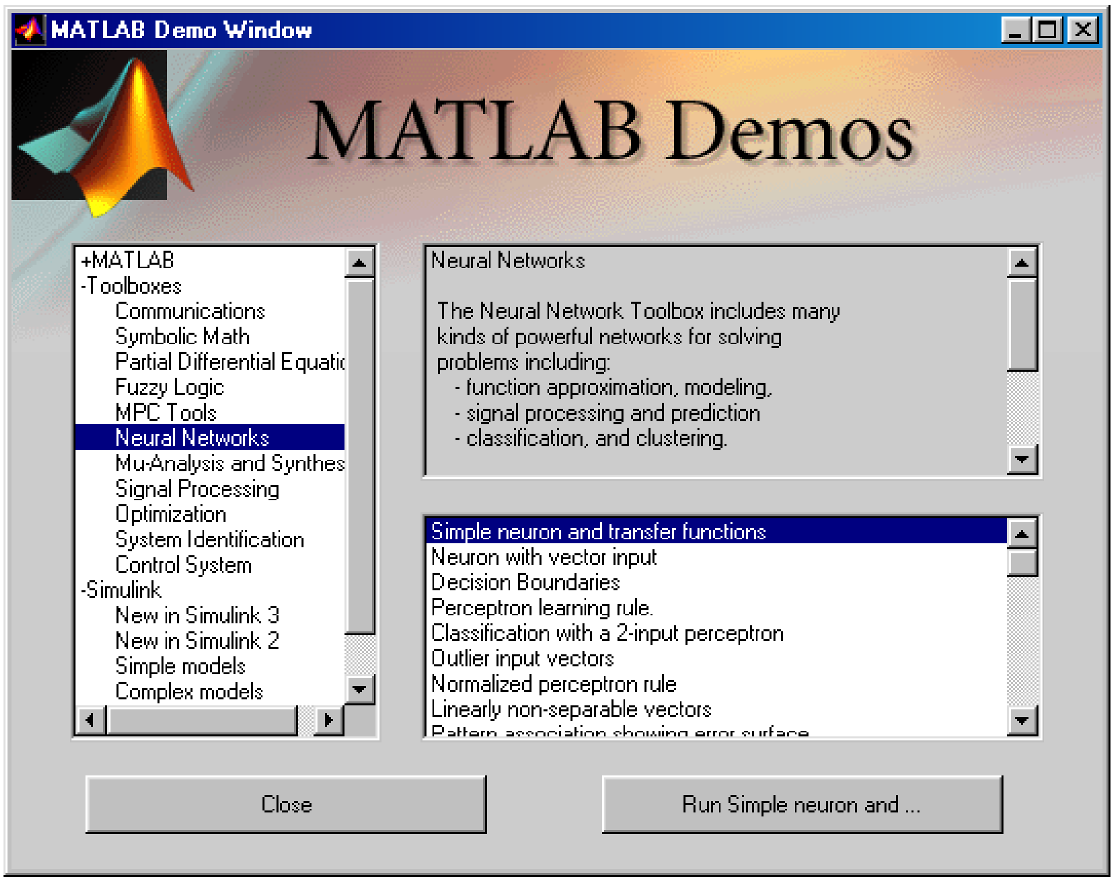
NEURAL NETWORKS - EXERCISES WITH MATLAB AND SIMULINK

BASIC FLOW DIAGRAM


CREATE A NETWORK OBJECT AND INITIALIZE IT
Use command *newff**
TRAIN THE NETWORK
Use command *train* (batch training)
TO COMPARE RESULTS COMPUTE THE OUTPUT OF THE NETWORK WITH TRAINING DATA AND VALIDATION DATA
Use command *sim*

*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.



+MATLAB
-Toolboxes
Communications
Symbolic Math
Partial Differential Equatic
Fuzzy Logic
MPC Tools
Neural Networks
Mu-Analysis and Synthes
Signal Processing
Optimization
System Identification
Control System
-Simulink
New in Simulink 3
New in Simulink 2
Simple models
Complex models

Neural Networks

The Neural Network Toolbox includes many
kinds of powerful networks for solving
problems including:
- function approximation, modeling,
- signal processing and prediction
- classification, and clustering.

Simple neuron and transfer functions
Neuron with vector input
Decision Boundaries
Perceptron learning rule.
Classification with a 2-input perceptron
Outlier input vectors
Normalized perceptron rule
Linearly non-separable vectors
Pattern association showing error surface

EXAMPLE 1: Consider *humps* function in MATLAB. It is given by

y = 1 ./ ((x-.3).^2 + .01) + 1 ./ ((x-.9).^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].

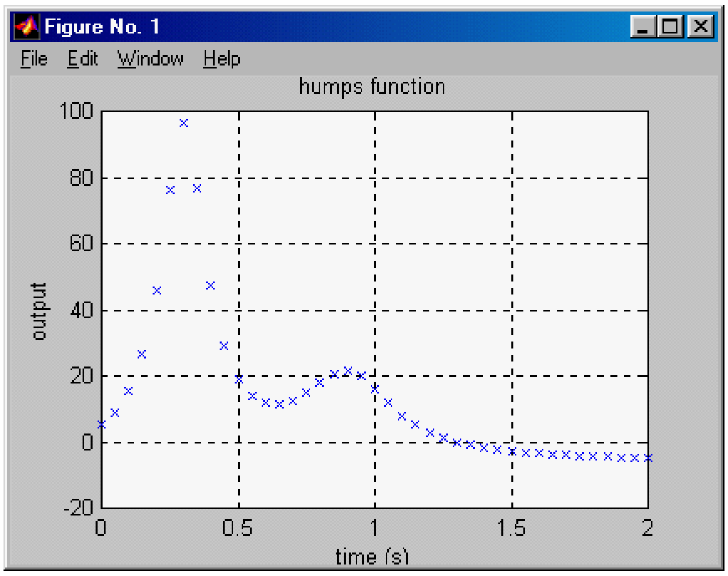
a) Fit a multilayer perceptron network on the data. Try different network sizes and different teaching algorithms.
b) Repeat the exercise with radial basis function networks.

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:

e. Define the structure of the network. Choose activation functions and initialize the neural network parameters, weights and biases, either providing them yourself or using initializing routines.
   MATLAB command for MLPN initialization is *newff.*
f. Define the parameters associated with the training algorithm like error goal, maximum number of epochs (iterations), etc.
g. Call the ttraining algorithm. In MATLAB the command is *train*.

%   DESIGN THE NETWORK
%   ==================

%First try a simple one – feedforward (multilayer perceptron) network

net=newff([0 2], [5,1], {'tansig','purelin'},'traingd');

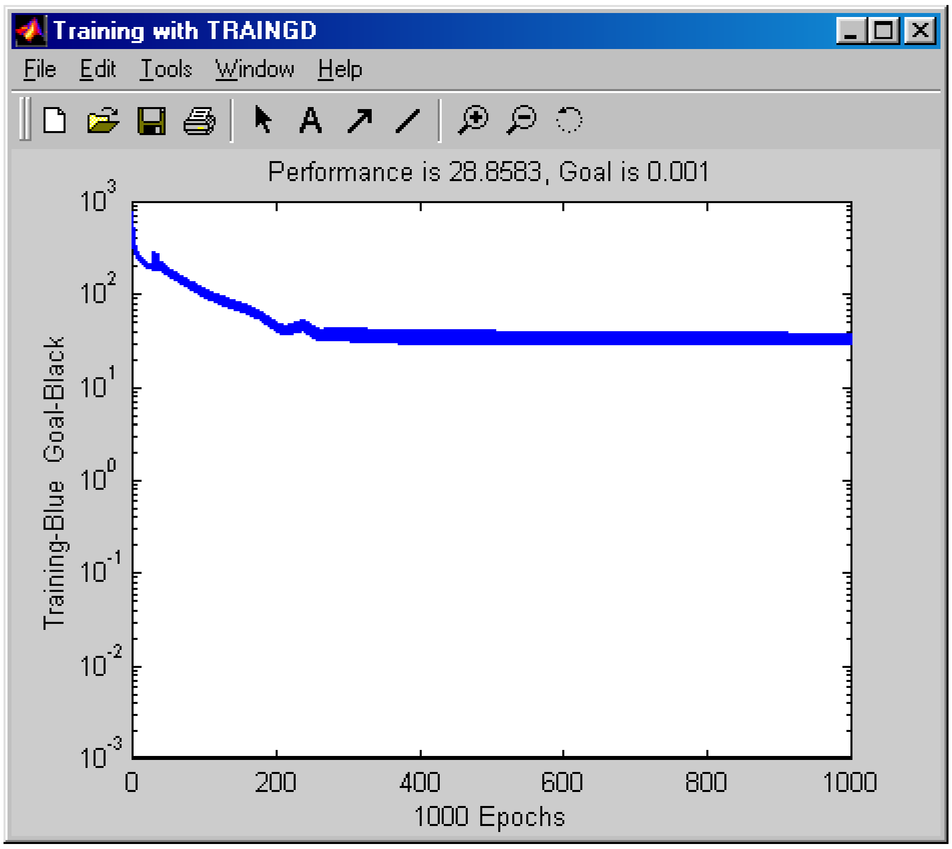
% Here newff defines feedforward network architecture.

```
%  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.

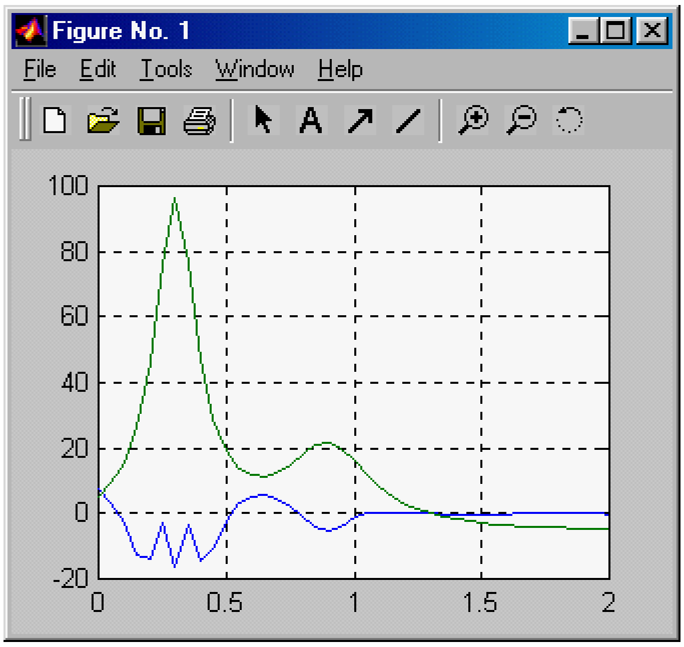


It is also clear that even if more iterations will be performed, no improvement is in store. Let us still check how the neural network approximation looks like.

% 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.

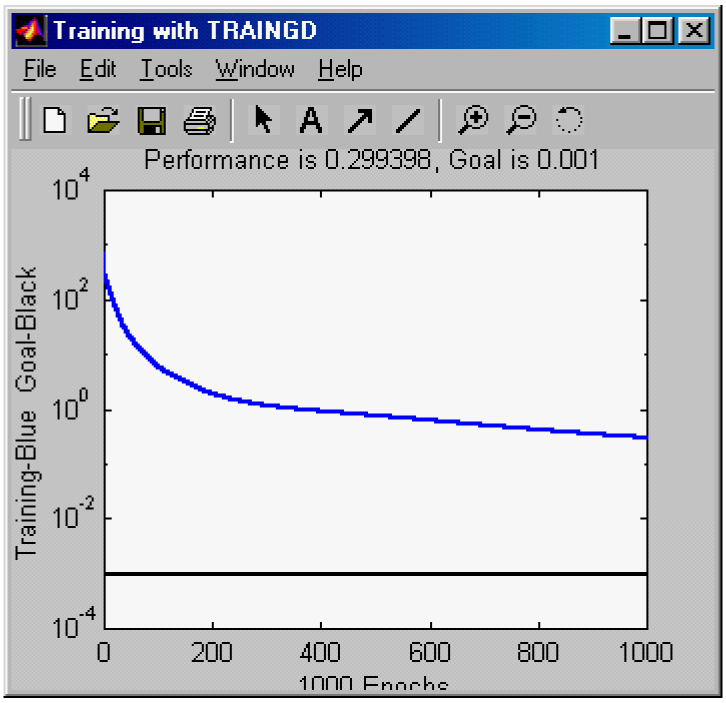
```
net=newff([0 2], [20,1], {'tansig','purelin'},'traingd');
```

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.

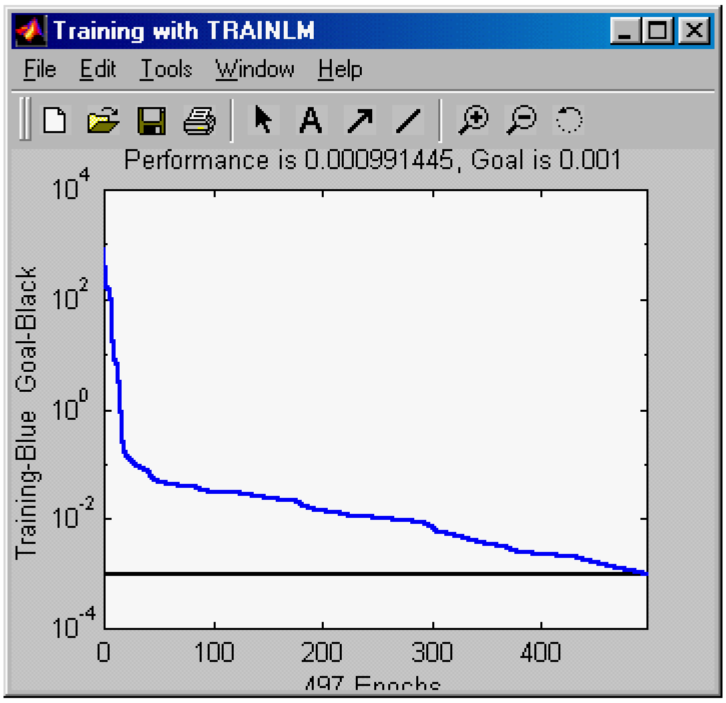
Try **Levenberg-Marquardt** – *trainlm.* Use also smaller size of network – 10 nodes in the first hidden layer.

```
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 = trainlm(net, P, T);
```

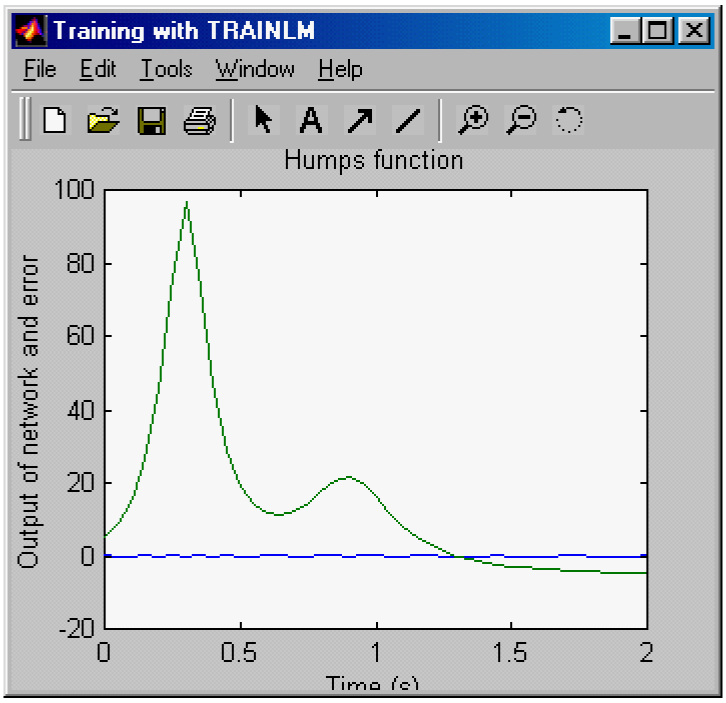
```
TRAINLM, Epoch 0/1000, MSE 830.784/0.001, Gradient 1978.34/1e-010
….
TRAINLM, Epoch 497/1000, MSE 0.000991445/0.001, Gradient 1.44764/1e-010
TRAINLM, Performance goal met.
```

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.

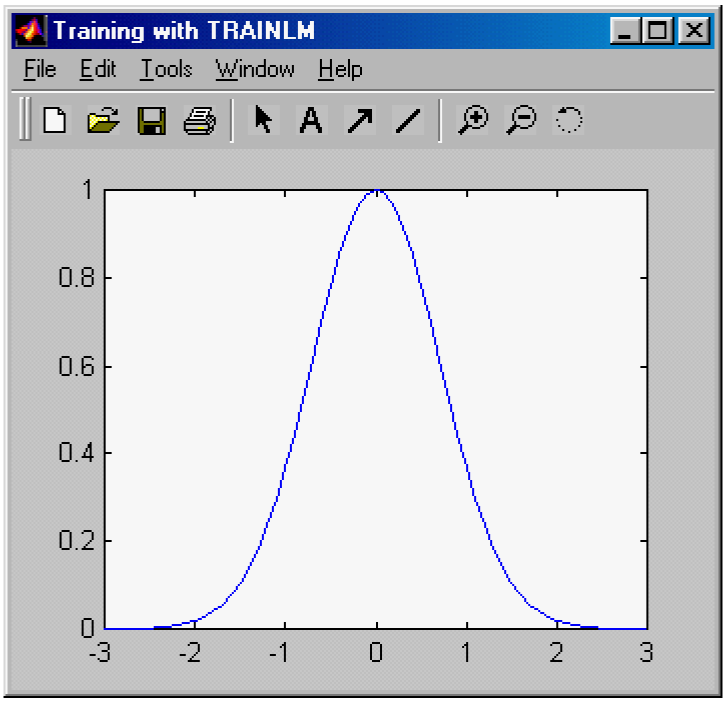
b) RADIAL BASIS FUNCTION NETWORKS

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;
```

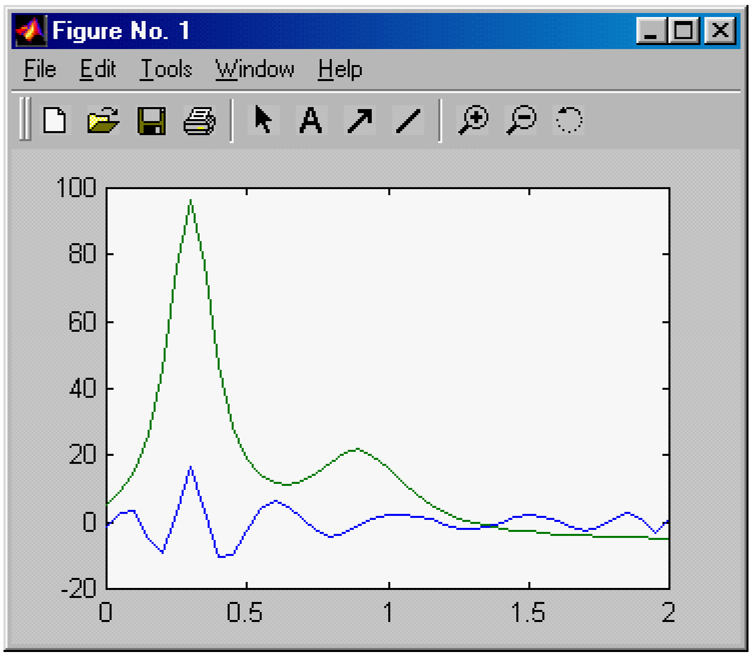
The simplest form of *newrb* command is

```
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.

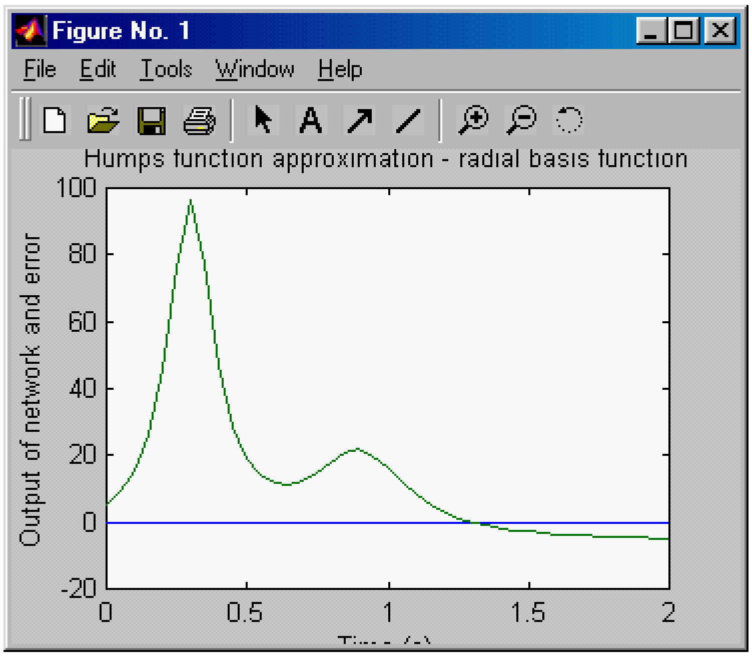
In our example choose *goal* = 0.02 and *spread* = 0.1.

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.*

EXAMPLE 2. Consider a surface described by $z = \cos(x) \sin(y)$ defined on a square $-2 \le x \le 2, -2 \le y \le 2$.

a) Plot the surface *z* as a function of *x* and *y*. This is a demo function in MATLAB, so you can also find it there.
b) Design a neural network, which will fit the data. You should study different alternatives and test the final result by studying the fitting error.

SOLUTION

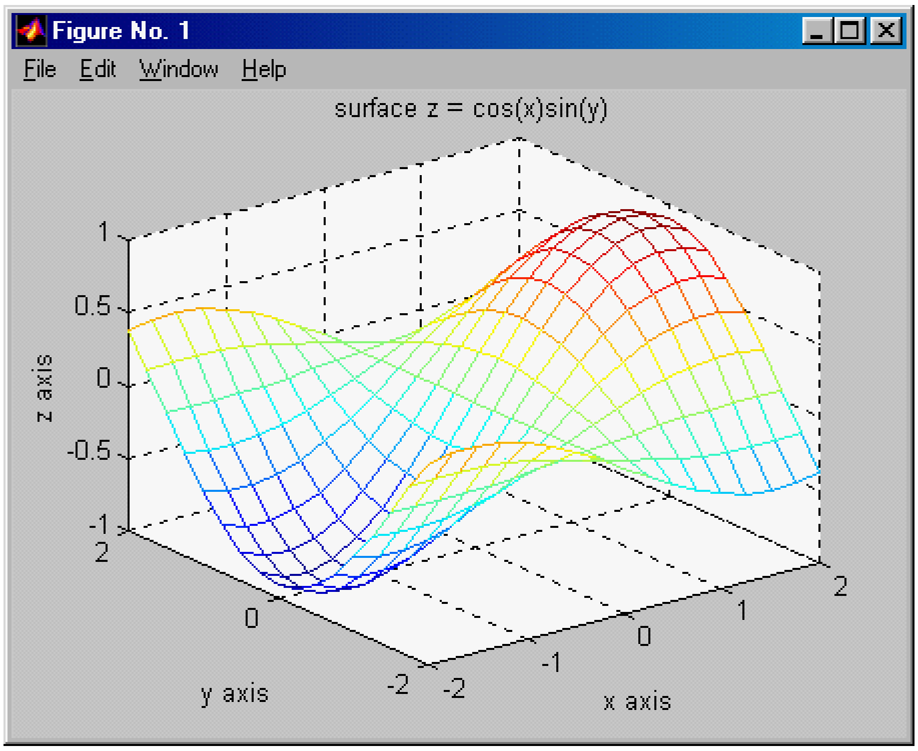
Generate data

```
x = -2:0.25:2; y = -2:0.25:2;
z = cos(x)'*sin(y);
```

Draw the surface (here grid size of 0.1 has been used)

```
mesh(x,y,z)
xlabel('x axis'); ylabel('y axis'); zlabel('z axis');
 title('surface z = cos(x)sin(y)');
gi=input('Strike any key ...');
```

pause



Store data in input matrix P and output vector T
P = [x;y]; T = z;

Use a fairly small number of neurons in the first layer, say 25, 17 in the output.
Initialize the network

net=newff([-2 2; -2 2], [25 17], {'tansig' 'purelin'},'trainlm');

Apply Levenberg-Marquardt algorithm

%Define parameters

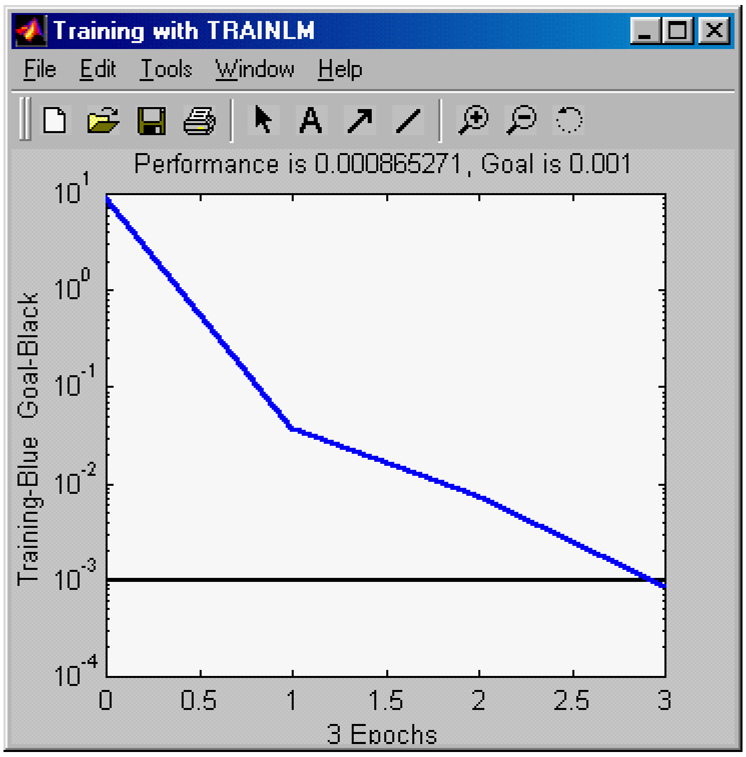
net.trainParam.show = 50;
net.trainParam.lr = 0.05;
net.trainParam.epochs = 300;
net.trainParam.goal = 1e-3;

%Train network
net1 = train(net, P, T);

gi=input('Strike any key ...');

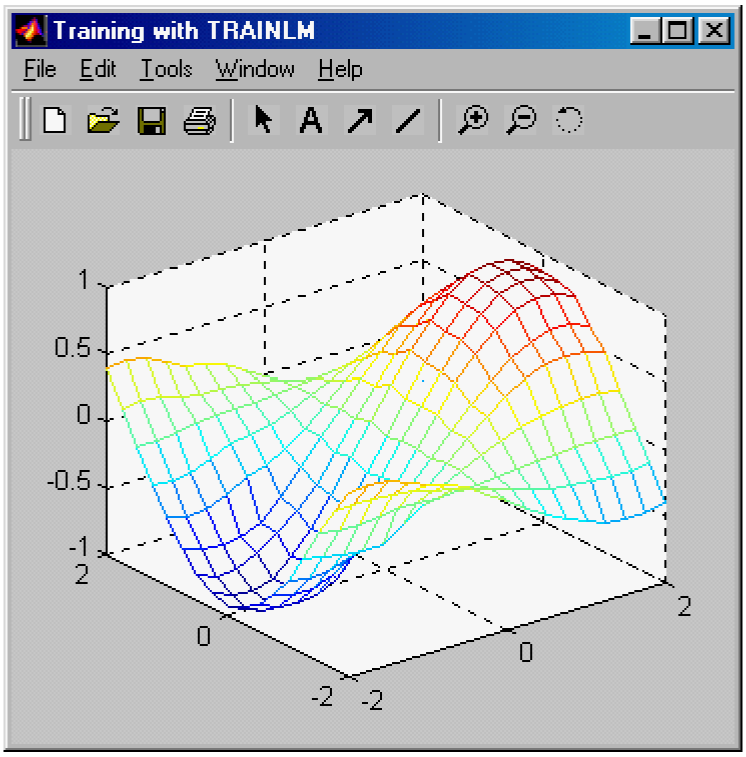
TRAINLM, Epoch 0/300, MSE 9.12393/0.001, Gradient 684.818/1e-010
TRAINLM, Epoch 3/300, MSE 0.000865271/0.001, Gradient 5.47551/1e-010
TRAINLM, Performance goal met.

Plot how the error develops



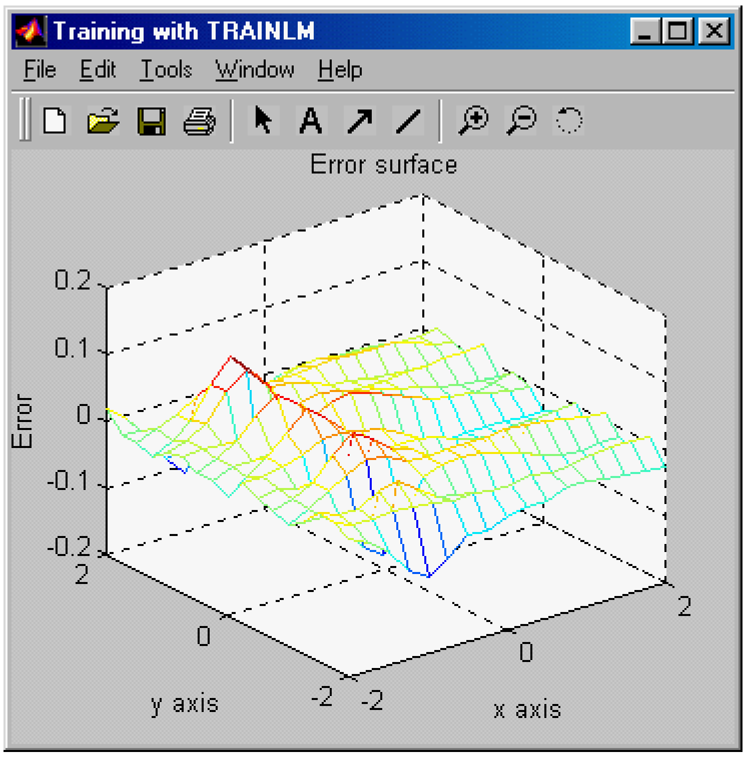
Simulate the response of the neural network and draw the corresponding surface

a= sim(net1,P);
mesh(x,y,a)



The result looks satisfactory, but a closer examination reveals that in certain areas the approximation is not so good. This can be seen better by drawing the error surface.

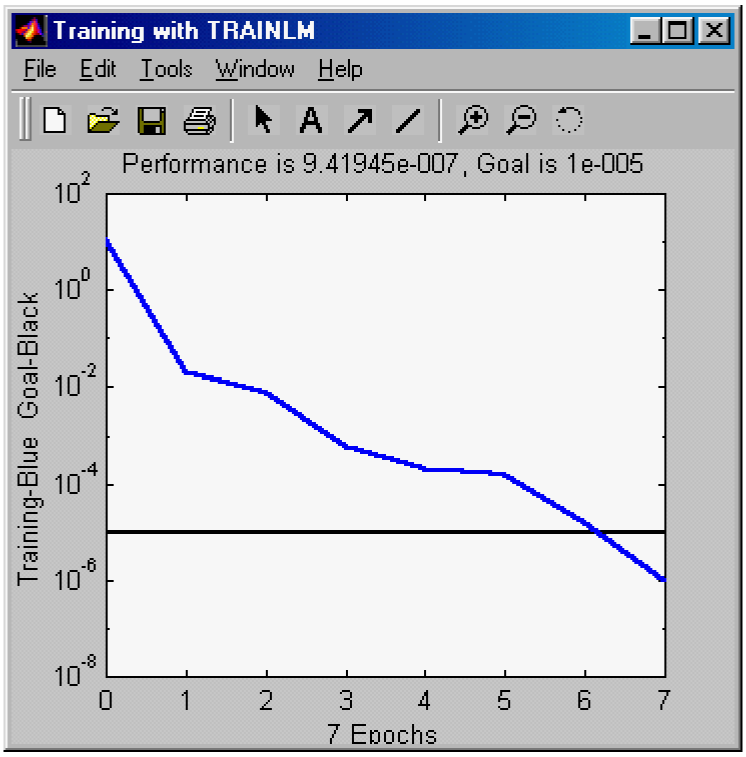
% Error surface
mesh(x,y,a-z)
xlabel('x axis'); ylabel('y axis'); zlabel('Error'); title('Error surface')



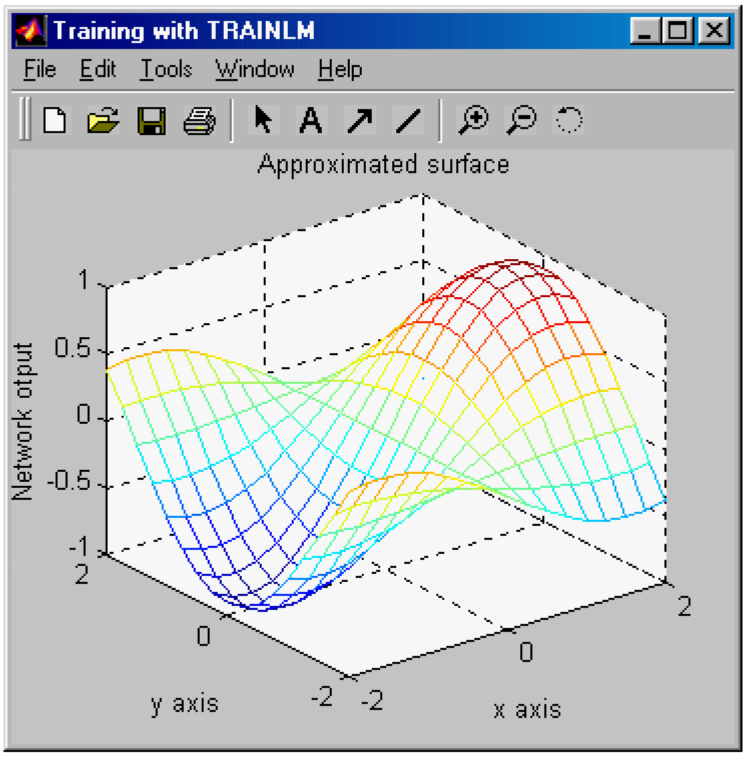
gi=input('Strike any key to continue......');

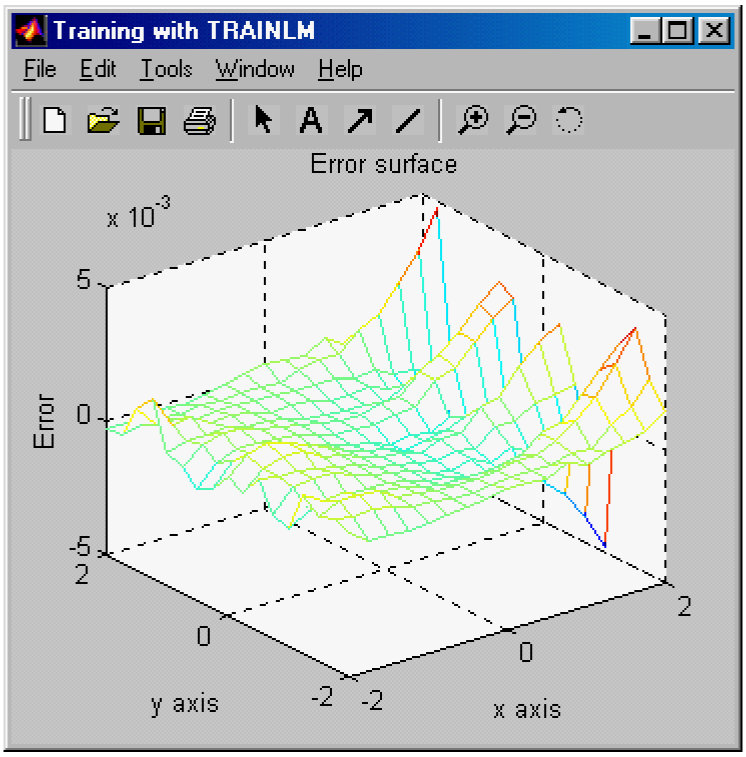
% Maximum fitting error
Maxfiterror = max(max(z-a))
Maxfiterror = 0.1116

Depending on the computing power of your computer the error tolerance can be made stricter, say 10^{-5}. The convergence now takes considerably more time and is shown below.



Producing the simulated results gives the following results





EXAMPLE 3: Consider Bessel functions $J_\alpha(t)$, which are solutions of the differential equation

$$t^2\ddot{y} + t\dot{y} + (t^2 - \alpha^2)y = 0 .$$

Use backpropagation network to approximate first order Bessel function J_1 , α=*1*, when t $\in$ [0,20].

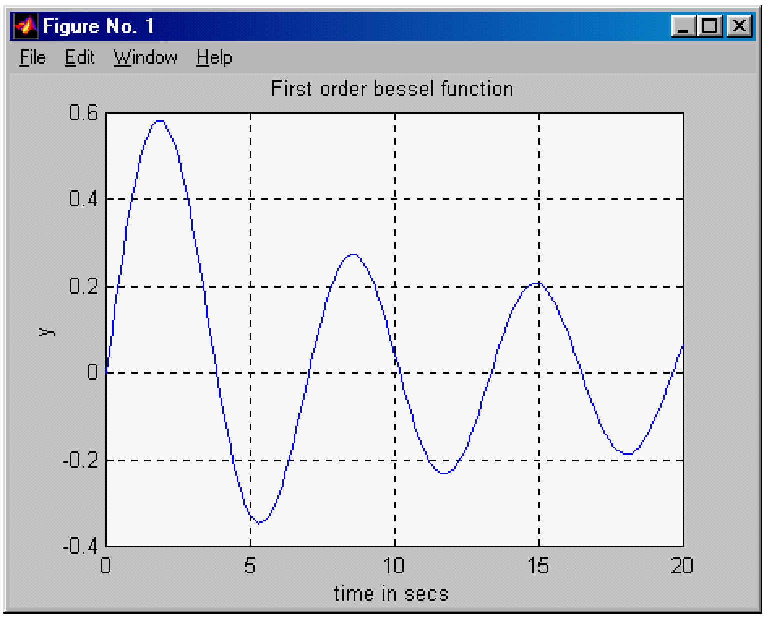
a) Plot J_1(t).
b) Try different structures to for fitting. Start with a two-layer network.
You might also need different learning algorithms.

SOLUTION:

1. First generate the data.

MATLAB has Bessel functions as MATLAB functions.

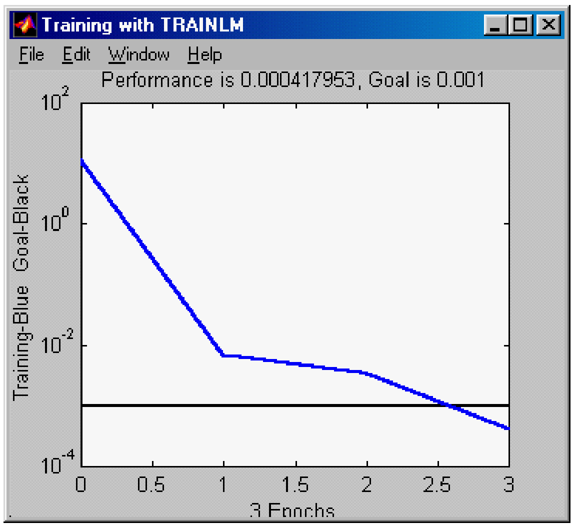
```
t=0:0.1:20; y=bessel(1,t);
plot(t,y)
grid
xlabel('time in secs');ylabel('y'); title('First order bessel function');
```



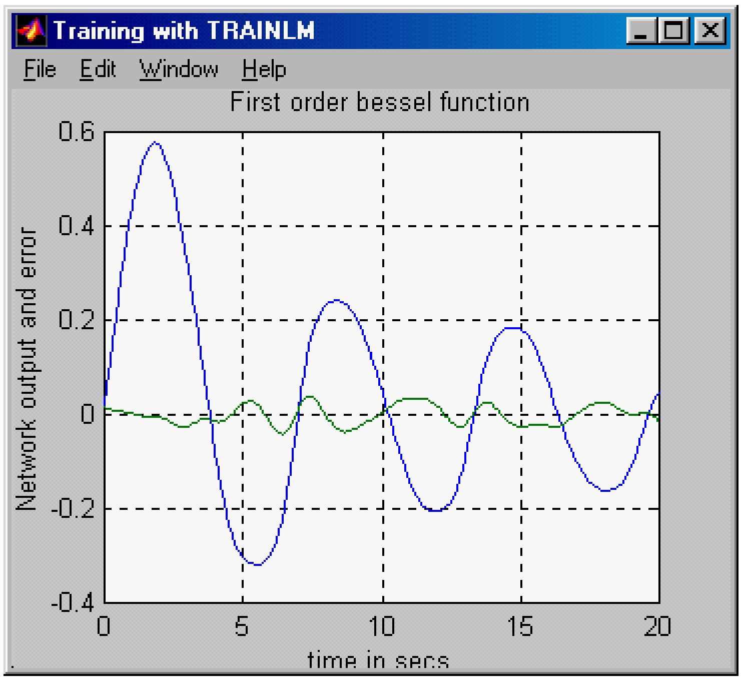
2. Next try to fit a backpropagation network on the data. Try Levenberg-Marquardt.

```
P=t; T=y;

%Define network. First try a simple one

net=newff([0 20], [10,1], {'tansig','purelin'},'trainlm');

%Define parameters

net.trainParam.show = 50;
net.trainParam.lr = 0.05;
net.trainParam.epochs = 300;
net.trainParam.goal = 1e-3;

%Train network
net1 = train(net, P, T);
```

```
TRAINLM, Epoch 0/300, MSE 11.2762/0.001, Gradient 1908.57/1e-010
TRAINLM, Epoch 3/300, MSE 0.000417953/0.001, Gradient 1.50709/1e-010
TRAINLM, Performance goal met.
```



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

%Plot result and compare
plot(P,a,P,a-T)
xlabel('time in secs');ylabel('Network output and error');
title('First order bessel function'); grid
```

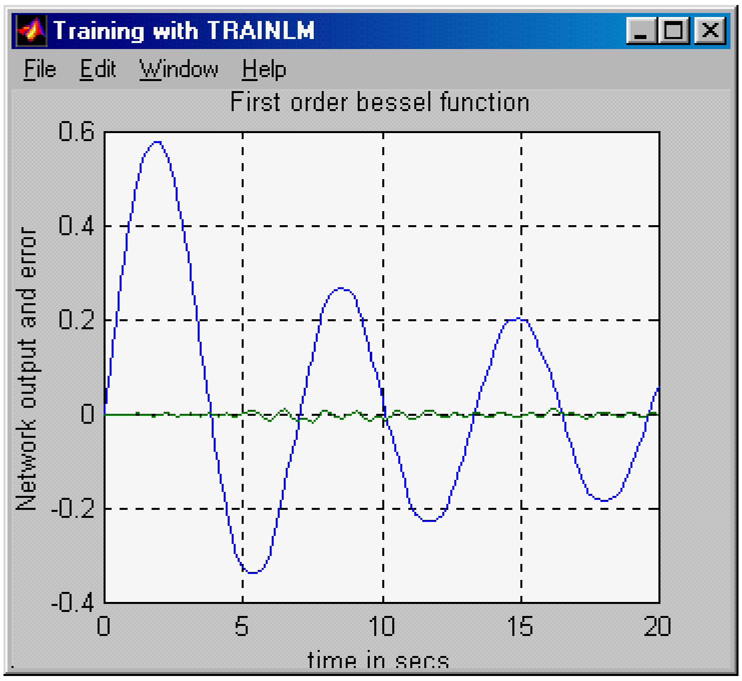


Since the error is fairly significant, let's reduce it by doubling the nodes in the first hidden layer to 20 and decreasing the error tolerance to 10^{-4}.

The training is over in 8 iterations

```
TRAINLM, Epoch 0/300, MSE 4.68232/0.0001, Gradient 1153.14/1e-010
TRAINLM, Epoch 8/300, MSE 2.85284e-005/0.0001, Gradient 0.782548/1e-010
TRAINLM, Performance goal met.
```

After plotting this results in the following figure.



The result is considerably better, although it would still require improvement. This is left as further exercise to the reader.

EXAMPLE 4: Study, if it is possible to find a neural network model, which produces the same behavior as Van der Pol equation.

$$\ddot{x} + (x^2 - 1)\dot{x} + x = 0$$

or in state-space form

$$\dot{x}_1 = x_2(1 - x_1^2) - x_1$$
$$\dot{x}_2 = x_1$$

Use different initial functions.

Apply vector notation

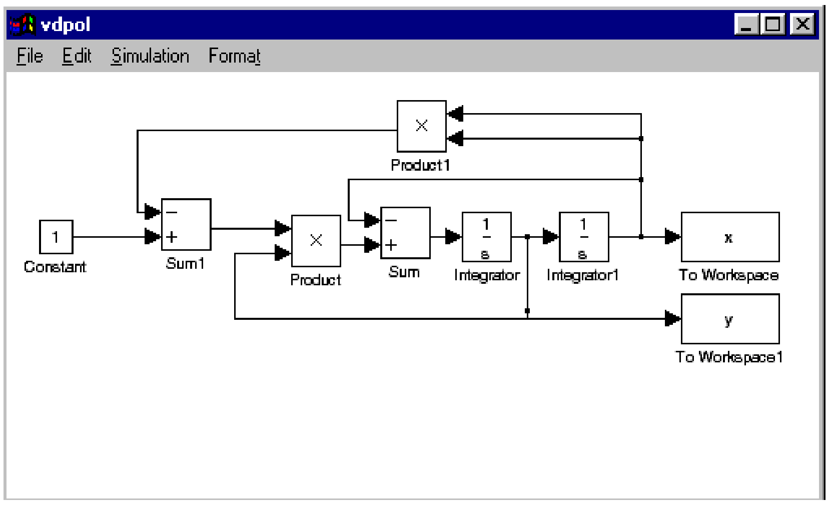
$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x})$$

where

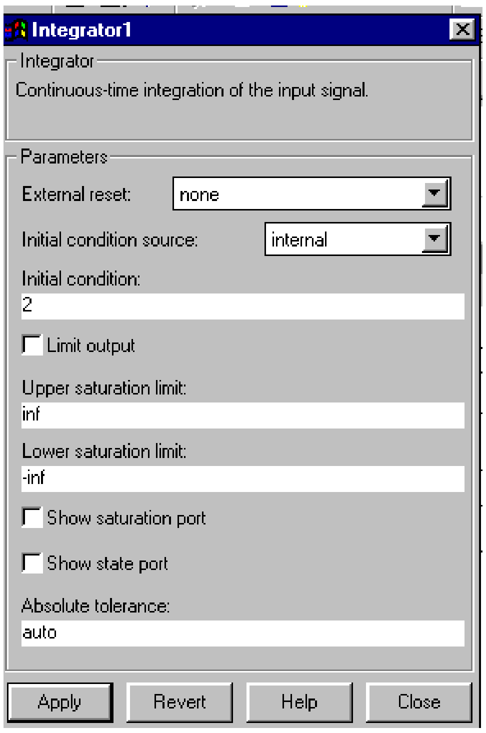
$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \text{ and } \mathbf{f}(\mathbf{x}) = \begin{bmatrix} f_1(x_1, x_2) \\ f_2(x_1, x_2) \end{bmatrix} = \begin{bmatrix} x_2(1 - x_1^2) - x_1 \\ x_1 \end{bmatrix}$$

SOLUTION:

First construct a Simulink model of Van der Pol system. It is shown below. Call it *vdpol.*



Recall that initial conditions can be defined by opening the integrators. For example the initial condition $x_1(0) = 2$ is given by opening the corresponding integrator.


Integrator1

Integrator

Continuous-time integration of the input signal.

Parameters

External reset: none

Initial condition source: internal

Initial condition:
2

Limit output

Upper saturation limit:
inf

Lower saturation limit:
-inf

Show saturation port

Show state port

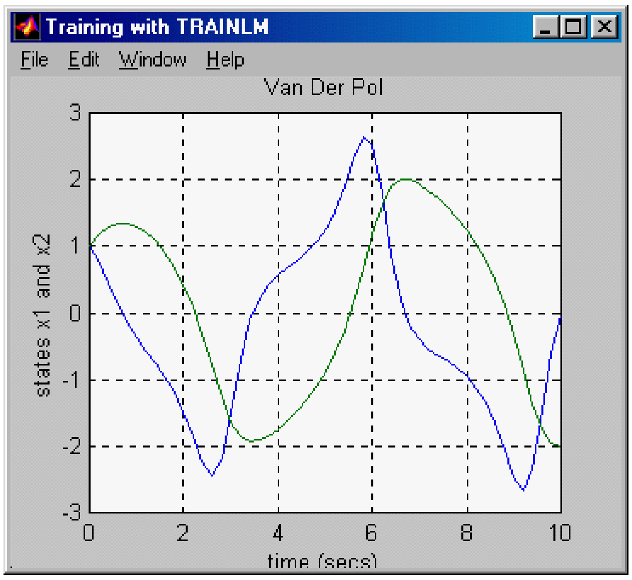
Absolute tolerance:
auto

Use initial condition $x_1(0) = 1$, $x_2(0) = 1$.

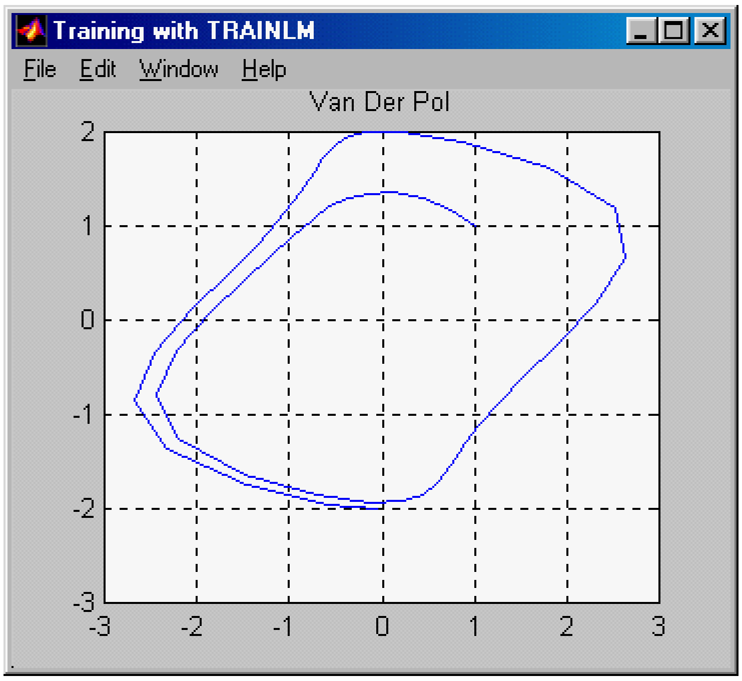
% Define the simulation parameters for Van der Pol equation
% The period of simulation: tfinal = 10 seconds;
tfinal = 10;

% Solve Van der Pol differential equation
[t,x]=sim('vdpol',tfinal);

% Plot the states as function of time
plot(t,x)
xlabel(‘time (secs)'); ylabel('states x1 and x2'); title('Van Der Pol'); grid
gi=input(Strike any key ...');



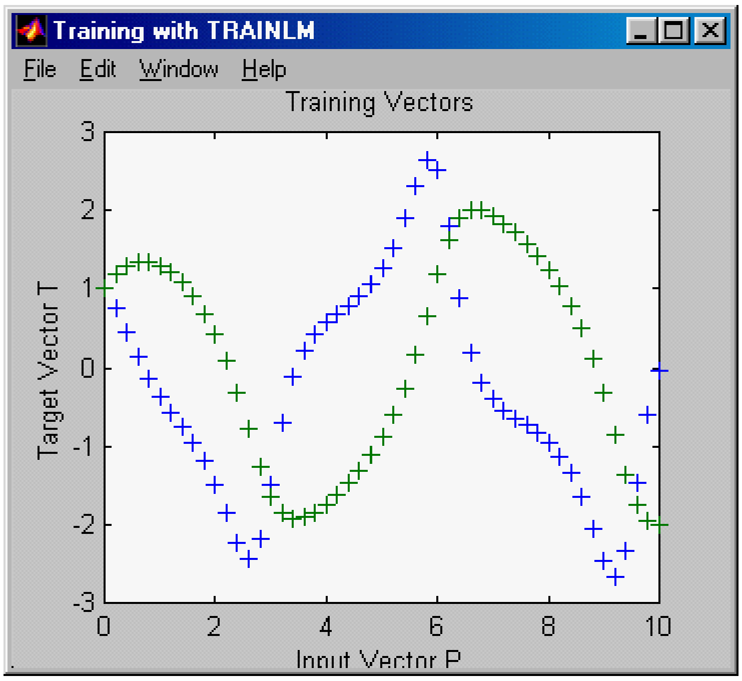
%Plot also the phase plane plot
plot(x(:,1),x(:,2)), title('Van Der Pol'),grid
gi=input(Strike any key ...');



% Now you have data for one trajectory, which you can use to teach a neural network
% Plot the data (solution). Observe that the output vector includes both

P = t';
T = x';

plot(P,T,'+');
title('Training Vectors');
xlabel('Input Vector P');
ylabel('Target Vector T');
gi=input('Strike any key ...');



% Define the learning algorithm parameters, radial basis function network chosen

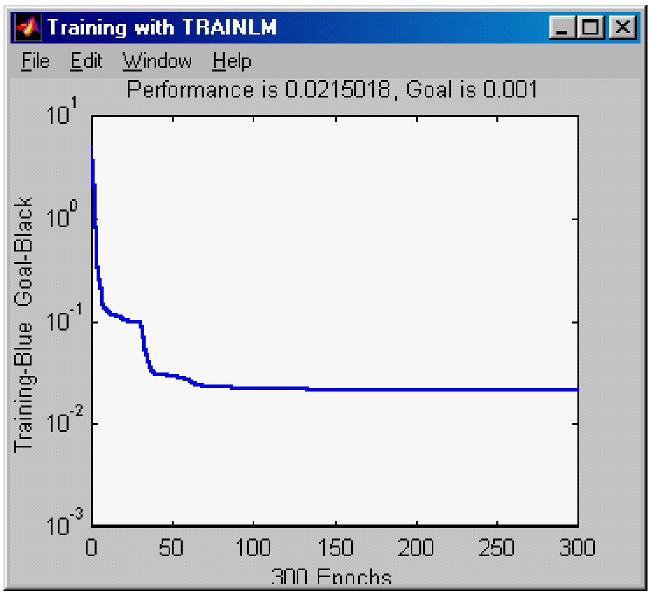
net=newff([0 20], [10 2], {'tansig','purelin'},'trainlm');

%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);gi=input(' Strike any key...');

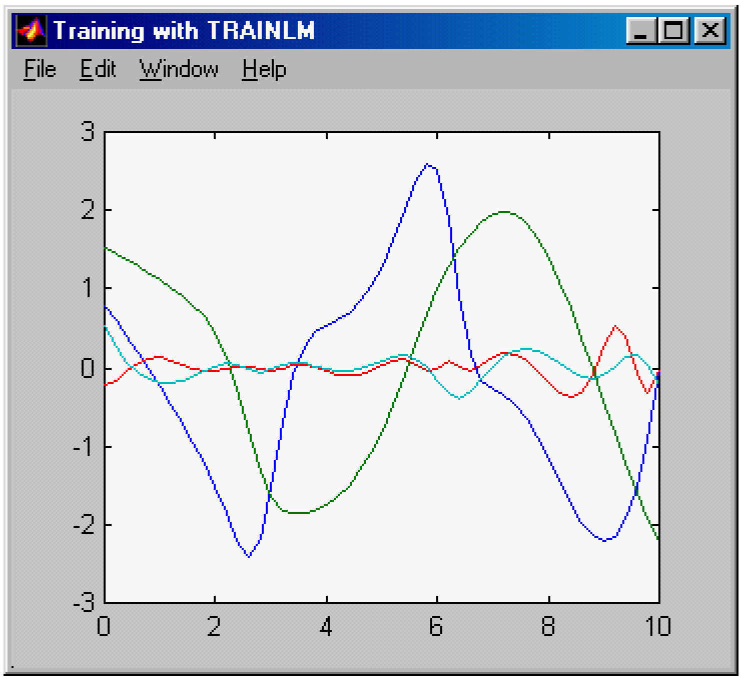
```
TRAINLM, Epoch 0/300, MSE 4.97149/0.001, Gradient 340.158/1e-010
TRAINLM, Epoch 50/300, MSE 0.0292219/0.001, Gradient 0.592274/1e-010
TRAINLM, Epoch 100/300, MSE 0.0220738/0.001, Gradient 0.777432/1e-010
TRAINLM, Epoch 150/300, MSE 0.0216339/0.001, Gradient 1.17908/1e-010
TRAINLM, Epoch 200/300, MSE 0.0215625/0.001, Gradient 0.644787/1e-010
TRAINLM, Epoch 250/300, MSE 0.0215245/0.001, Gradient 0.422469/1e-010
TRAINLM, Epoch 300/300, MSE 0.0215018/0.001, Gradient 0.315649/1e-010
TRAINLM, Maximum epoch reached, performance goal was not met.
```



The network structure is too simple, since convergence is not achieved. Let’s see how close the network solution is to the simulated solution.

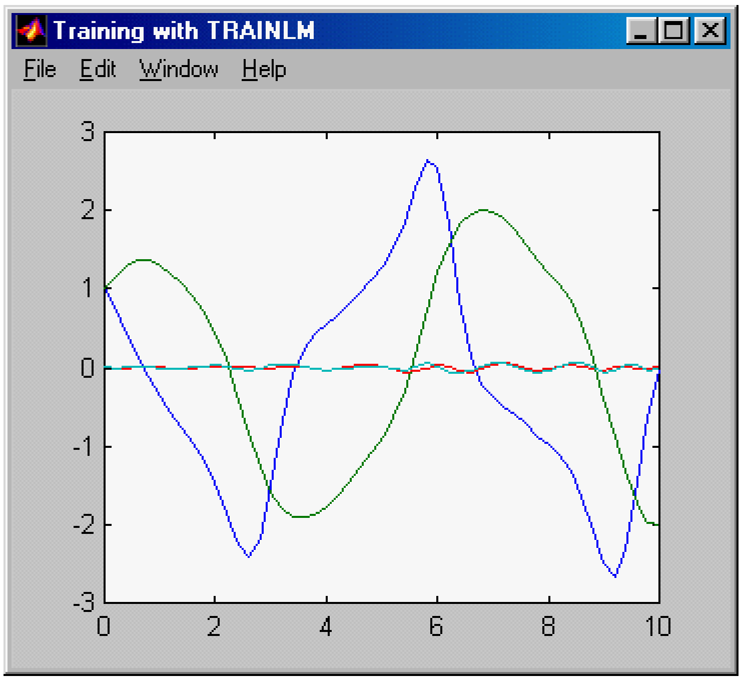
%Simulate result
a= sim(net1,P);

%Plot the result and the error
plot(P,a,P,a-T)
gi=input('Strike any key...');



The result is not very good, so let’s try to improve it.

Using 20 nodes in the hidden layer gives a much better result.



Try to further improve the multilayer perceptron solution.

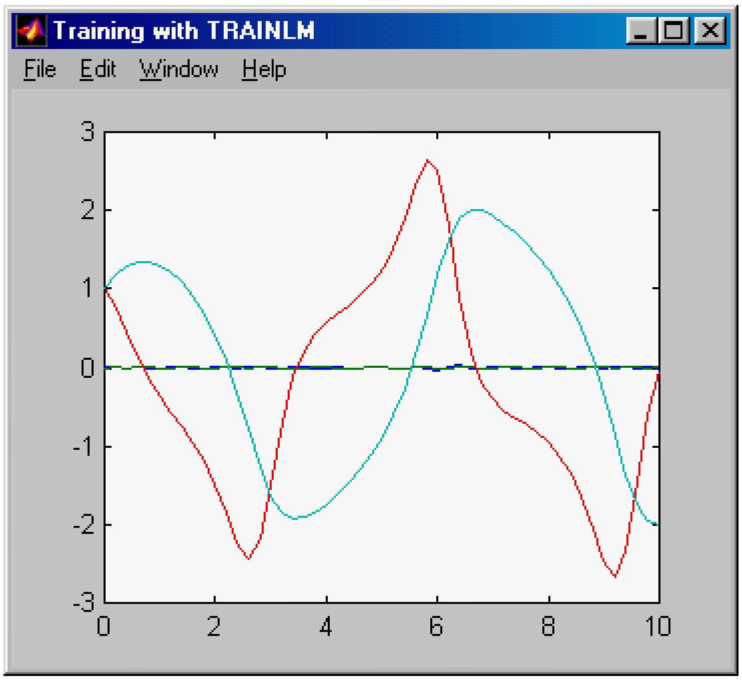
Use next radial basis function network with *spread = 0.01*.

net1=newrb(P,T,0.01);

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

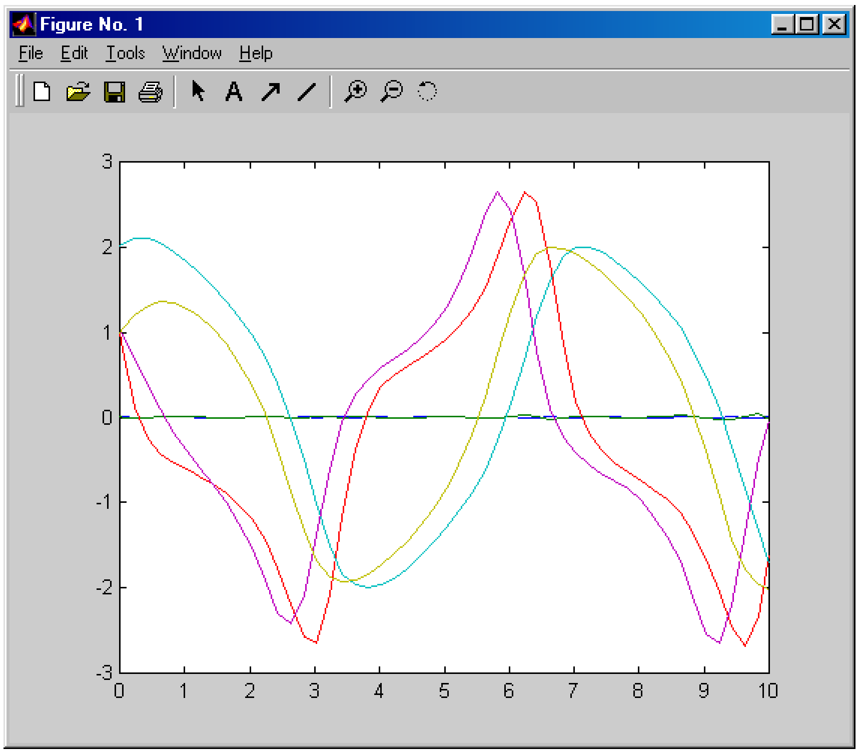
%Plot the result and the error
plot(P,a-T,P,T)

The result is displayed in the following figure.



The result is comparable with the previous one.

Try also other initial conditions. If $x_1(0) = 2$, and another net is fitted, with exactly the same procedure, the result is shown in the following figure.



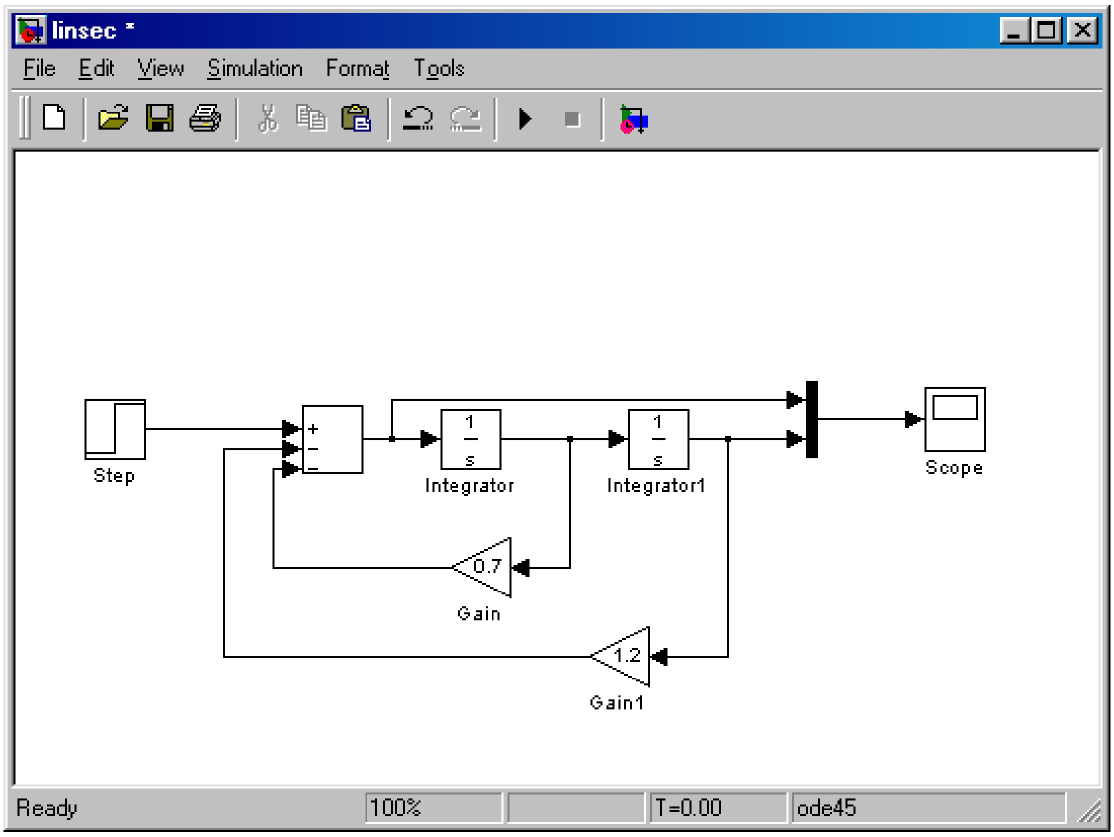
It is apparent that for a new initial condition, a new neural network is needed to approximate the result. This is neither very economic nor practical procedure. In example 6, a more general procedure is presented, which is often called *hybrid model*. The right-hand side of the state equation, or part of it, is approximated with a neural network and the resulting, approximating state-space model is solved.

EXAMPLE 5. Solve linear differential equation in the interval [0, 10].

$$\ddot{x} + 0.7\dot{x} + 1.2x = u(t)$$

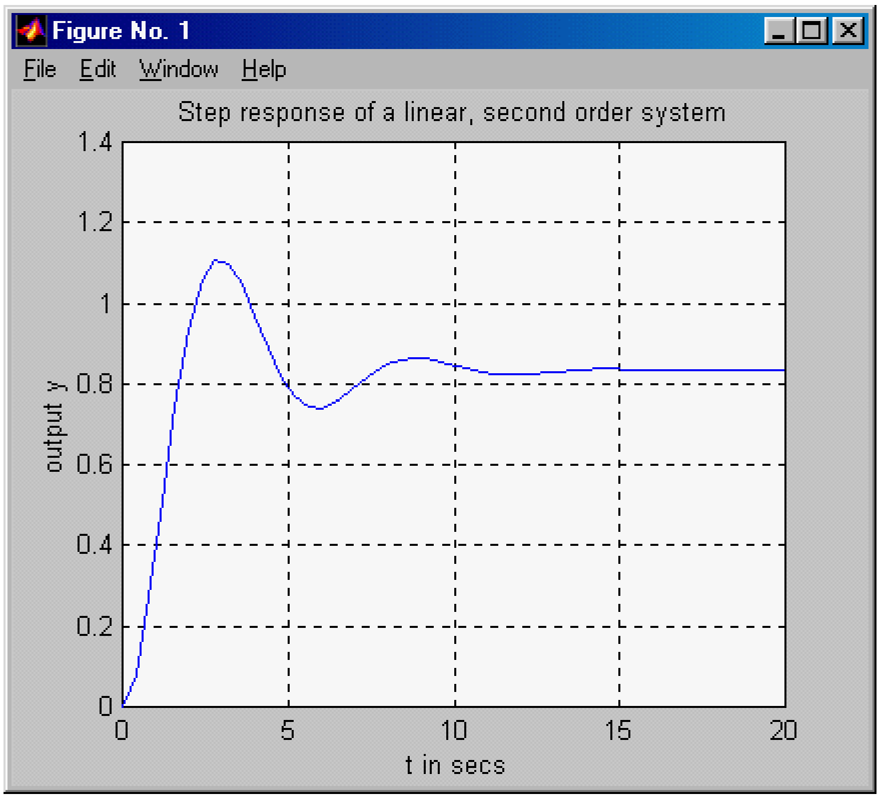
when *u(t)* is a unit step. The initial conditions are assumed to be zero. Assume also that the step takes place at t = 0 s. (In Simulink the default value is 1 s.). The output *x(t)* is your data.
Fit multilayer perceptron and radial basis function networks on the data and compare the result with the original data. Add noise to the data and do the fit. Does neural network filter data?

SOLUTION: Set up the SIMULINK configuration for the system. Call it *linearsec*. In simulation use fixed step, because then the number of simulation points remains in better control. If variable-step methods are used, automatic step-size control usually generates more data points than you want.



Go to MATLAB Command window. Simulate and plot the result.
[t1,x1]=sim('linsec',20);
plot(t1,x1(:,2))

xlabel(' t in secs');ylabel('output y'); title('Step response of a linear, second order system');
grid



Now you have input-output data *[t,y]*. Proceed as in the previous examples. Here only radial basis functions are used.

```
P = t1';
T = x1(:,2)';

plot(P,T,'r+');
title('Training Vectors');
xlabel('Input Vector P');
ylabel('Target Vector T');
gi=input('Strike any key ...');
```

Number of simulation data points and size of the networks should be carefully considered.

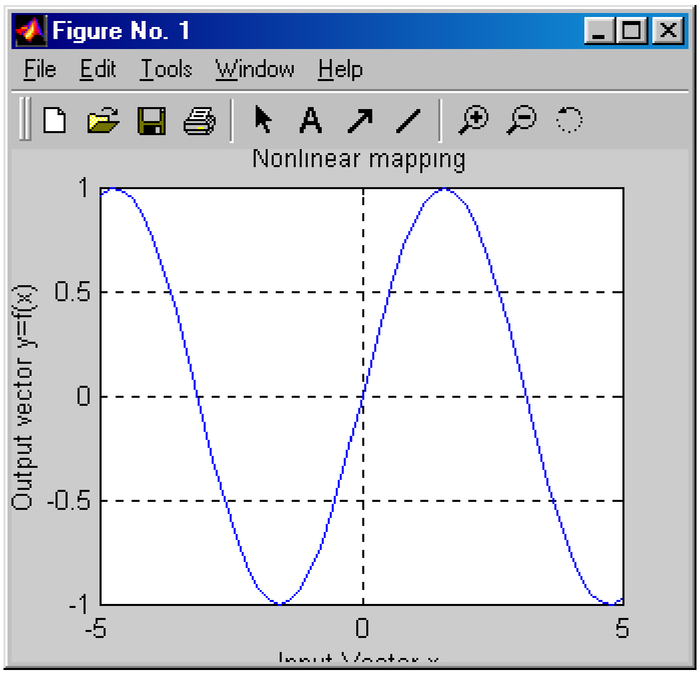
Complete the exercise.

EXAMPLE 6: Simulate the hybrid system

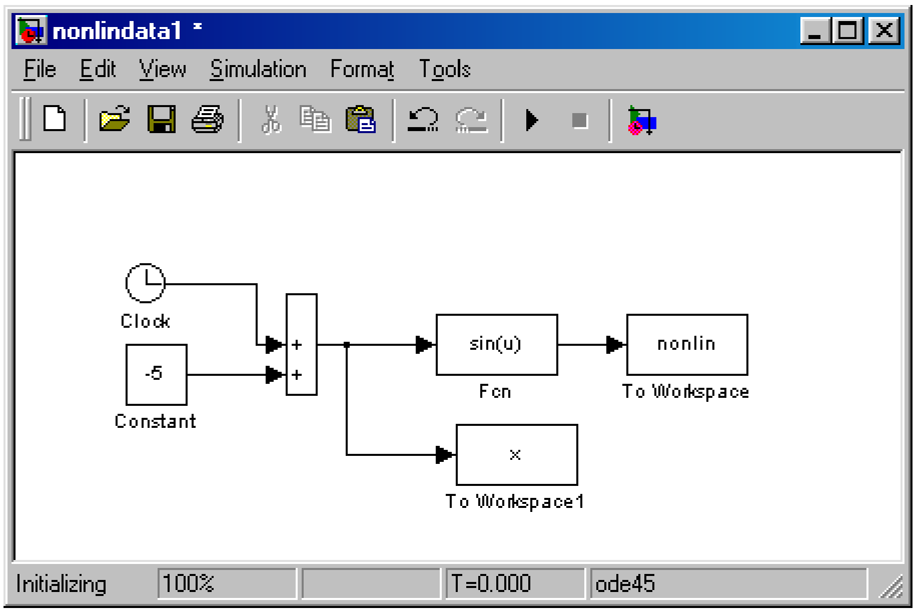
$$\ddot{x} = -\dot{x} - f(x)$$

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 $y=f(x)=sin(x)$.

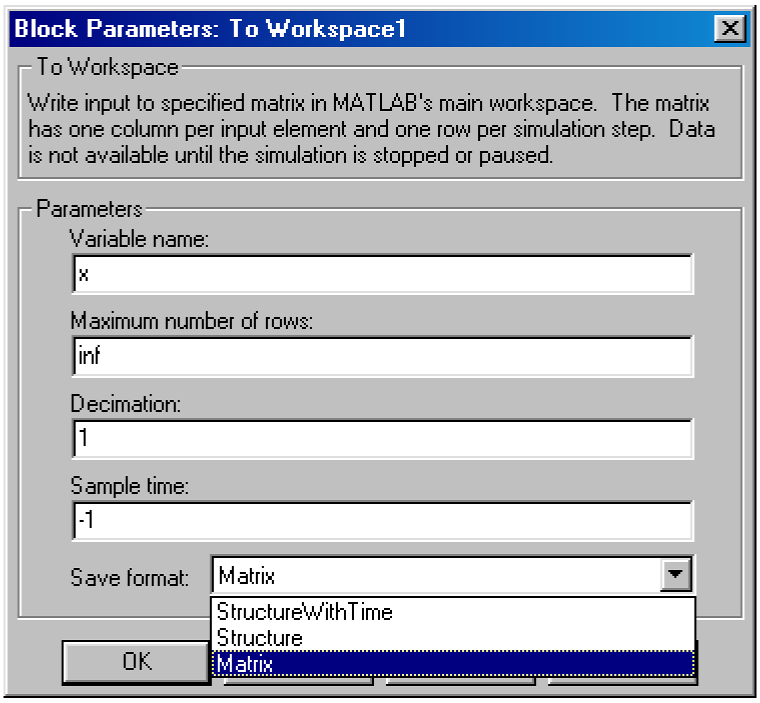
-5.0000	0.9589
-4.6000	0.9937
-4.2000	0.8716
-3.8000	0.6119
-3.4000	0.2555
-3.0000	-0.1411
-2.6000	-0.5155
-2.2000	-0.8085
-1.8000	-0.9738
-1.4000	-0.9854
-1.0000	-0.8415
-0.6000	-0.5646
-0.2000	-0.1987
0.2000	0.1987
0.6000	0.5646
1.0000	0.8415
1.4000	0.9854
1.8000	0.9738
2.2000	0.8085
2.6000	0.5155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



Write input to specified matrix in MATLAB's main workspace. The matrix has one column per input element and one row per simulation step. Data is not available until the simulation is stopped or paused.

Variable name: x

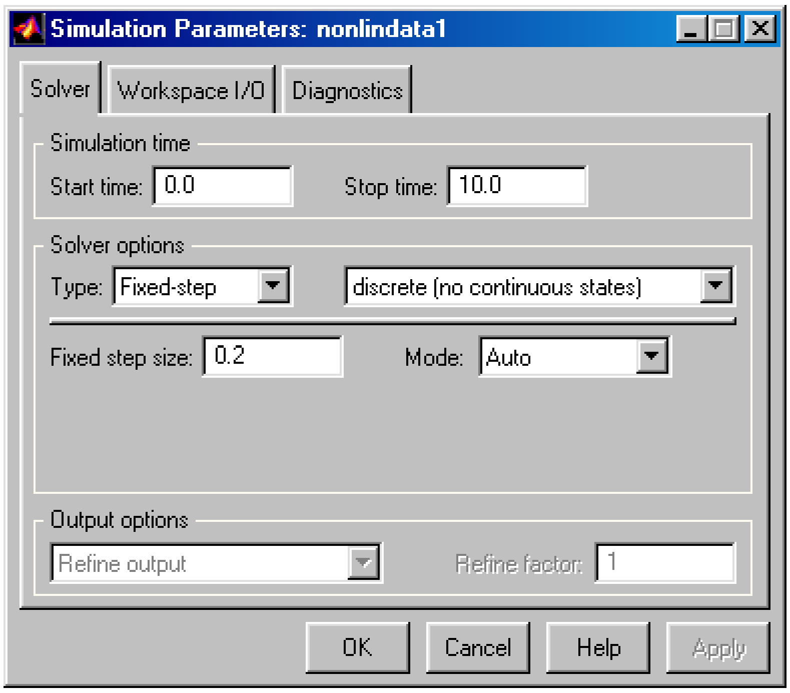
Maximum number of rows: inf

Decimation: 1

Sample time: -1

Save format: Matrix

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.



Simulation Parameters: nonlindata1

Solver | Workspace I/O | Diagnostics

Simulation time
Start time: 0.0   Stop time: 10.0

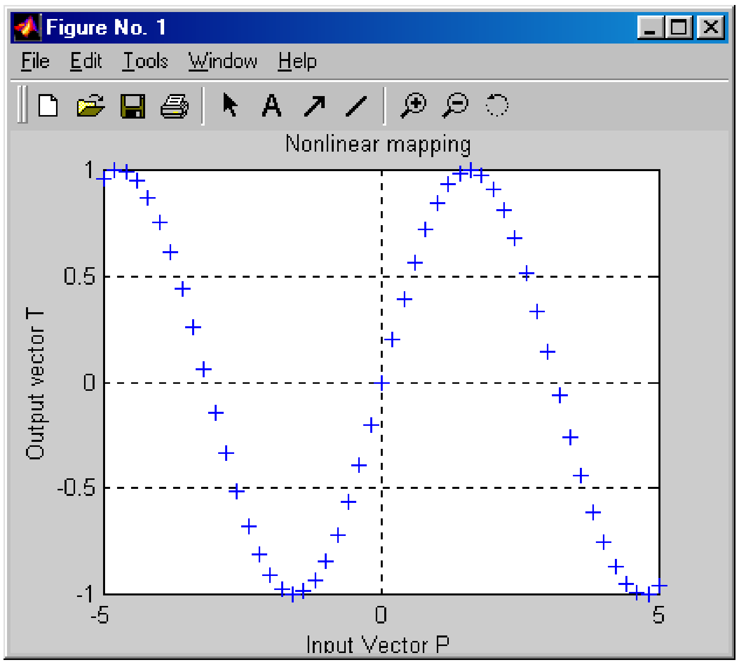
Solver options
Type: Fixed-step   discrete (no continuous states)
Fixed step size: 0.2   Mode: Auto

Output options
Refine output   Refine factor: 1

OK   Cancel   Help   Apply

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 ...');

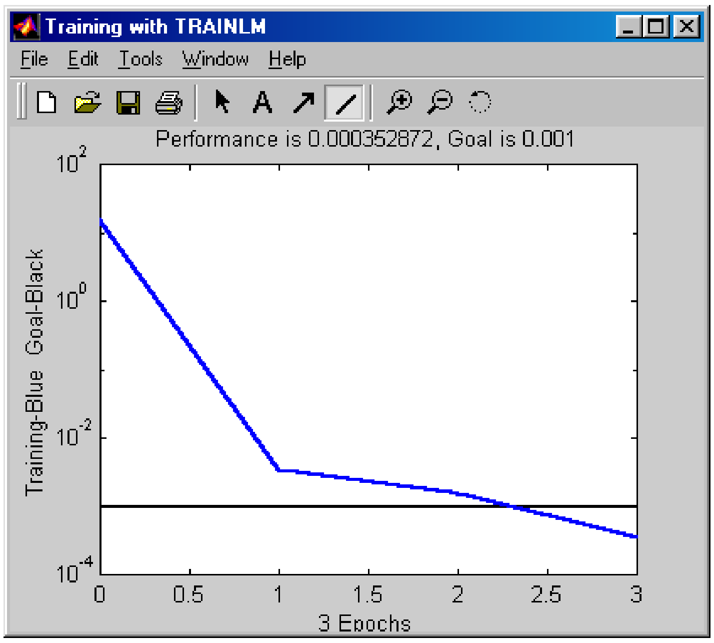


% LEVENBERG-MARQUARDT:
net=newff([-6 6], [20,1], {'tansig','purelin'},'trainlm');

```
%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);
```

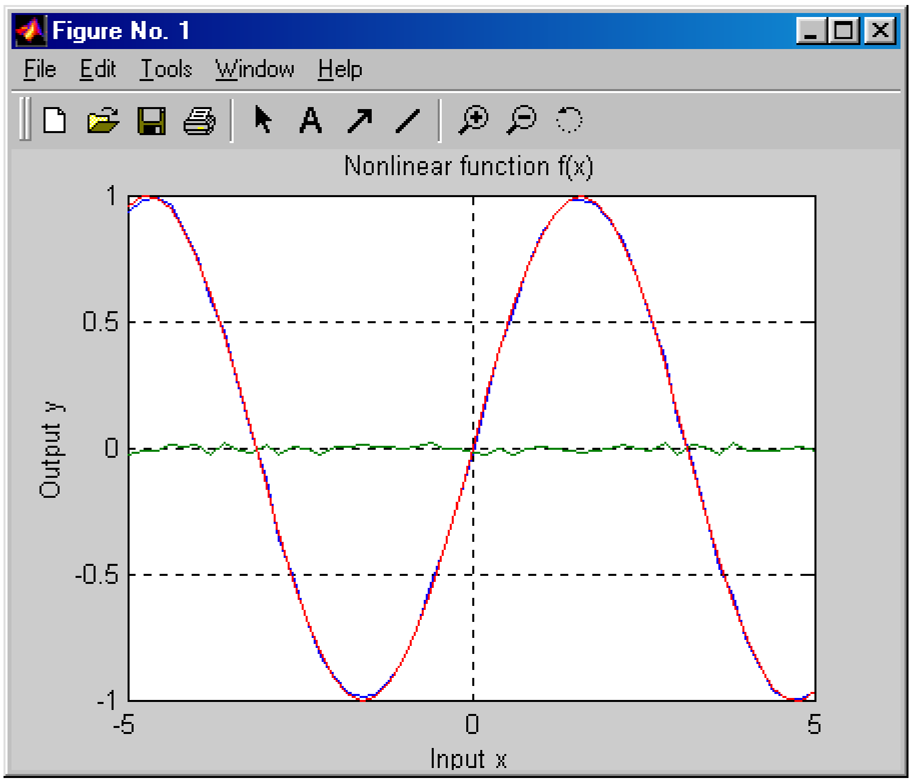
```
TRAINLM, Epoch 0/500, MSE 15.6185/0.001, Gradient 628.19/1e-010
TRAINLM, Epoch 3/500, MSE 0.000352872/0.001, Gradient 0.0423767/1e-010
TRAINLM, Performance goal met.
```

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



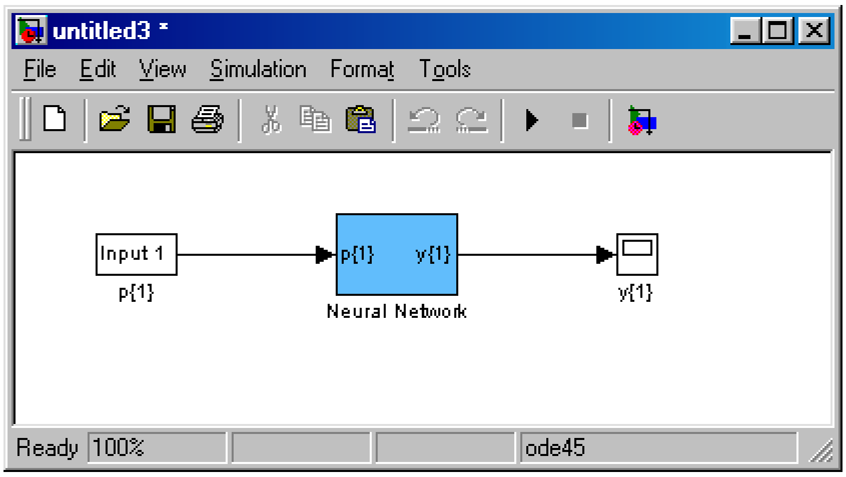
The result of the approximation by multilayer perceptron network is shown below together with the error.

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

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

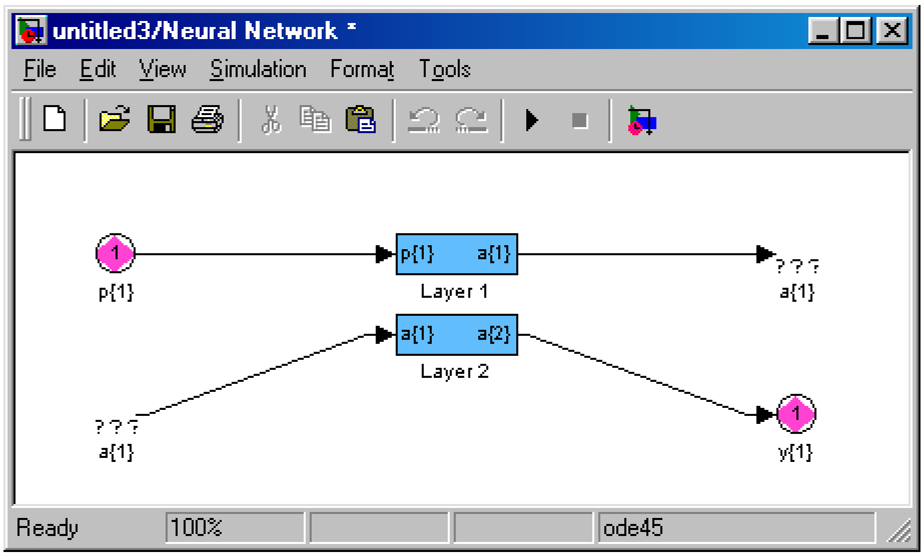


There is still some error, but let us proceed. Now we are ready to tackle the problem of solving the hybrid problem in SIMULINK. In order to move from the command side to SIMULINK use command *gensim*. This will transfer the information about the neural network to SIMULINK and at the same time it automatically 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.

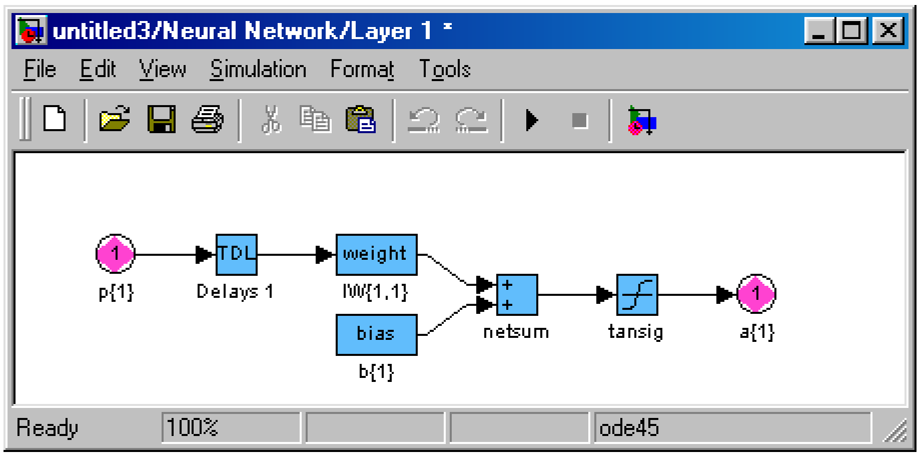
gensim(net1,-1)



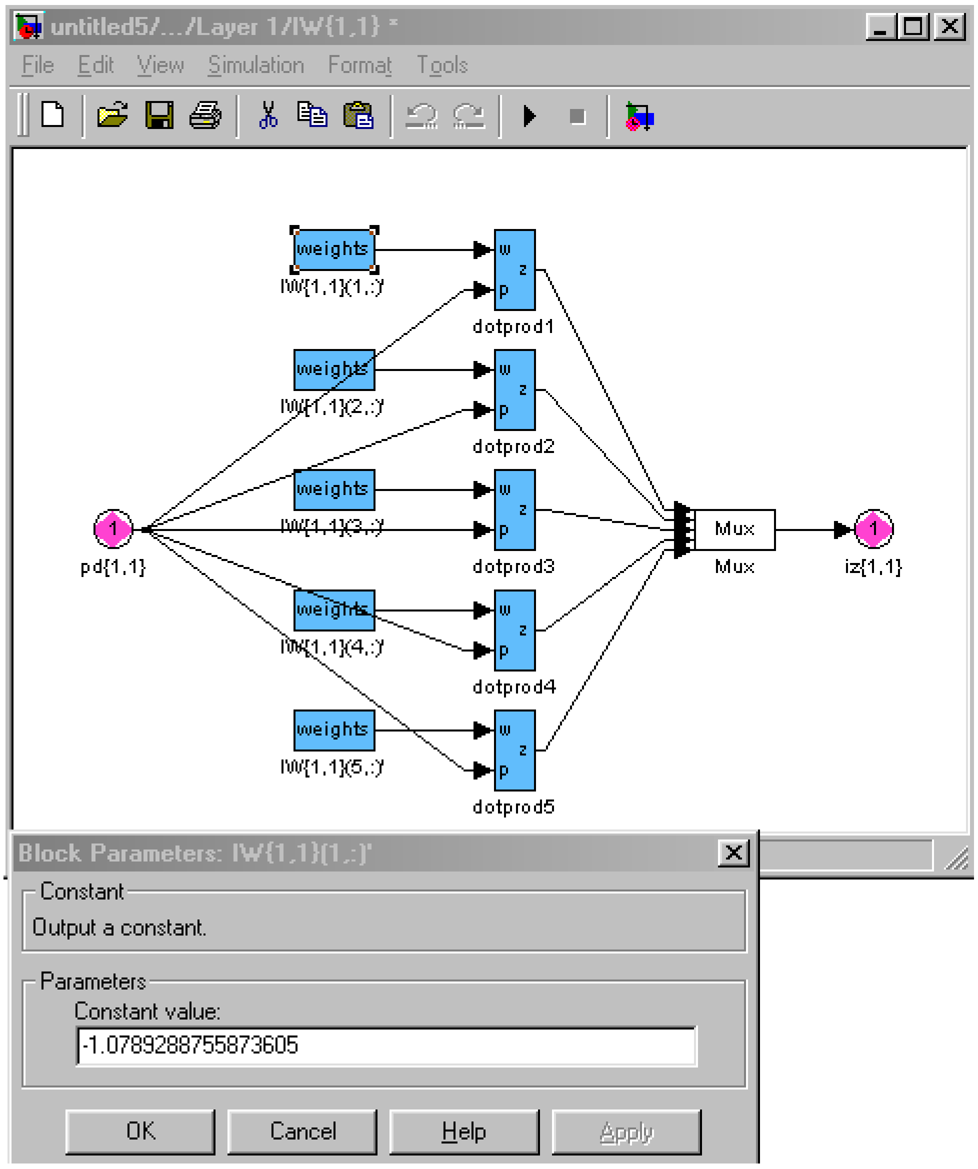
If you open the Neural Network block, you can see more details.



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.


untitled3/Neural Network/Layer 1 *
File Edit View Simulation Format Tools
1
p{1}
TDL
Delays 1
weight
IW{1,1}
bias
b{1}
+
+
netsum
tansig
1
a{1}
Ready
100%
ode45

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.



Block Parameters: IW{1,1}(1,:)'

Constant

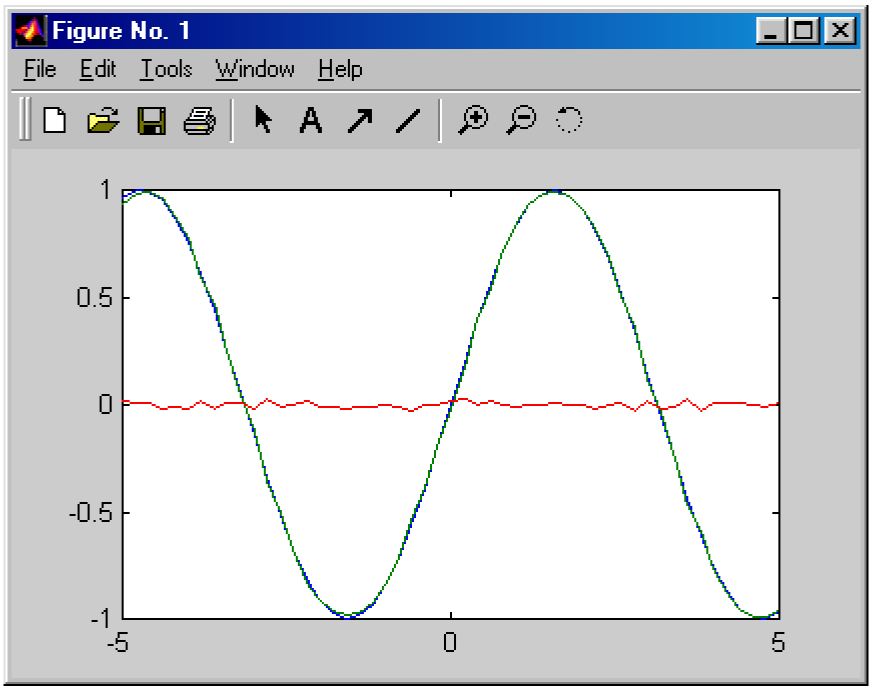
Output a constant.

Parameters

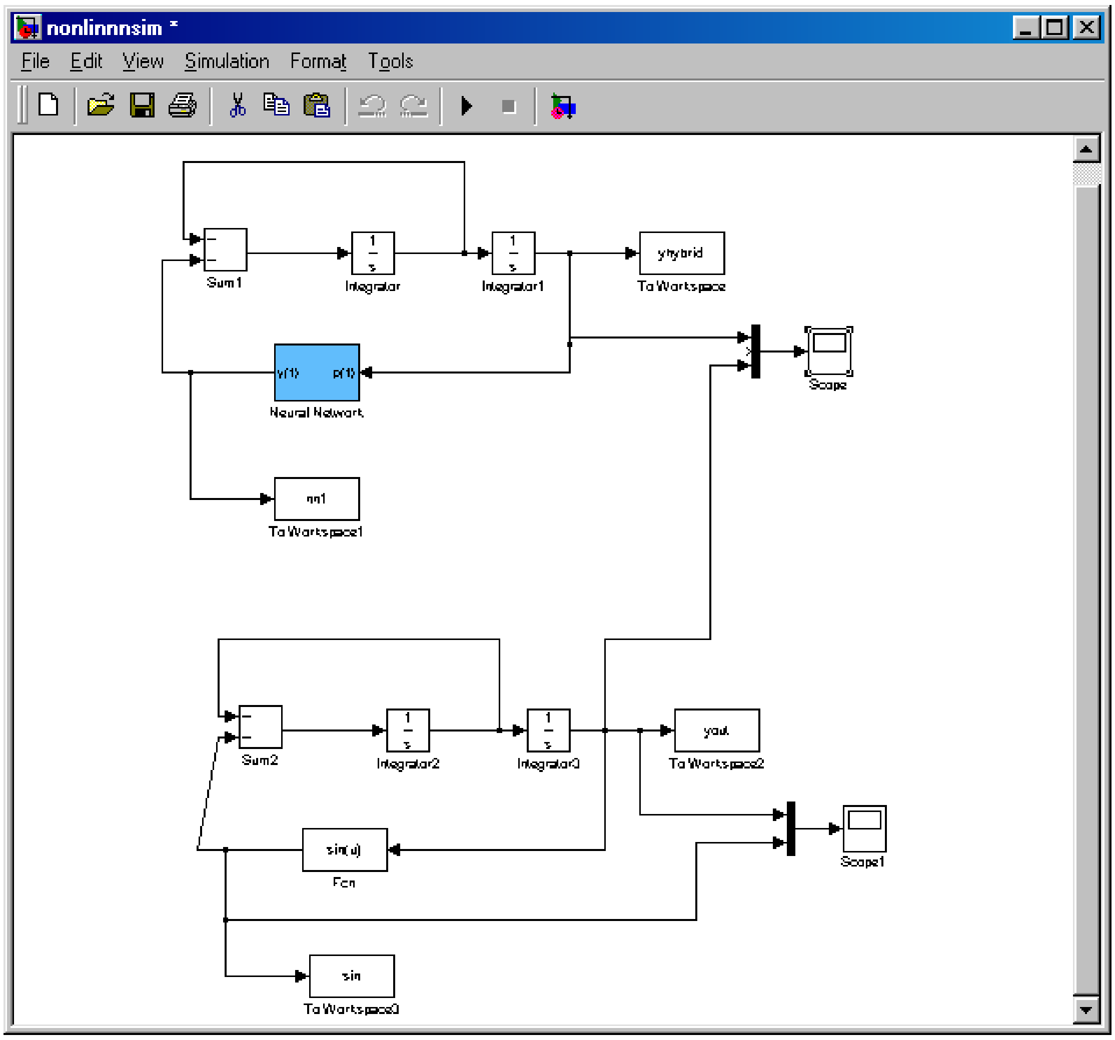
Constant value:

-1.0789288755873605

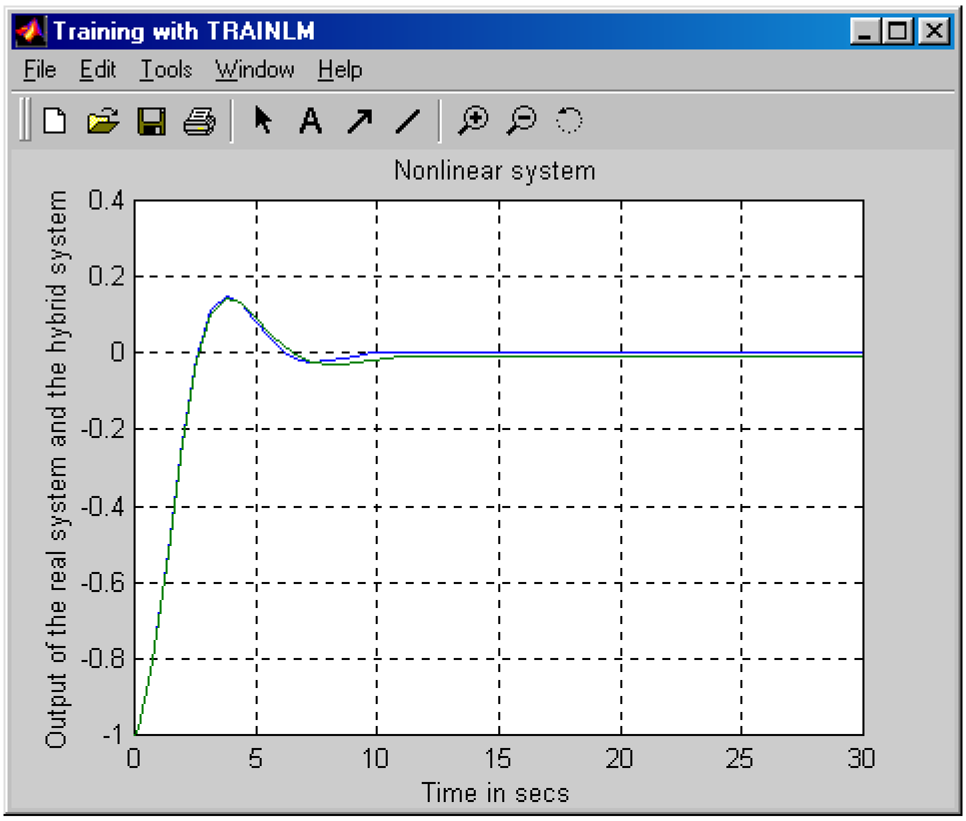
To convince ourselves that the block generated with *gensim* really approximates the given data, we'll feed in values of x in the range [-5,5]. Use the following SIMULINK configuration.



Now simulate the system.

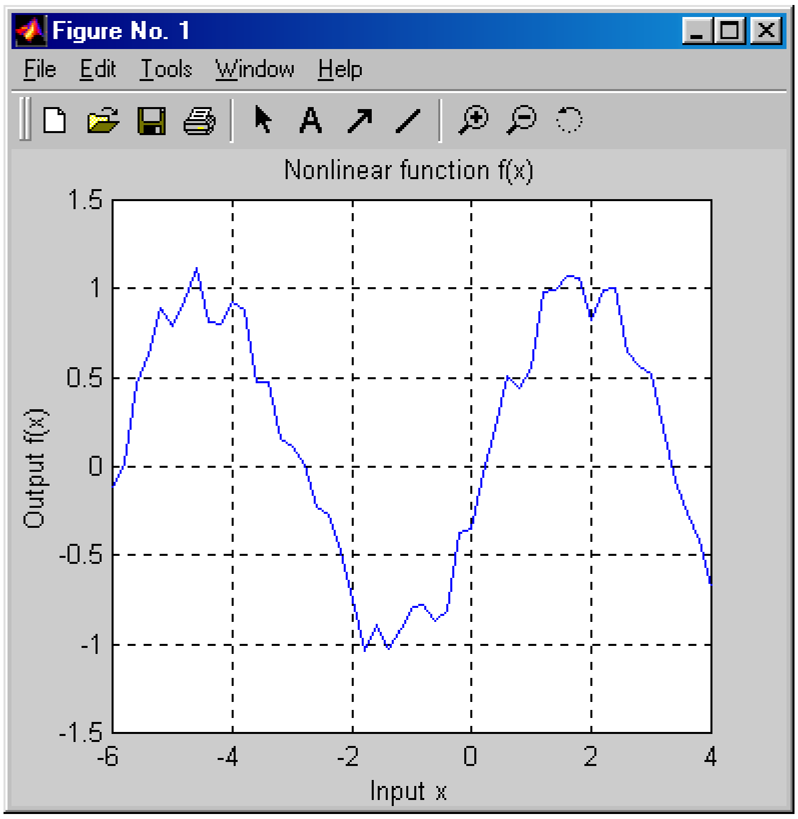


The result is plotted below.
plot(tout,yout,tout,yhybrid)
title('Nonlinear system'); xlabel('Time in secs');
ylabel('Output of the real system and the hybrid system'); grid;

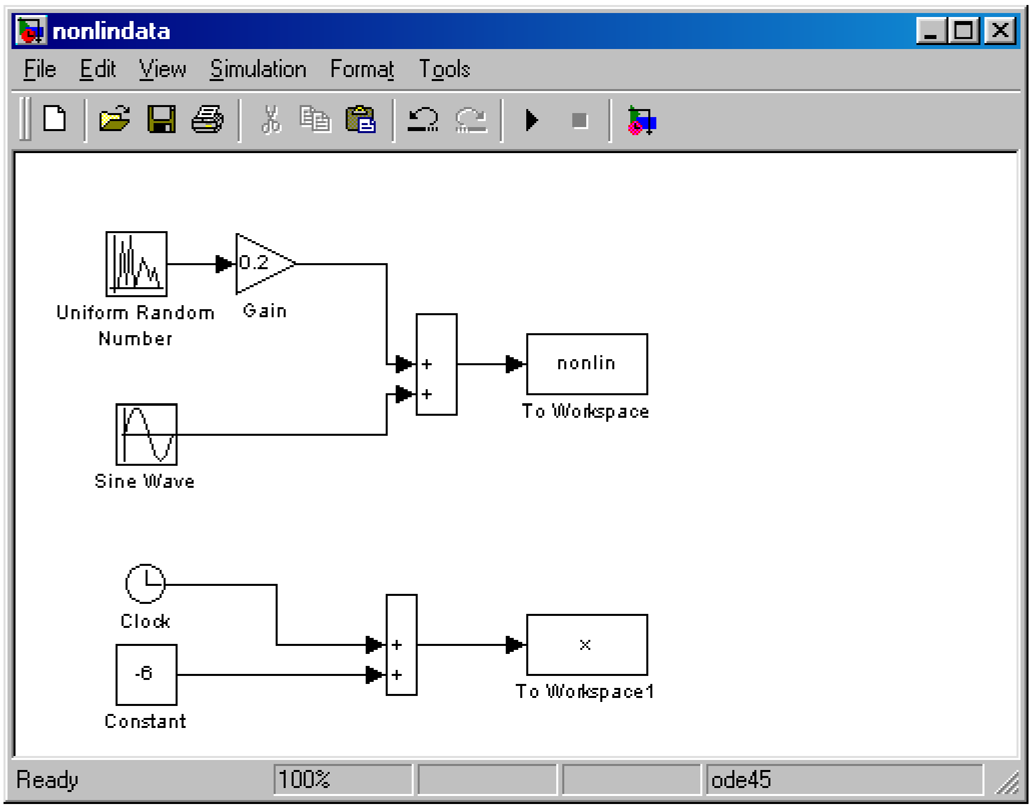


Careful study shows that some error remains. Further improvement can be obtained either by adding the network size or e.g. tightening the error tolerance bound.

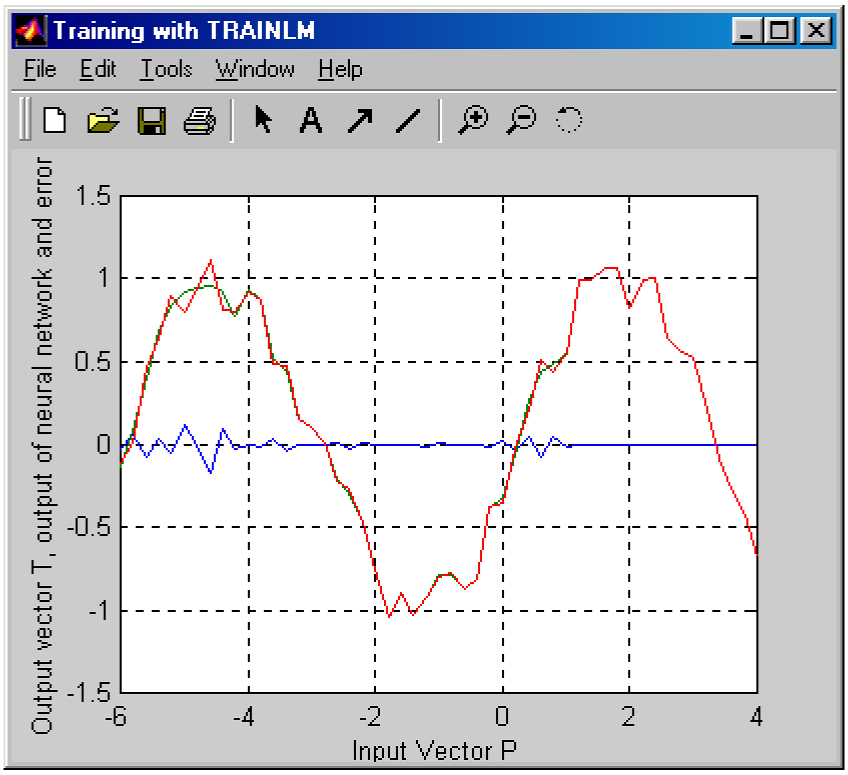
If the data is noise corrupted the procedure is the same, except that it is recommended that data preprocessing is performed. The data is shown below.



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.

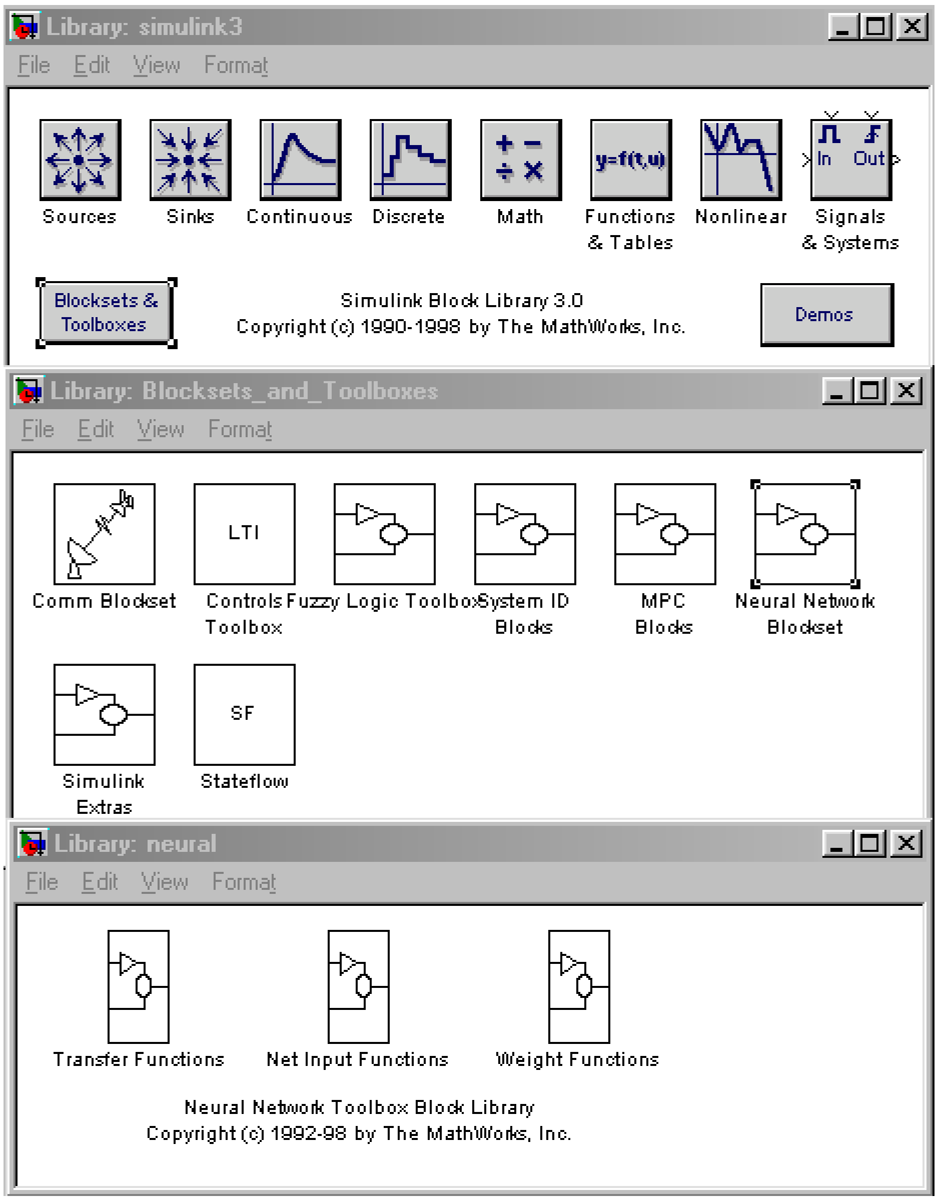


As can be seen in the figure, there is still error, but since the given data is noisy, we are reasonably happy with the fit. Preprocessing of data, using e.g. filtering, should be carried out before neural network fitting.

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.

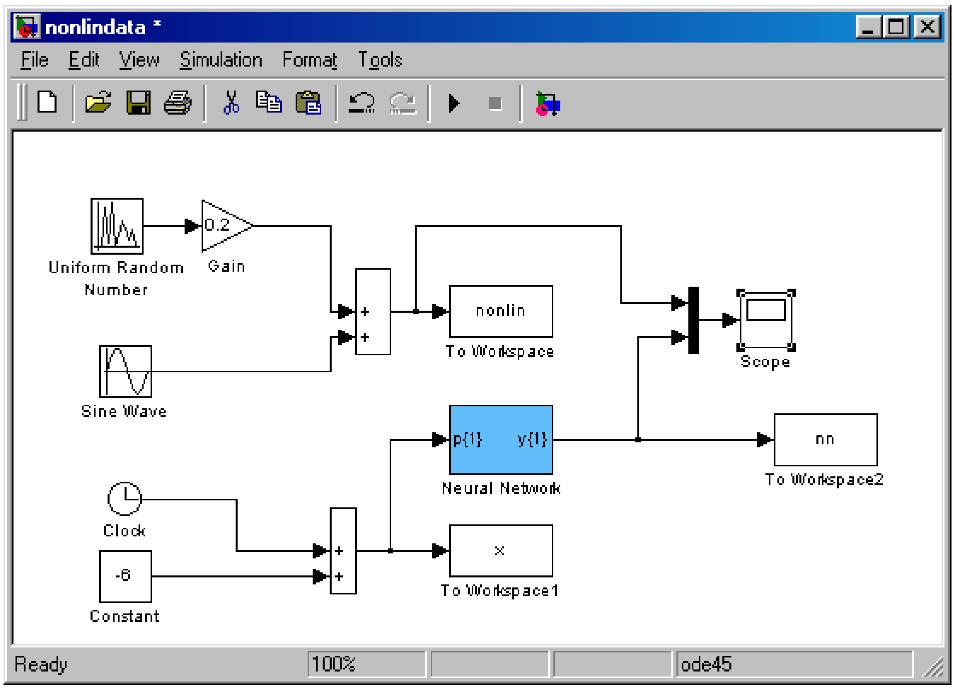


Simulink Block Library 3.0
Copyright (c) 1990-1998 by The MathWorks, Inc.

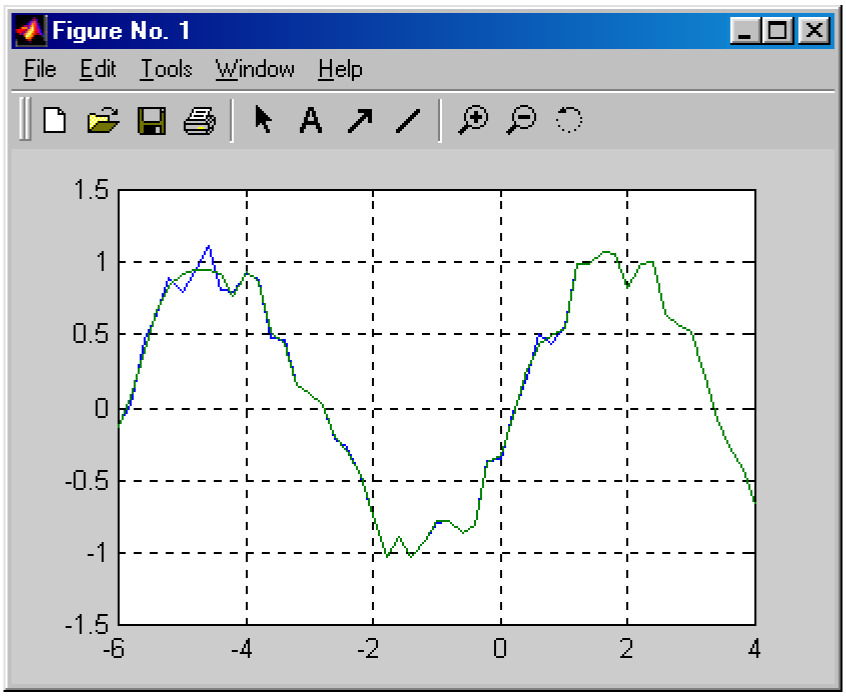
Neural Network Toolbox Block Library
Copyright (c) 1992-98 by The MathWorks, Inc.

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.

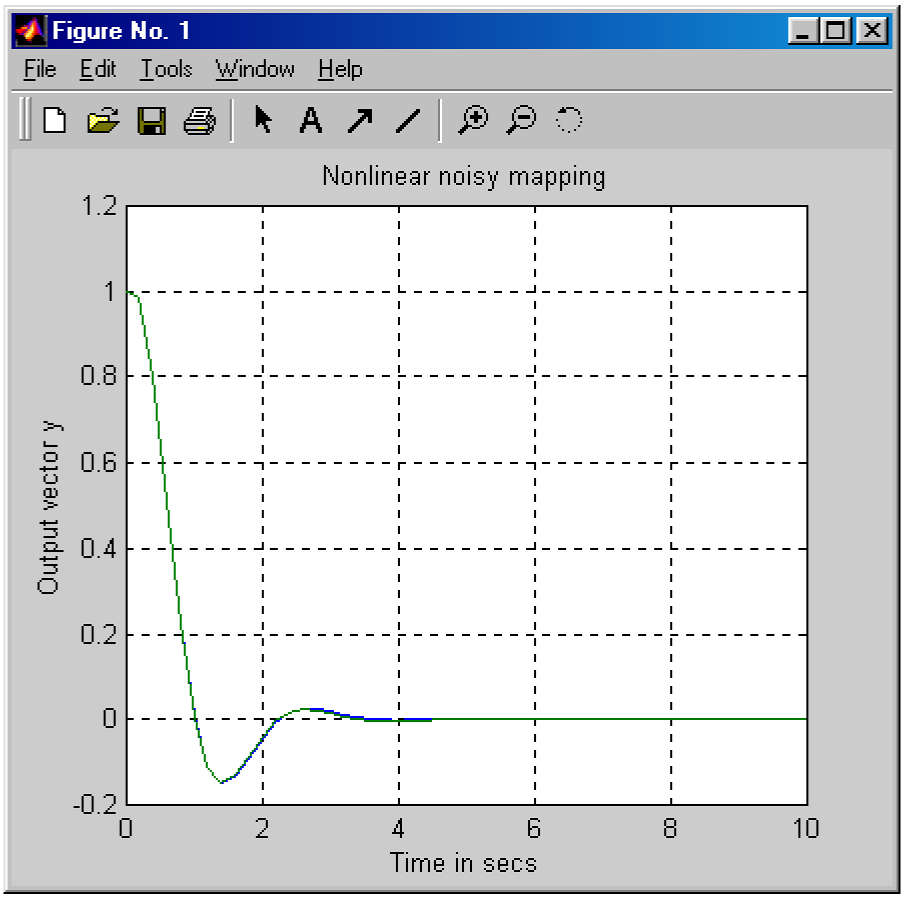
To convince ourselves that the block generated with *gensim* really approximates the given data, we'll feed in values of x in the range [-6,6]. Use the following SIMULINK configuration for comparison.



The result is quite satisfactory.



Now simulate the system.



Simulation result looks quite reasonable.

PROBLEM 1. Function approximation.

a. Generate by whatever means input data for the function

$$y(x) = x^2 + 3x$$

when

$$-4 \leq x \leq 4 .$$

b. Plot the data.

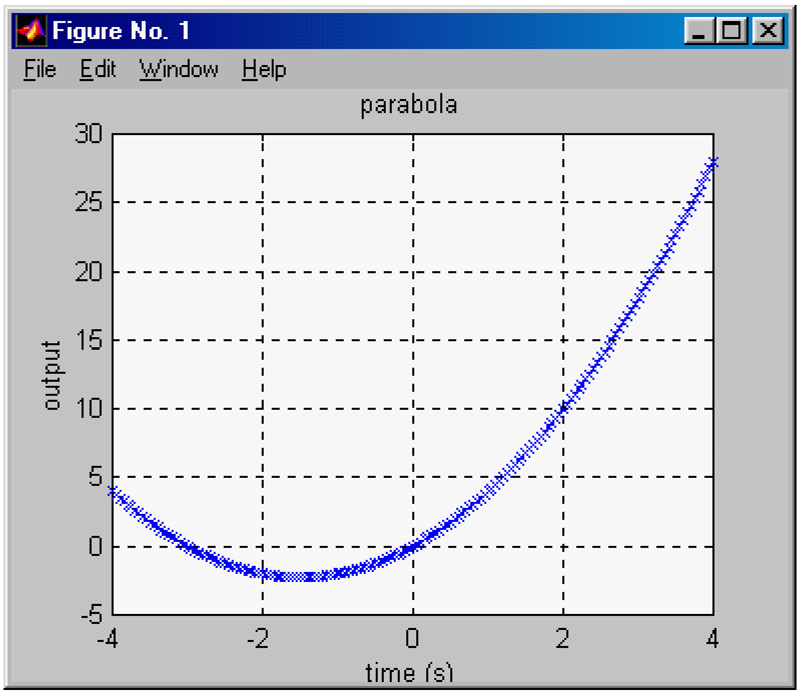
c. Fit multilayer perceptron and radial basis function networks on the data and compare with the original.

SOLUTION:

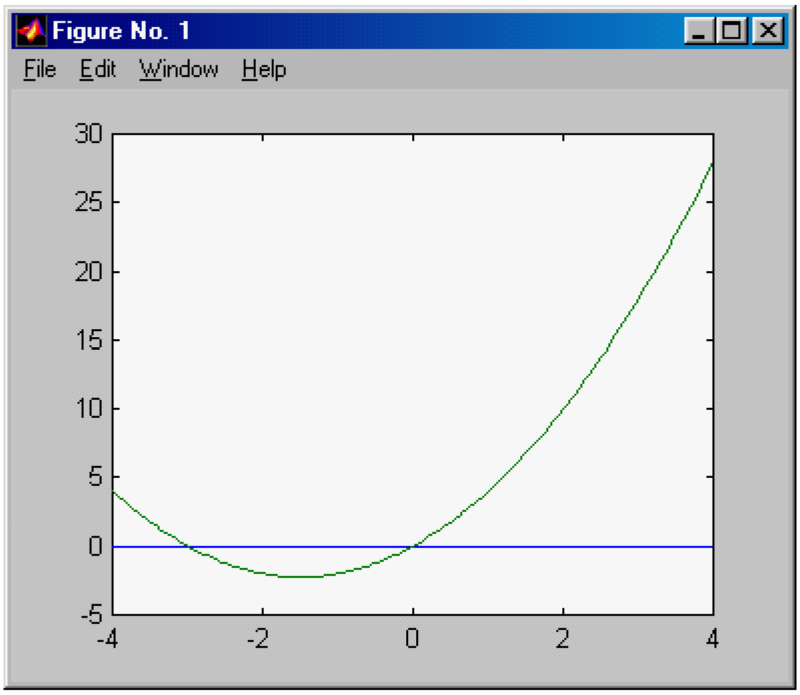
```
x=-4:0.05:4; y=x.*x+3*x;
P=x;T=y;
```

Plot the data

```
plot(P,T,'o')
 grid;  xlabel('time (s)'); ylabel('output');  title('parabola')
```



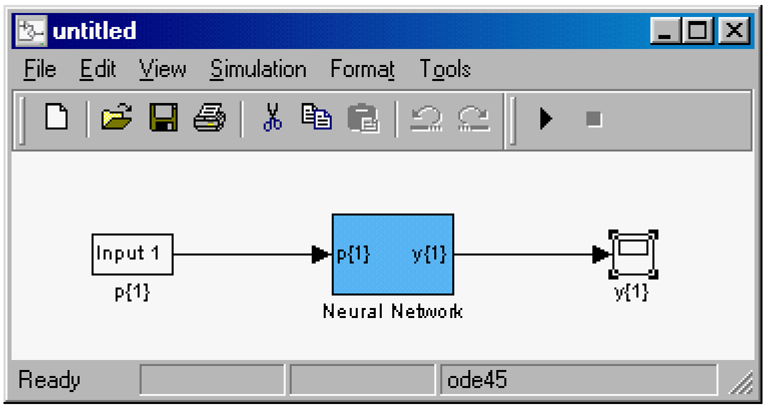
```
net1=newrb(P,T,0.01);

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

%Plot the result and the error
plot(P,a-T,P,T)
```



Generate the corresponding SIMULINK model.

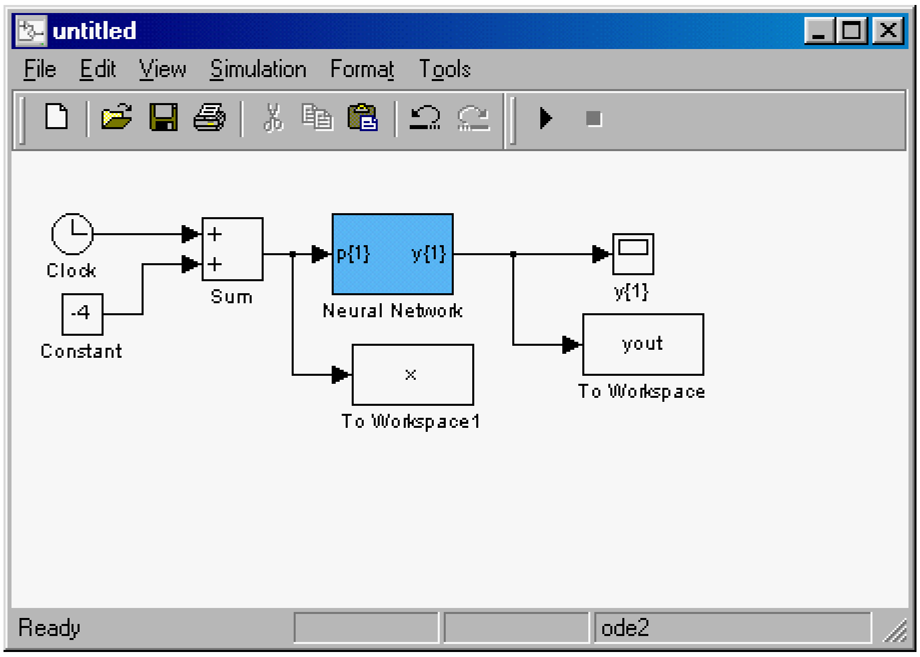
gensim(net1,-1)

This results in the following configuration

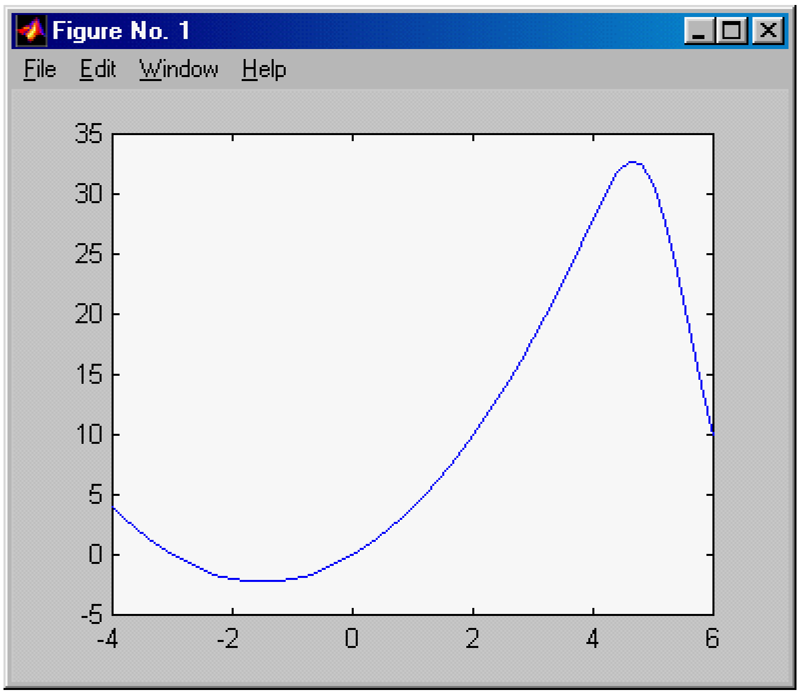


Change the input so that it will cover the range [-4, 4]. One way to do it is shown below.
You need to open SIMULINK by typing Simulink in MATLAB command side.

Change also the simulation parameters to fixed parameter simulation. The method is not critical. Use e.g. Heun’s method.

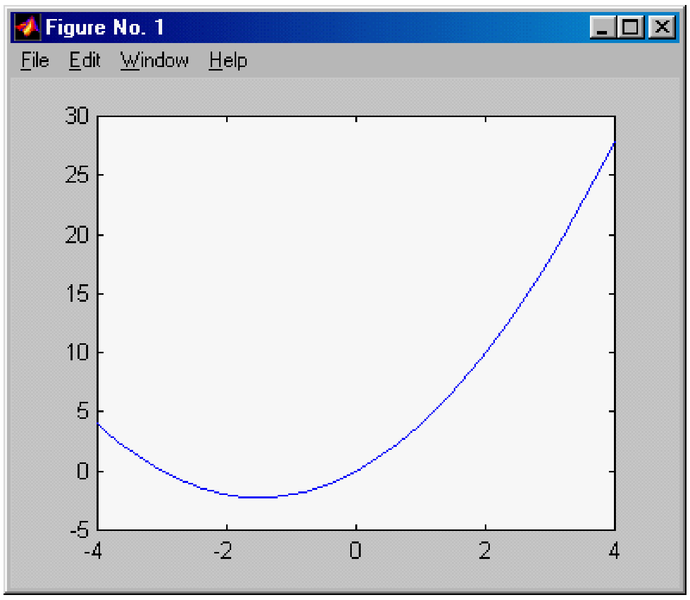


When simulated and plotted on MATLAB command side, *plot (x,yout)* results in



The result does not look like a parabola, but a closer examination reveals that in the interval [-4,4] the approximation is fine. Because the default value of simulation is 10 s, it can be observed that from –4 we go to up to 6 s, which is of course outside the range. Therefore the simulation should be limited only to 8 s.

Now the figure looks all right.



PROBLEM 2. Consider

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

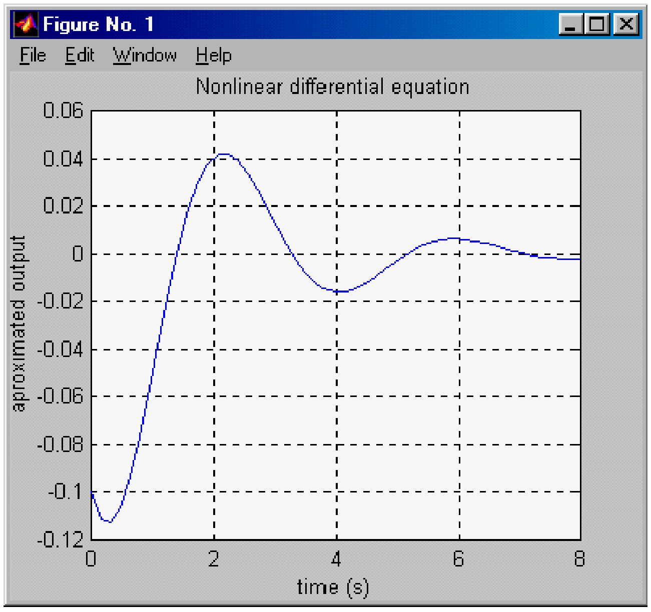
when $f(x) = x^2 + 3x$.

a. Simulate the *system response* for exact *f(x)* and the neural network approximation. Use different initial conditions. Compare the results.

b. Add noise to *f(x)*, $f(x) = x^2 + 3x + noise$.
Noise is bandlimited white noise (default from Simulink block).

SOLUTION: Continuation from problem 1: We have verified in SIMULINK that the neural network approximation is good. Let us use it in differential equation solution.

plot(x,ynn)
grid; xlabel('time (s)'); ylabel(' aproximated output'); title('Nonlinear differential equation')



PROBLEM 3. Repeat problem 1 using ***Kohonen's Self-Organizing Map (SOM).***

SOLUTION: Generate parabolic data and plot it. Note that input *x* and output *y* are now combined and put in matrix *P*.

```
x=-1:0.05:1; y=x.*x/3+2;
P = [x; y];
plot(P(1,:),P(2,:),'+r')
```

Define the SOM network. Try a simple one. The map will be a 1-dimensional layer of 10 neurons.

```
net=newsom([-1 1;0 1],[10]);
```

The first argument specifies two inputs, each with a range of 0 to 1.
The second determines the network is one dimensional with 10 neurons.

Define parameters in the algorithm. First use only the basic scheme, in which the maximum number of iterations is determined.

```
net.trainParam.epochs =1000;
```

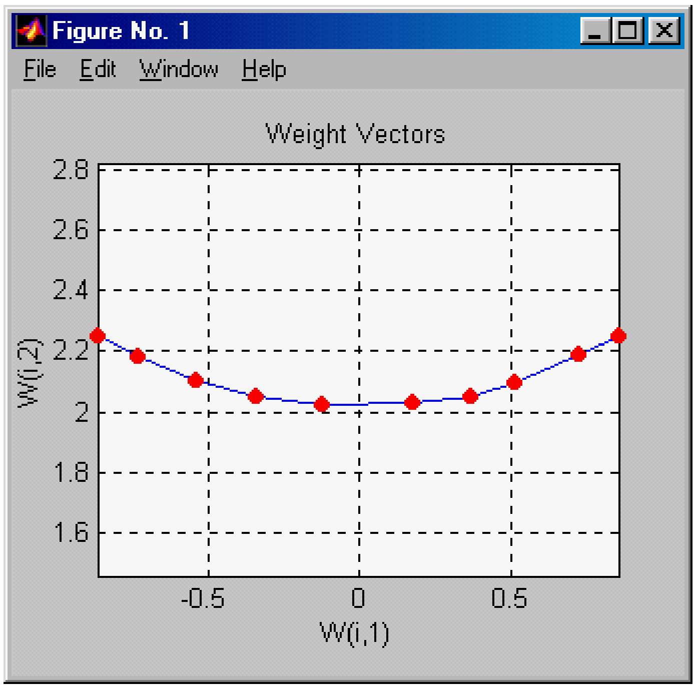
Train network by calling train
```
net1=train(net,P);
```

Now plot the trained network with PLOTSOM:
```
plotsom(net1.iw{1,1},net1.layers{1}.distances)
```

Next simulate result
% The map can now be used to classify inputs, like [1; 0].

```
a=sim(net,[1;0])
```

% Either neuron 1 or 10 should have an output of 1, as the above
% input vector was at one end of the presented input space.
% The first pair of numbers indicate the neuron, and the single indicates its output.



PROBLEM 4. Neural function networks are good function approximators, when the function to be approximated is *smooth* enough. This means that function is at least continuous on compact set, because then the Weierstrass theorem applies. (Check your calculus book, if you have forgotten what this theorem is all about).

What has not been studied is how well they suit for approximating hard nonlinearities.

Use SIMULINK to generate input-output data for

a. saturation nonlinearity and
b. relay.

Determine, which neural network structure would be best for the above.
HINT: For deadzone nonlinearity the data generation can be done using sin-function input for appropriate time interval.

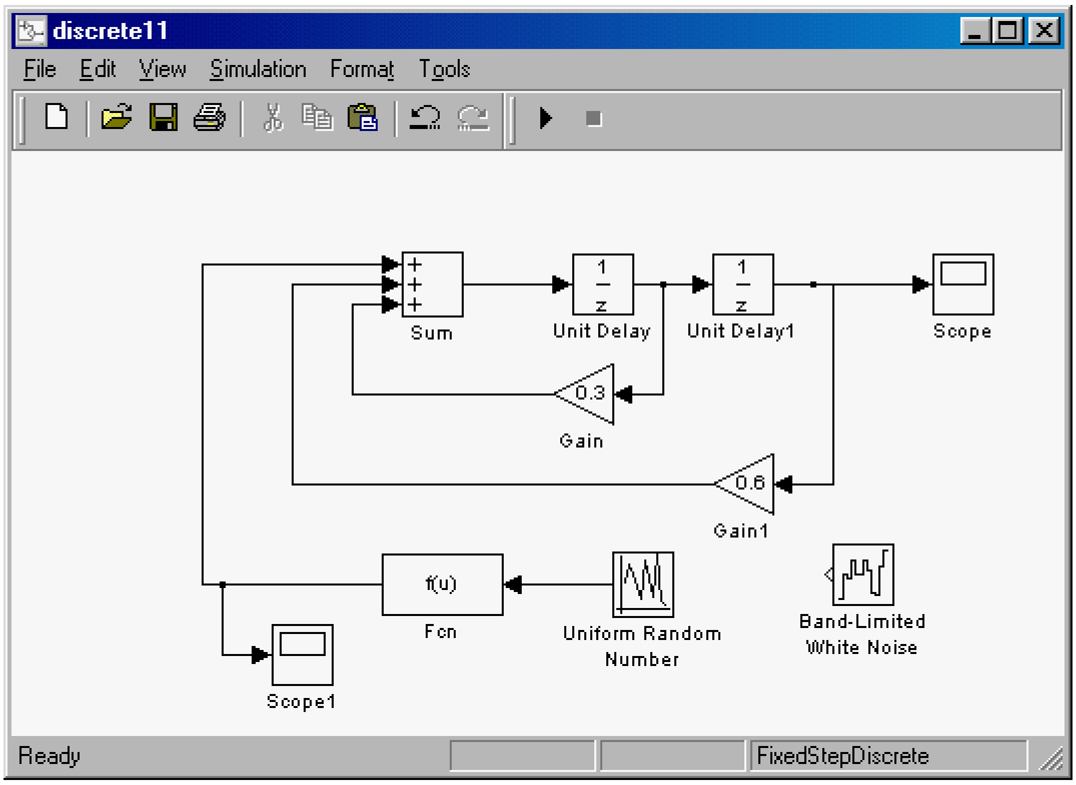
PROBLEM 5. A system is described by a first order difference equation

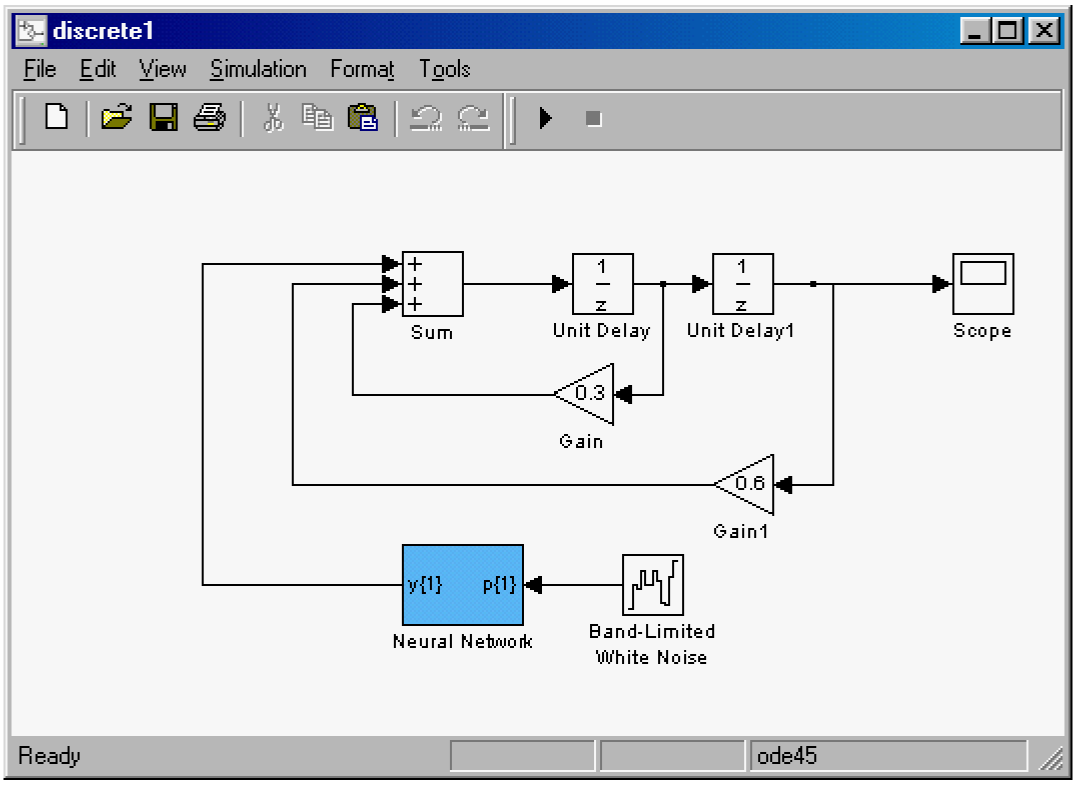
$$y(k+2) = 0.3y(k+1) + 0.6y(k) + f(u(k))$$

where $f(u(k)) = u^3 + 0.3u^2 - 0.4u$ and *u(k)* is random noise.

c. Generate data in appropriate range for *f(u)* and fit a neural network on the data.
d. Simulate the system response for exact *f(u)* and the neural network approximation. Compare the results. Initial condition could be zero. Input can be assumed to be noise.

SOLUTION:



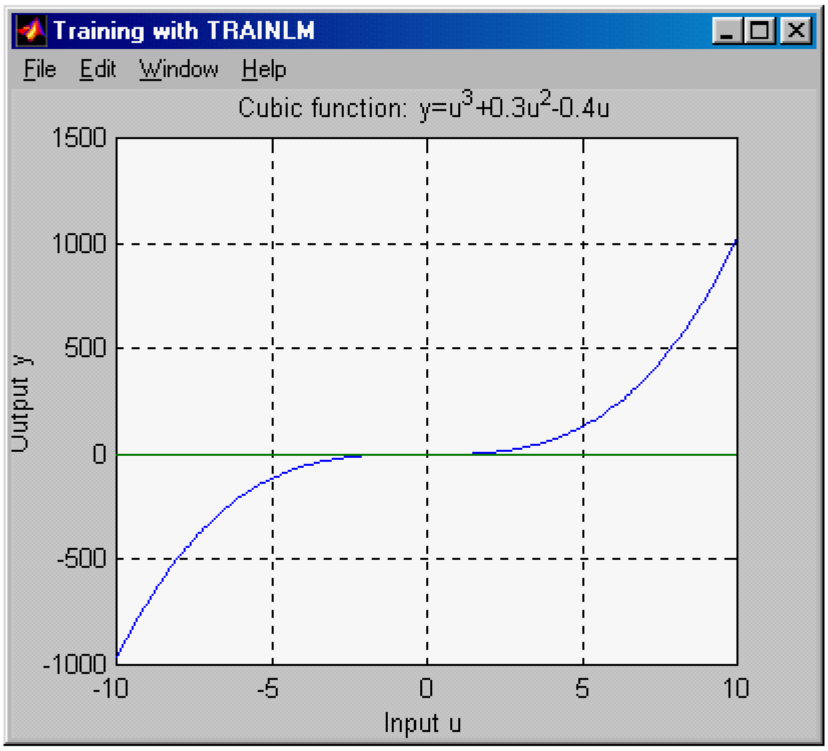


The neural network approximation can be obtained in various ways, e.g., analogously to Example 1, using the following commands

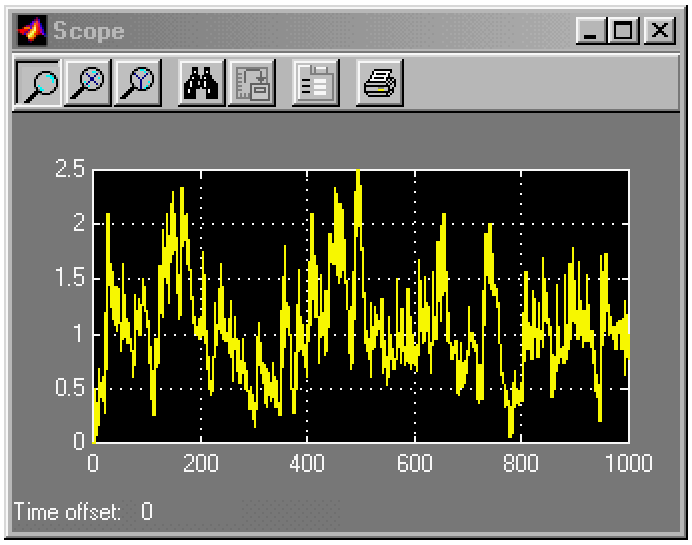
```
%Generate parabolic data

u=-10:0.1:10; y=u.^3+0.3*u.^2-0.4*u;
P=u;T=y;

%Define network
net=newff([-10 10], [10,1], {'tansig','purelin'},'trainlm');

%Define parameters
net.trainParam.show = 50;
net.trainParam.lr = 0.05;
net.trainParam.epochs = 300;
net.trainParam.goal = 1e-3;

%Train network
net1 = train(net, P, T);
```

Plot the result

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

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

Title('Cubic function: y=u^3+0.3u^2-0.4u')
xlabel('Input u');ylabel('Output y')
```



Result of simulation – uniform random number, exact *f(u).*



Questions about simulation:

WHAT TO DO IF INPUT GOES OUT OF RANGE? What is the default action in SIMULINK? Scaling (analog computation – since you are now faced with analog components)

What about noise – random numbers or bandlimited noise (SIMULINK problem)

PROBLEM 4. Consider

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

when $f(x) = x^2 + 3x$.

a. Generate data in appropriate range for *f(x)* and fit a neural network on the data.
b. Simulate *system response* for exact *f(x)* and the neural network approximation.
   Use different initial conditions.
   Compare the results.

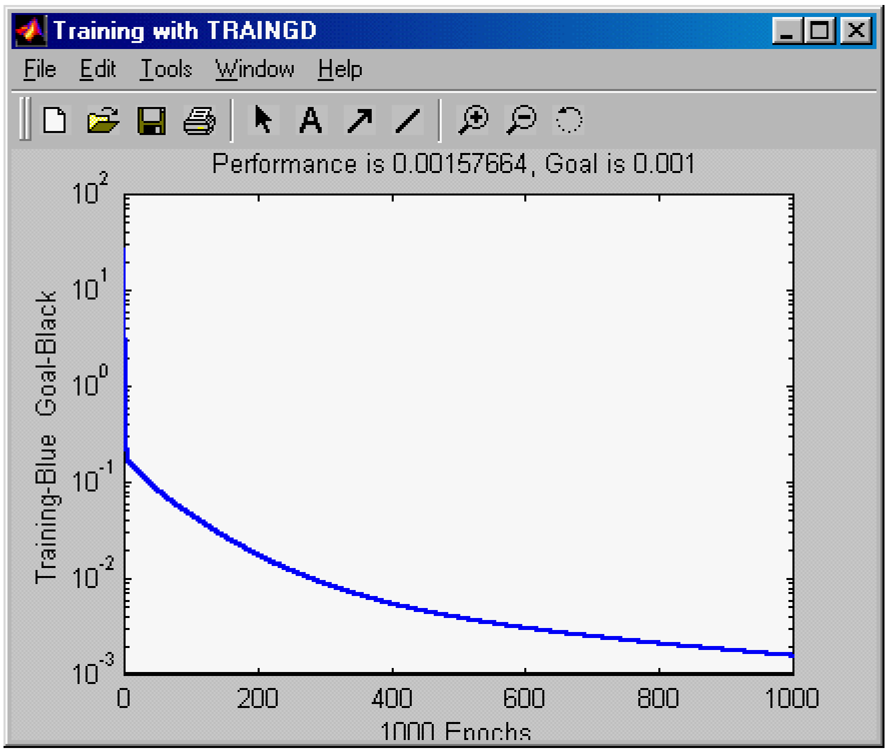
SOLUTION: Continuation from problem 1: We have verified in SIMULINK that the neural network approximation is good. Use it in the differential equation simulation.

PROBLEM 6. Fit a multilayer perceptron network on *y = 2x.*

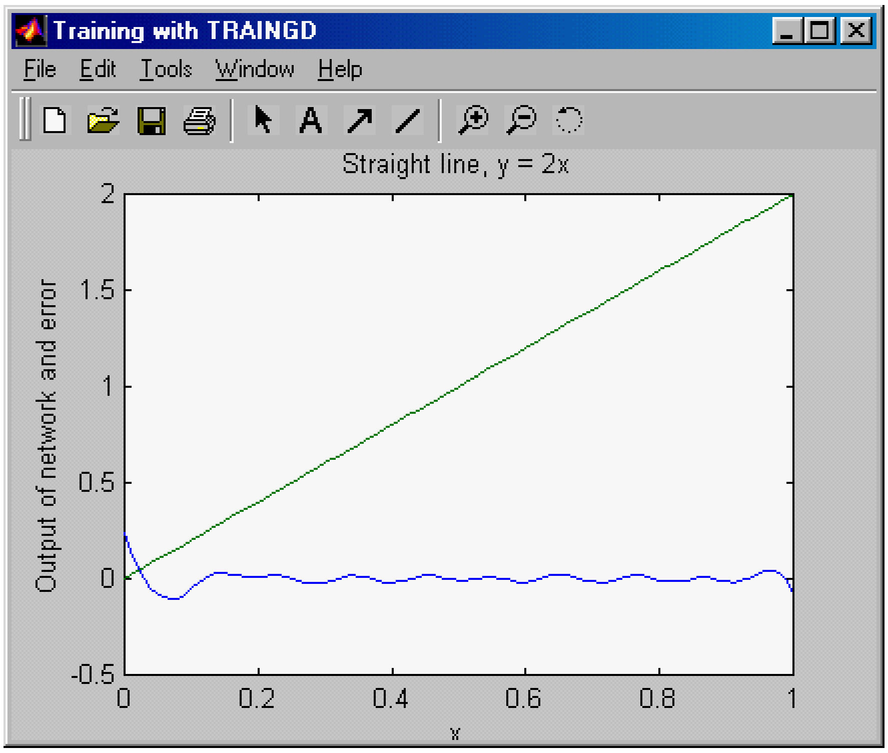
Prune the network size with NNSYSID.

SOLUTION:

```
x=0:0.01:1; y=2*x;
P=x;T=y;
net=newff([0 2], [20,1], {'tansig','purelin'},'traingd');
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

net1 = train(net, P, T);
```

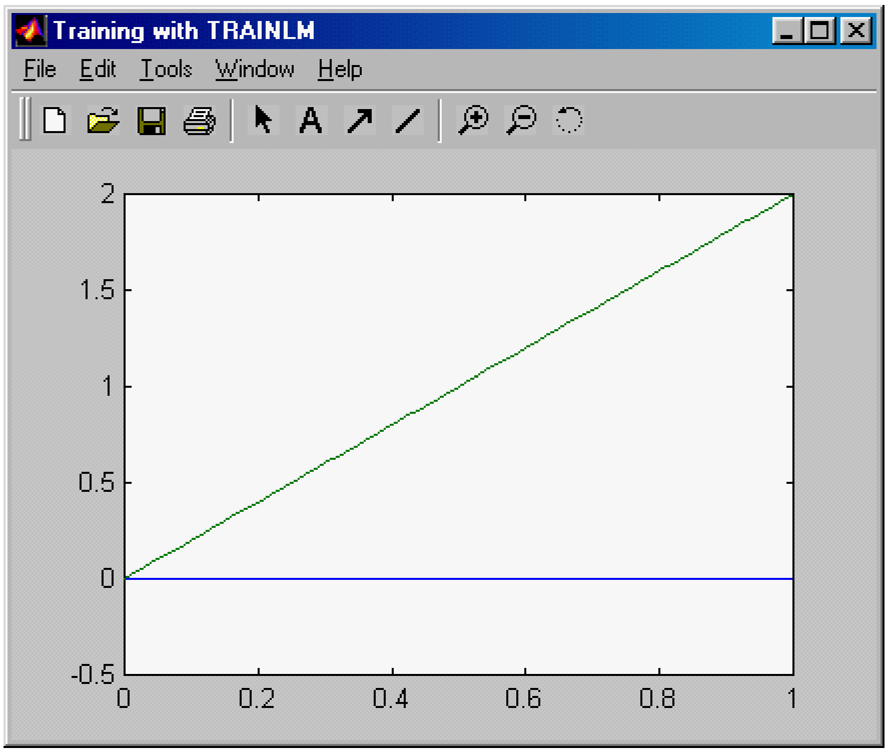


```
a= sim(net1,P);

%Plot the result and the error
plot(P,a-T,P,T)
```



This apparently contains fairly significant error.

Use the same network size, but apply Levenberg-Marquardt algorithm.



APPENDIX

NEWSOM Create a self-organizing map.

Syntax

net = newsom (PR,[d1,d2,...],tfcn,dfcn,olr,osteps,tlr,tns)

Description

Competitive layers are used to solve classification problems.

```
NET = NEWSOM (PR,[D1,D2,...],TFCN,DFCN,OLR,OSTEPS,TLR,TNS) takes,
  PR     - Rx2 matrix of min and max values for R input elements.
  Di     - Size of ith layer dimension, defaults = [5 8].
  TFCN   - Topology function, default = 'hextop'.
  DFCN   - Distance function, default = 'linkdist'.
  OLR    - Ordering phase learning rate, default = 0.9.
  OSTEPS - Ordering phase steps, default = 1000.
  TLR    - Tuning phase learning rate, default = 0.02;
  TND    - Tuning phase neighborhood distance, default = 1.
and returns a new self-organizing map.
```

The topology function TFCN can be HEXTOP, GRIDTOP, or RANDTOP.
The distance function can be LINKDIST, DIST, or MANDIST.

Examples

The input vectors defined below are distributed over
a 2-dimension input space varying over [0 2] and [0 1].
This data will be used to train a SOM with dimensions [3 5].

```
P = [rand(1,400)*2; rand(1,400)];
net = newsom([0 2; 0 1],[3 5]);
plotsom(net.layers{1}.positions)
```

Here the SOM is trained and the input vectors are plotted with
the map which the SOM's weights has formed.

```
net = train(net,P);
plot(P(1,:),P(2,:),'.g','markersize',20)
hold on
plotsom(net.iw{1,1},net.layers{1}.distances)
hold off
```

Properties

SOMs consist of a single layer with the NEGDIST weight function,
NETSUM net input function and the COMPET transfer function.

The layer has a weight from the input, but no bias.
The weight is initialized with MIDPOINT.

Adaptation and training are done with ADAPTWB and TRAINWB1,
which both update the weight with LEARNSOM.

See also SIM, INIT, ADAPT, TRAIN, ADAPTWB, TRAINWB1.

GENSIM Generate a SIMULINK block to simulate a neural network.

Syntax

gensim(net,st)

Description

GENSIM(NET,ST) takes these inputs,
  NET - Neural network.
  ST - Sample time (default = 1).
and creates a SIMULINK system containing a block which
simulates neural network NET with a sampling time of ST.

If NET has no input or layer delays (NET.numInputDelays
and NET.numLayerDelays are both 0) then you can use -1 for ST to
get a continuously sampling network.

Example

net = newff([0 1],[5 1]);
gensim(net)

NNSYSID

Use data from example 4 – discrete11.

```
[us,uscale]=dscale(u'); [youts,yscale]=dscale(yout');
```

SOMETHING STRANGE – no scaling if plotted! Maybe SYSID Toolbox.