Time and
Neural Networks

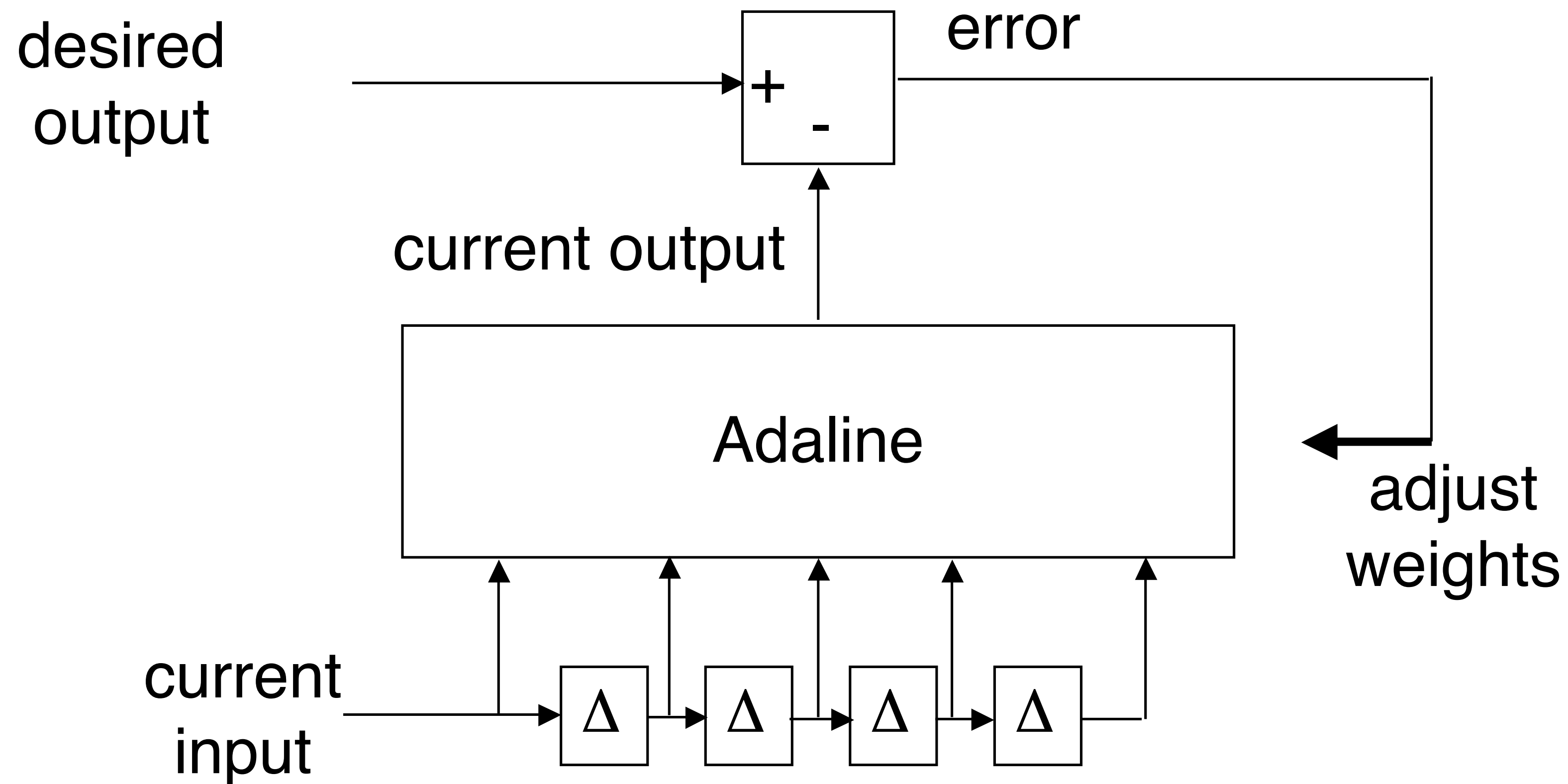
Thus far

- Networks have been “combinational”; input pattern presented at once
- Now we wish to consider cases where network inputs and learned behavior can include **functions of time**

Models to be Considered Here

- Mimicking certain I/O behavior (aka "system identification")
- Time-series prediction
- Adaptive (or Active) noise cancellation
- Time-Delay Neural Networks (TDNN, TLFF)
- Control: Backpropagation through time (BPTT)
- FIR-Multi-layer networks (FIRNET)

Adaline Learning to Mimic



Training the Adaline Mimic

- Recall the Adaline training rule:

$$\Delta \mathbf{W} = \eta \cdot (\text{desired - actual output}) \cdot \mathbf{input}$$

- Here input vector is the current input, along with all the delayed inputs (one per weight)

Use of Matlab

- Matlab has some pre-programmed functions that will be useful for succinct expositions.
- Similar setups apply to many varieties of network.

Matlab newlin

net = newlin(PR, S, ID, LR)

creates a linear layer

PR	R x 2 matrix of **min and max values** for R input elements.
S	Number of elements in the output vector (= #neurons)
ID	Input delay vector, default = [0] (no delay), the elements indicate the number of time-steps the input is delayed, e.g. [0, 1] or [1, 2, 3]
LR	Learning rate, default = 0.01.

Example

- Suppose we want a network to learn to phase-shift its input by a constant amount, say by 1 time-step.
- This can obviously be done with a single delay, but we don’t tell the network what the weights should be.
- We want the network to **learn** the weights to achieve the phase shift.

Sample Pattern and Target

- P = {0 1 2 1 2 3 5 1 4 2 5 3}
- T = {0 0 1 2 1 2 3 5 1 4 2 5}
- So T is P delayed by 1 time-step

newlin example

```
% net = newlin(PR, S, ID, LR) creates network

>> net = newlin([0 5], 1, [0 1], 0.01);
```

% This code creates a single input (range of [0 5])
% linear layer with **one neuron**,
% input delays of 0 and 1,
% and a learning rate of 0.01.

Adapting (learning with) the linear layer on-line learning

% specify **target values** for output sequence

% adapt the net through **one cycle**, each step of which
% adjusts the weights (Widrow-Hoff learning).

```
>> [net,Y, E, Pf] = adapt(net, P, T);        % Note that net is replaced by result

>> sim(net, P)

[0.1474]  [0.4637]  [1.3012]  [1.5059]  [1.3012]  [2.1386]  [3.2925]  [3.0691]  [1.9340]  [2.8644]  [2.7714]  [3.7

T =

  [0]  [0]  [1]  [2]  [1]  [2]  [3]  [5]  [1]  [4]  [2]  [5]
```

So far, not great

Adapting (learning with) the linear layer on-line learning

% After about 10 repetitions of the following:

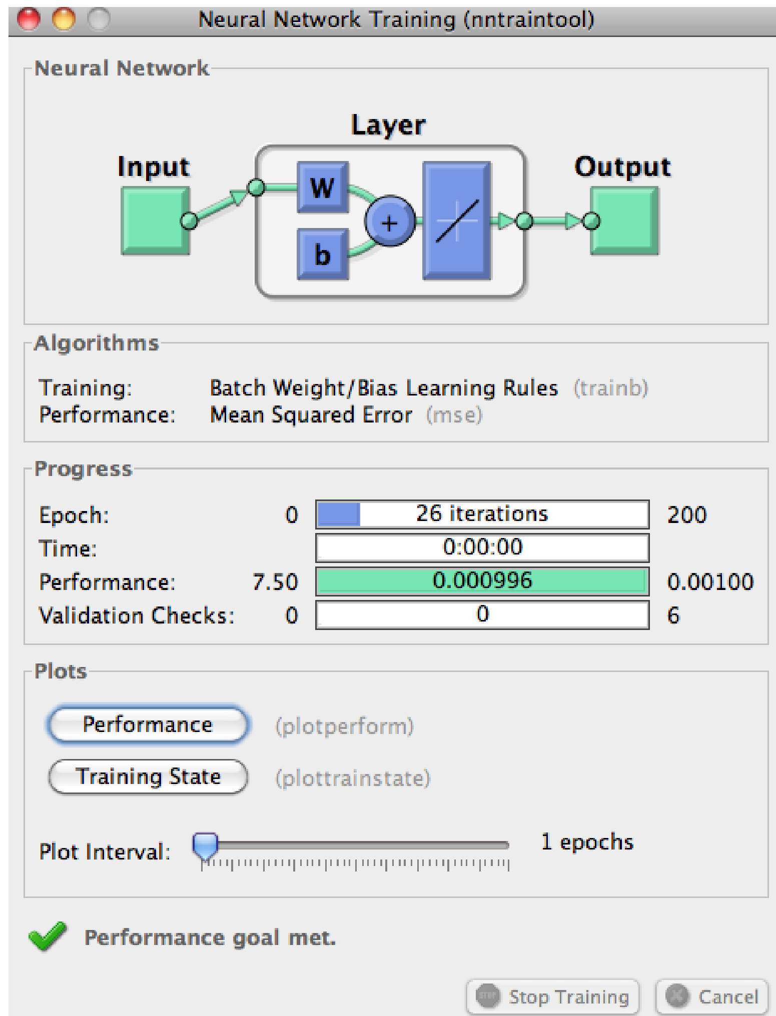
```
>> [net,Y, E, Pf] = adapt(net, P, T);          % Note that net is replaced by result

>> sim(net, P)

   [0.0974]    [0.0880]    [1.0591]    [2.0489]    [1.0591]    [2.0302]    [2.9919]    [4.9902]    [1.0404]    [4.0004]    [2.0115]    [4.9715]

T =

    [0]    [0]    [1]    [2]    [1]    [2]    [3]    [5]    [1]    [4]    [2]    [5]
```

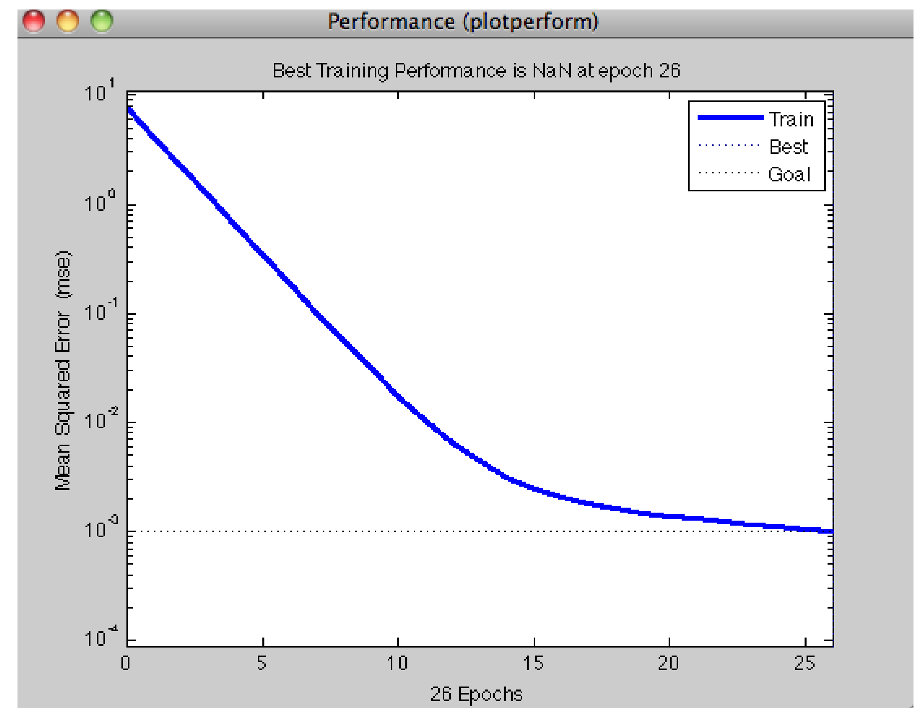
Much closer, wouldn't you say?

Training (vs. adapting) the linear layer
Runs the same input for multiple epochs

```
% re-initialize the net.

>> net = init(net);

>> sim(net, P)

ans =

   [0]   [0]   [0]   [0]   [0]   [0]   [0]   [0]   [0]   [0]   [0]   [0]


%Here we train the newly initialized layer on the concatenated sequences
% for 200 epochs to an error goal of 0.001.

>> net.trainParam.epochs = 200;
>> net.trainParam.goal = 0.001;

>> net = train(net,P,T);

>> sim(net, P)

 [0.0664]   [0.0531]   [1.0302]   [2.0340]   [1.0302]   [2.0074]   [2.9712]   [5.0053]   [1.0036]   [4.0016]   [1.9808]   [4.9787]

T =  [0]       [0]         [1]         [2]         [1]         [2]         [3]         [5]         [1]         [4]         [2]         [5]
```

Stopped due to goal being met





Behavior on *Unseen* Input

```
Pnew =

   [1]    [2]    [4]    [3]    [1]    [5]    [2]    [3]    [4]    [1]

>> sim(net, Pnew)

   [0.0531]    [1.0302]    [1.9941]    [3.9883]    [3.0244]    [0.9903]    [4.9920]    [2.0074]    [2.9845]    [4.0149]

% Pnew shifted [1]         [2]         [4]         [3]         [1]         [5]         [2]         [3]         [4]
```

newlin Training Details

Linear layers consist of a single layer with the **dotprod** weight function, **netsum** net input function, and **purelin** transfer function.

The layer has a weight from the input and a bias.
Weights and biases are initialized with **initzero**.

Adaptation and training are done with **adaptwb** and **trainwb**,
which both update weight and bias values with **learnwh**
(wh = the Widrow-Hoff rule = gradient descent).

Performance is measured with **mse**.

alternate newlin

net = newlin(PR, S, 0, P) % (i.e. **with Input delay = 0**)

takes an alternate argument,

PR R x 2 matrix of **min and max values** for R input elements.

S Number of elements in the output vector (= #neurons)

P Matrix of input vectors.

and returns a linear layer with the **maximum stable learning rate for learning with inputs P**.

Demo applin4

- An Adaline is trained to mimic a specific input-output behavior.
- The output is an attenuated version of the input.
- When subsequently presented with the input, the output is observed and the error computed.

Example: applin4 mimcry

% NEWLIN - Initializes a linear layer.
% ADAPT - Trains a linear layer with Widrow-Hoff rule.

% ADAPTIVE LINEAR SYSTEM IDENTIFICATION:

% Using the above functions a linear neuron is adaptively
% trained to model a linear system.

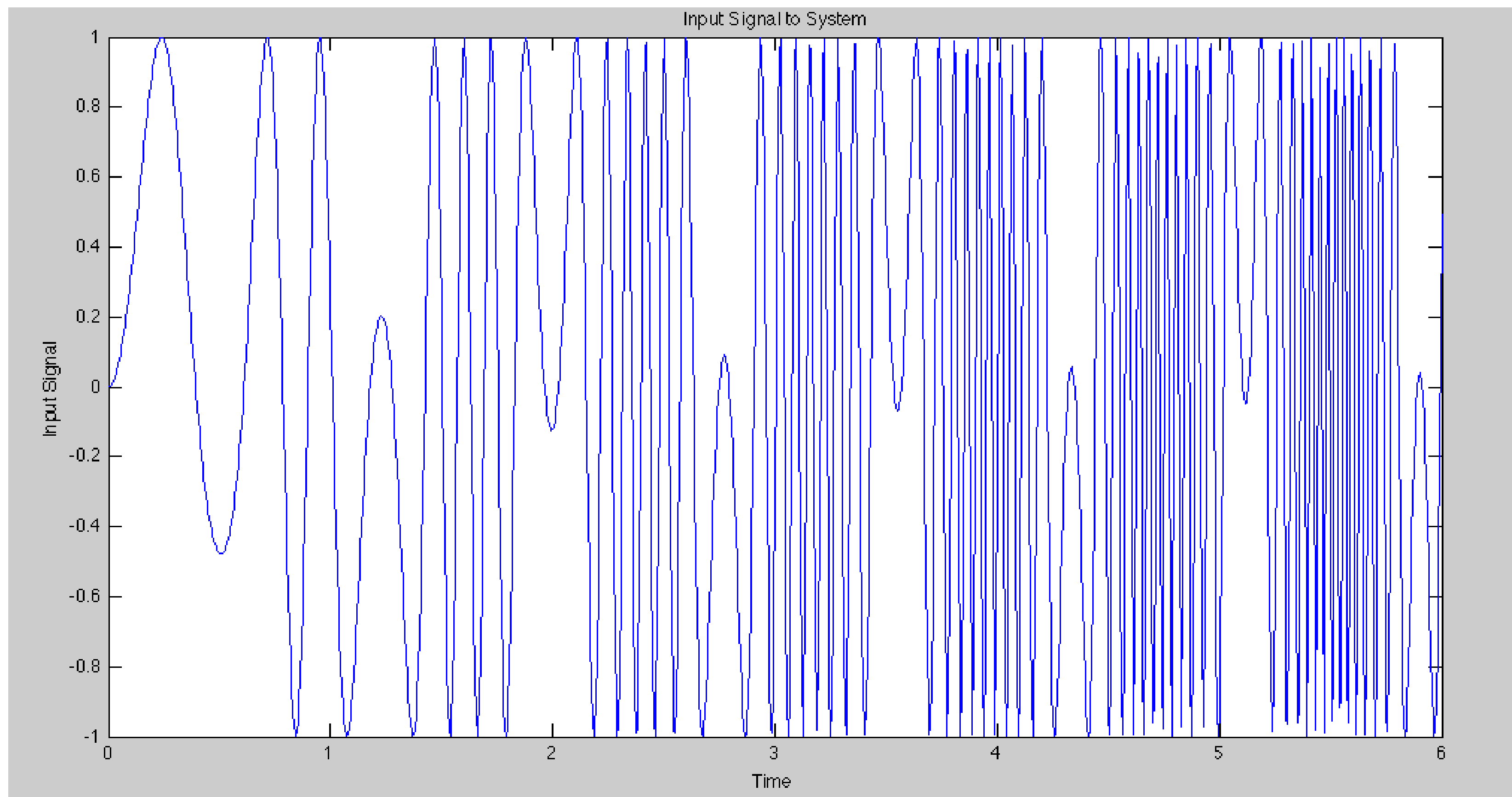
% **The linear neuron is able to adapt to changes in the**
% **model it is trying to mimic.**

applin4 input

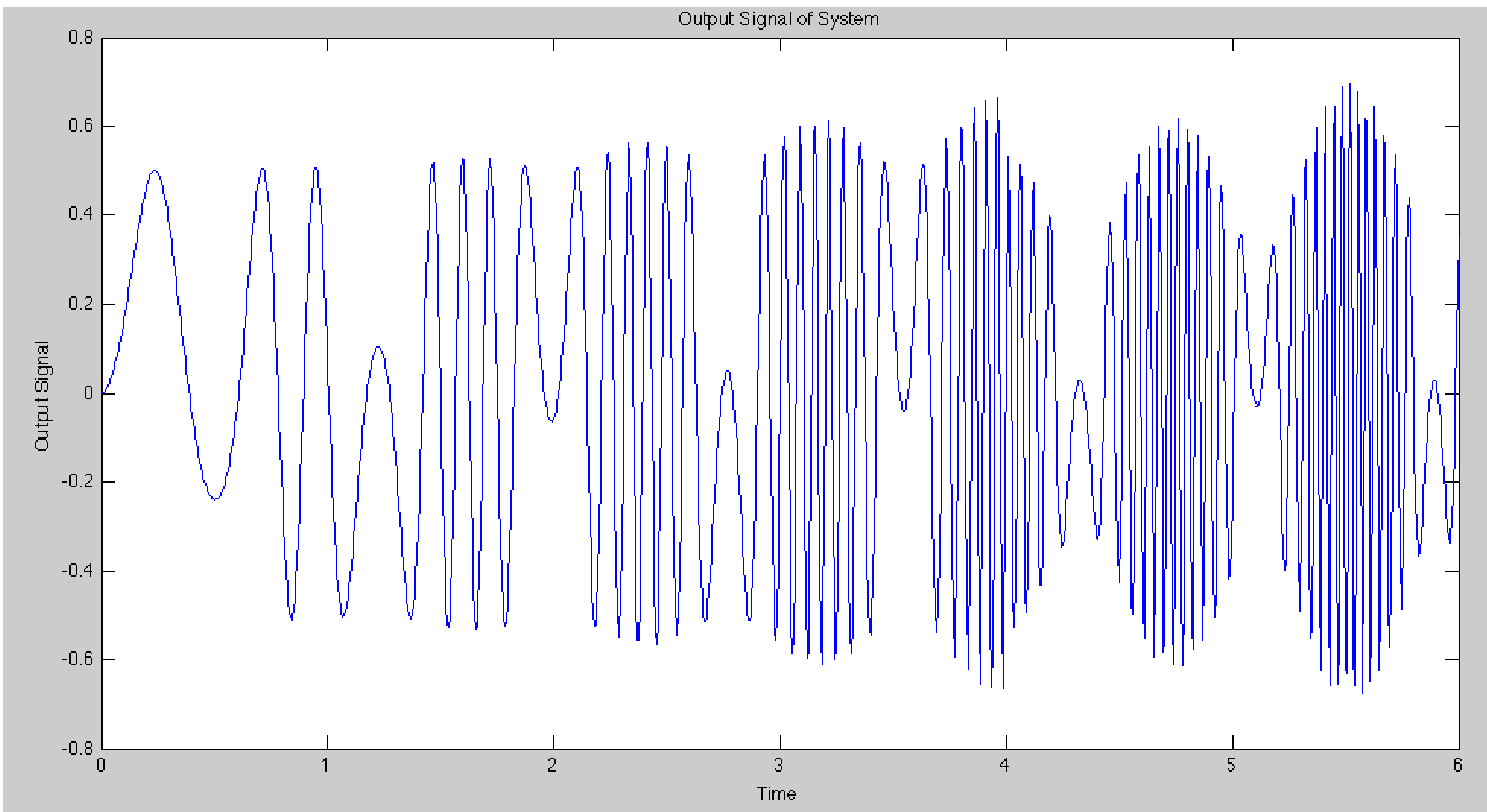
```
%    DEFINING A WAVE FORM
%    ====================
%    TIME1 and TIME2 define two segments of time.

time1 = 0:0.005:4;        % from 0 to 4 seconds
time2 = 4.005:0.005:6;    % from 4 to 6 seconds

%    TIME defines all the time steps of this simulation.

time = [time1 time2];     % from 0 to 6 seconds

%    X defines input signal to the system:

X = sin(sin(time*4).*time*8);
```

```
%    SPECIFYING SYSTEM OUTPUTS
%    ======================
%    Normally the outputs of the system would be measured.
%    Here we will generate some values.
%    T1 defines the outputs during seconds 0 through 4.

steps1 = length(time1);
[T1, state] = filter([1 -0.5],1,X(1:steps1));

%    At 4 seconds the system changes slightly.
%    T2 defines the outputs during seconds 4 through 6.

steps2 = length(time2);
T2 = filter([0.9 -0.6], 1, X((1:steps2) + steps1), state);

%    T defines the outputs during the entire interval.
T = [T1 T2];
```

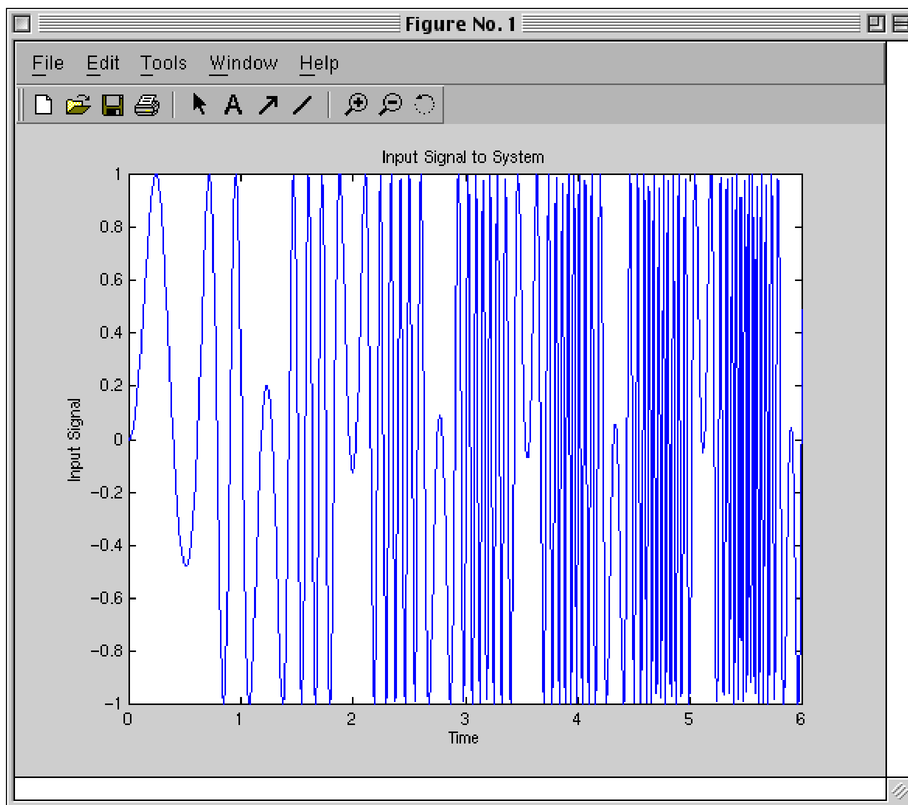
input



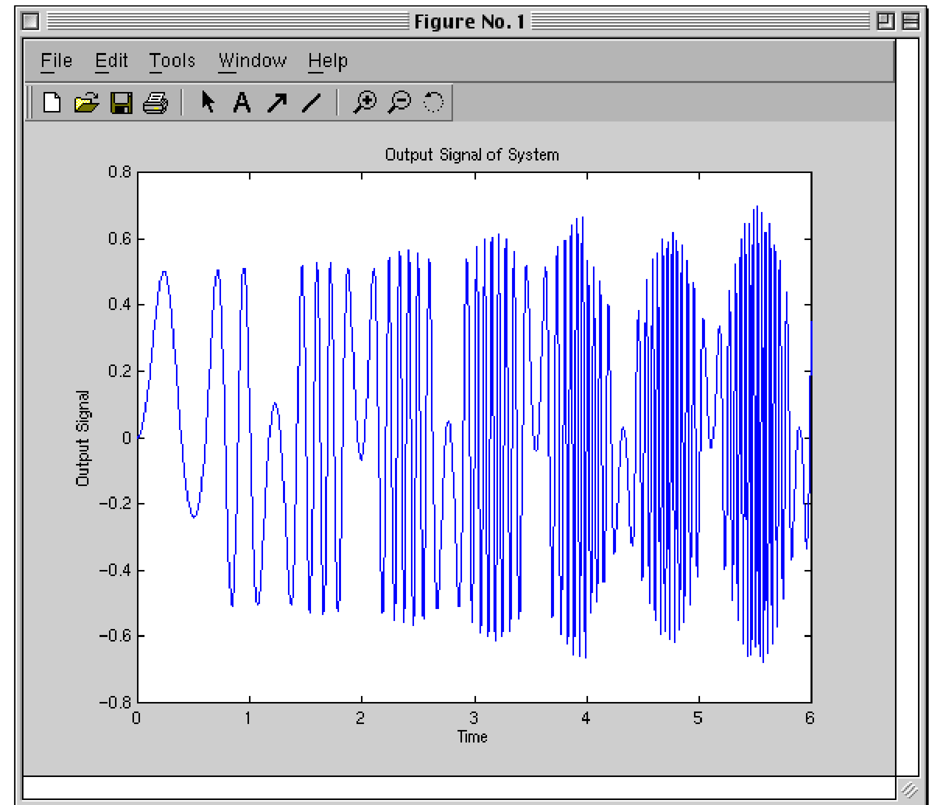
output



applin4: Input-Output Relation



Input



Desired Output

applin4 network definition

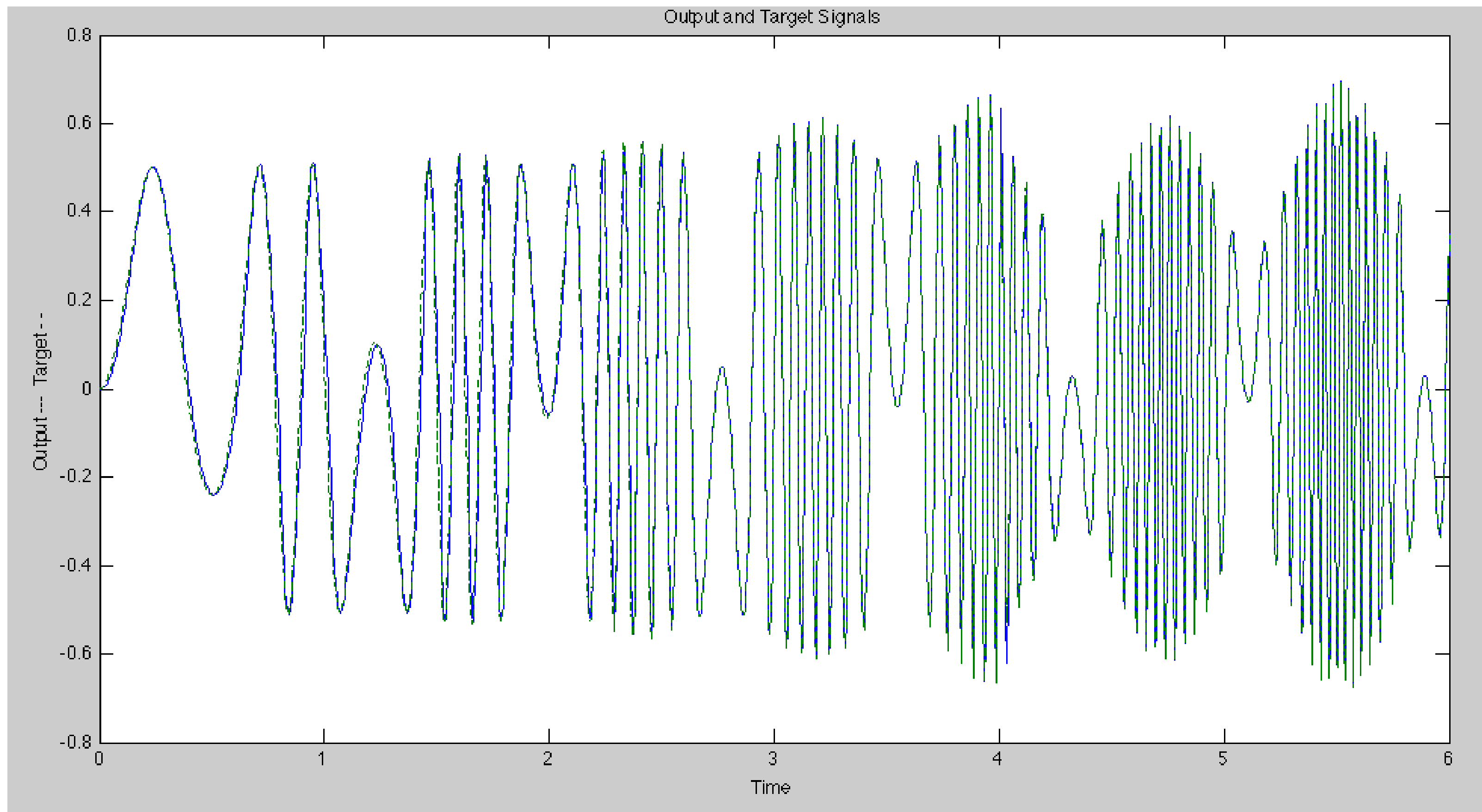
```
%   DEFINE THE NETWORK
%   ====================
%   NEWLIN generates a linear network.
%   We will use a learning rate of 0.5, and two
%   delays in the input.

lr = 0.5;
delays = [0 1];

net = newlin(minmax(cat(2,P{:})), 1, delays, lr);
```

applin4 adapting

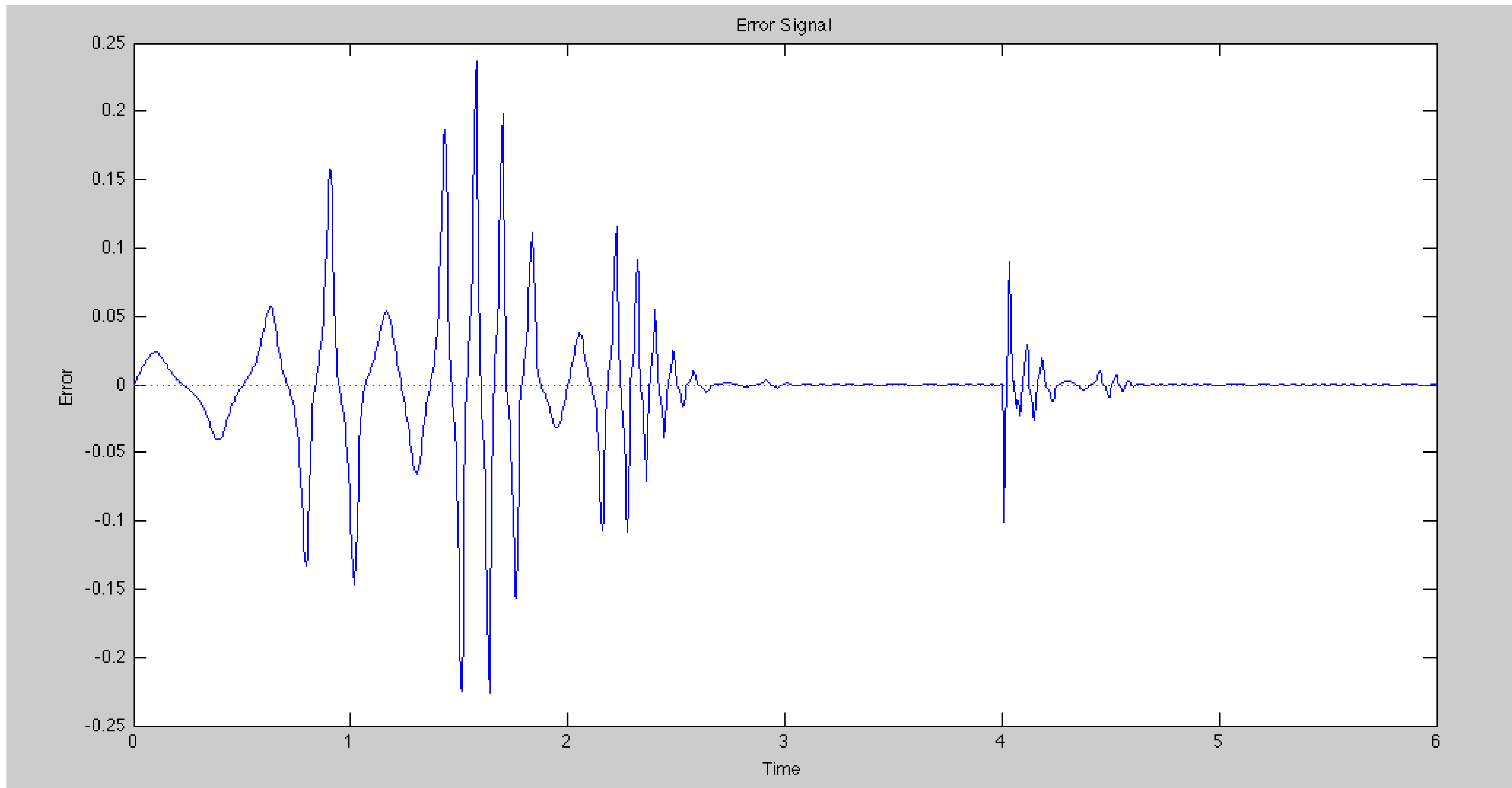
```
%    ADAPTING THE LINEAR NEURON
%    ==========================
%    ADAPT simulates adaptive linear neurons.  It takes the
%    initial network, an input signal, and a target signal,
%    and filters the signal adaptively.  The output signal and
%    the error signal are returned, along with new network.

[net,y,e]=adapt(net,P,T);
```

% Note: The input is only seen once!
% Adaptation takes place as you go.

applin4: actual output (blue) vs. target (green)

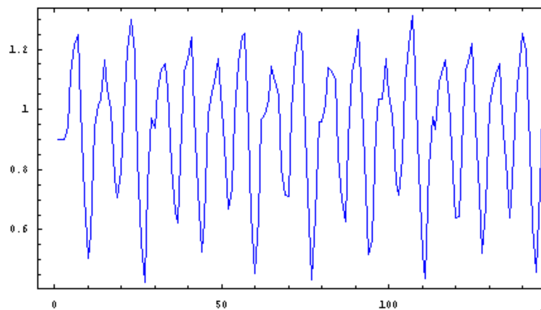


applin4: error
(note scale change)



Example: Time-Series Problems: "Predict the Future"

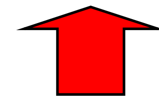
sampled data points



?

Trained Network

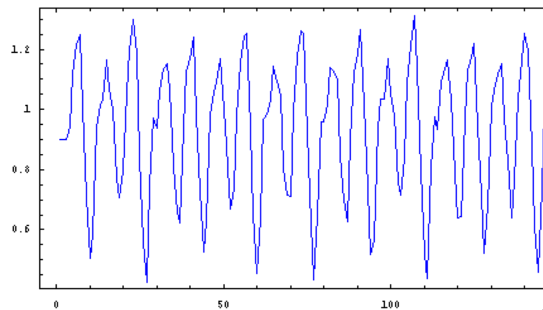
Answer (approx.)



What will the next sample input in the series be?

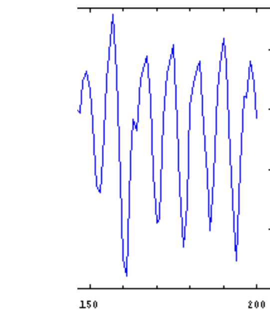
Time-Series Problems: "Predict the Future"

sampled data points



?

Trained Network



Better yet:
What will the **next *n*** sample inputs be, for nominal n?

Possible Applications

- Signal processing, filtering
- Sun-spot prediction
- Predict the degradation of the ozone layer
- Market analysis

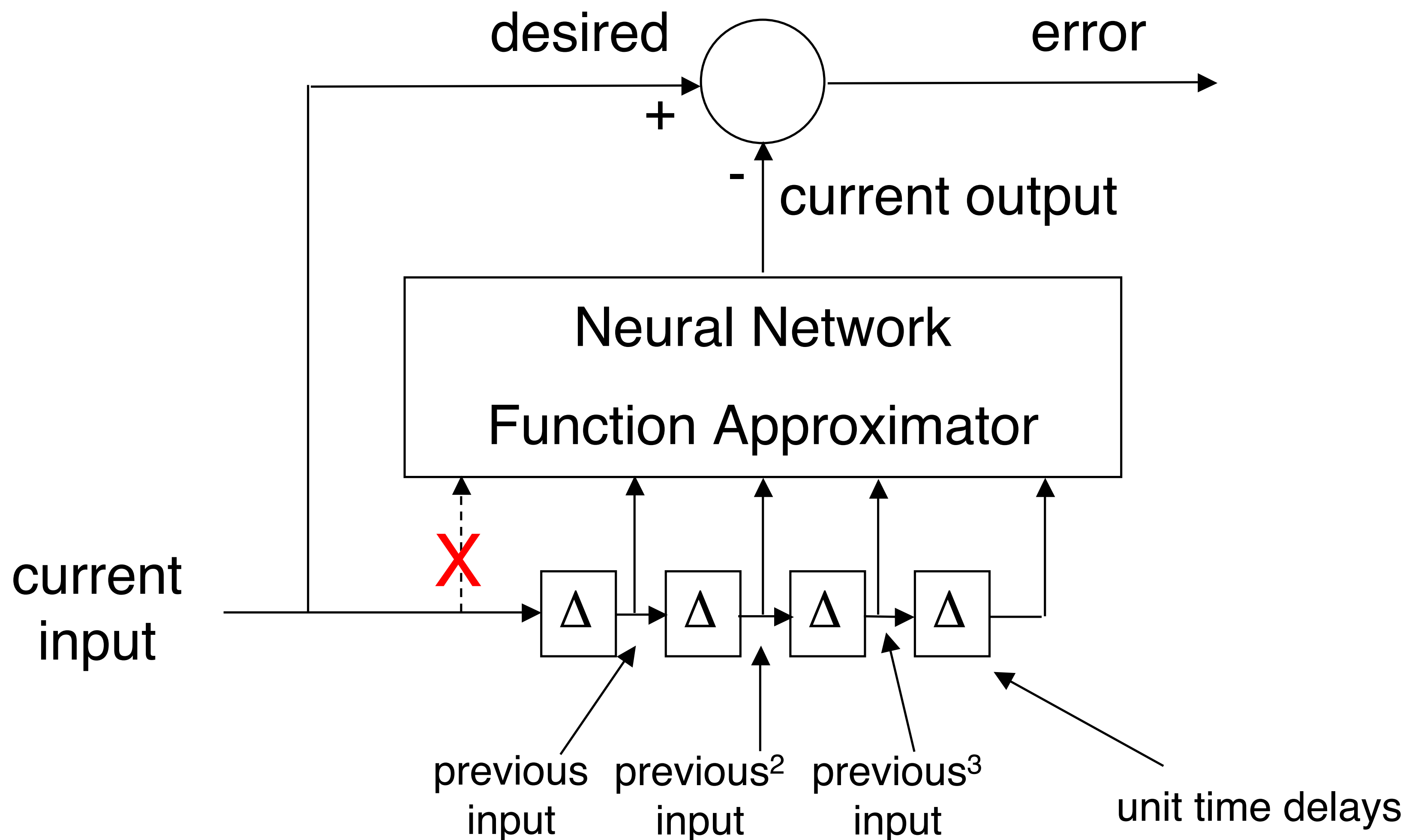
Learning to Predict

- A network can learn to **predict the next input** based on recent history of inputs, assuming some sort of regularity in the sequence.
- The previous n input values are held in a “delay line”.
- The current input value is withheld for training.
- The current error is

  desired - current network output

  where desired = current input

Predicting Using Error Value



Matlab newlind

- For a fixed input/output, there is more than one way to obtain the weights for an adaline.
  - Adaptation is one that we have seen earlier.
  - Another way is to **solve** for the weights so as to minimize MSE. For example, the pseudo-inverse method can be used for solving.
  - **newlind** is such a solver (The “d” stands for “design”.)
  - A third method, not yet studied, is RLS (Recursive Least Squares).

Matlab newlind

Syntax

```
net = newlind(P,T,Pi)
```

Description

`newlind(P,T,Pi)` takes these input arguments,

`P`	`R` x `Q` matrix of `Q` input vectors
`T`	`S` x `Q` matrix of `Q` target class vectors
`Pi`	`1` x `ID` cell array of initial input delay states

where each element `Pi{i,k}` is an `Ri` x `Q` matrix, and the default = `[]`,

and returns a linear layer designed to output `T` (with minimum sum square error) given input `P`.

Algorithm

`newlind` calculates weight `W` and bias `B` values for a linear layer from inputs `P` and targets `T` by solving this linear equation in the least squares sense:

```
[W b] * [P; ones] = T
```

newlind examples

You want a linear layer that outputs `T` given `P` for the following definitions:

```
P = [1 2 3];
T = [2.0 4.1 5.9];
```

Use `newlind` to design such a network and check its response.

```
net = newlind(P,T);
Y = sim(net,P)
```

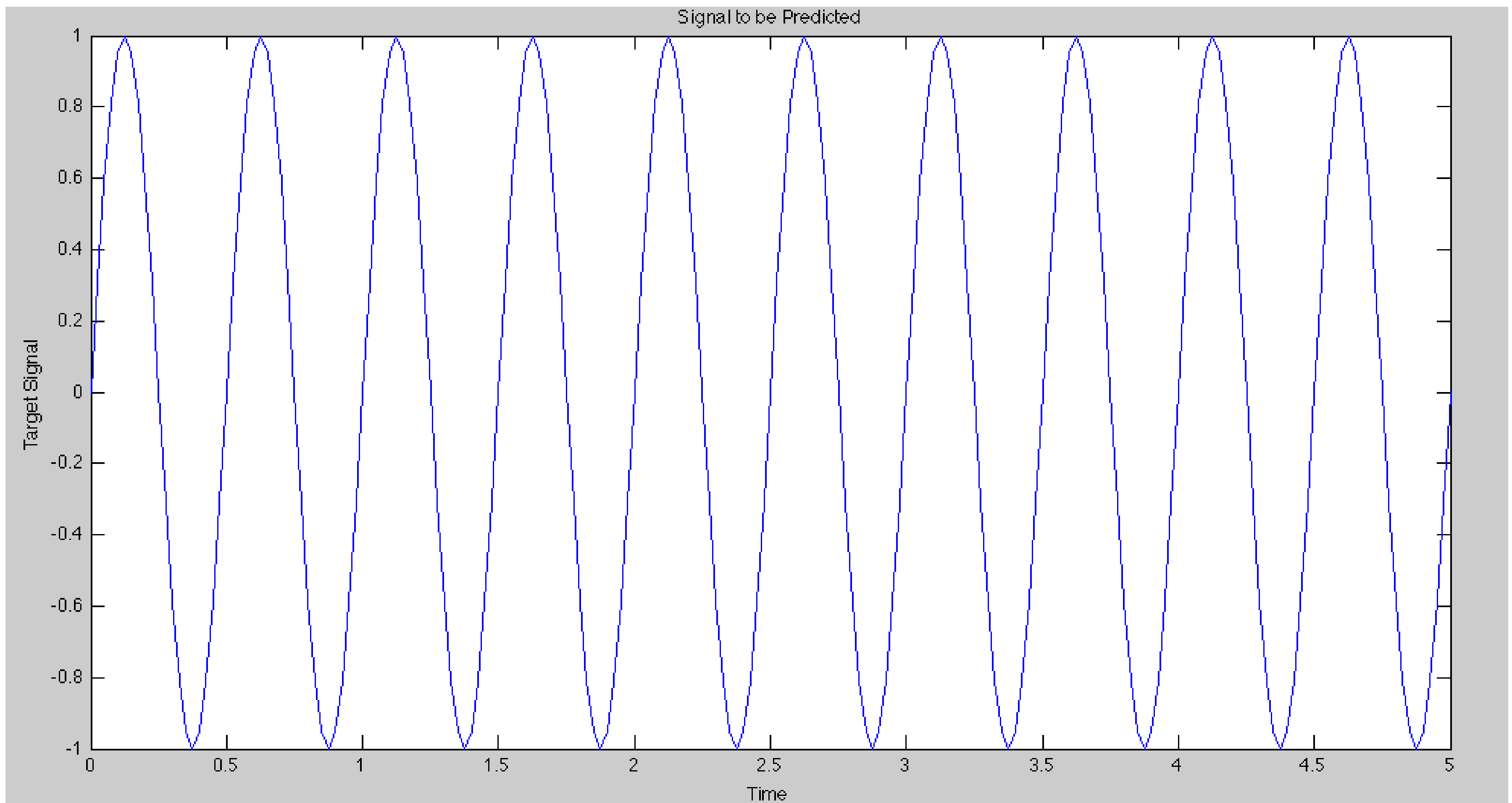
You want another linear layer that outputs the sequence `T` given the sequence `P` and two initial input delay states `Pi`.

```
P = {1 2 1 3 3 2};
Pi = {1 3};
T = {5.0 6.1 4.0 6.0 6.9 8.0};
net = newlind(P,T,Pi);
Y = sim(net,P,Pi)
```

applin1 uses **newlind** for prediction

```
%   DEFINING A WAVE FORM
%   ====================
%   TIME defines the time steps of this simulation.

time = 0:0.025:5;        % from 0 to 6 seconds

%   T defines the signal in time to be predicted:

T = sin(time*4*pi);
Q = length(T);

%   The input P to the network is the last five values of the signal T:

P = zeros(5,Q);
P(1,2:Q) = T(1,1:(Q-1));
P(2,3:Q) = T(1,1:(Q-2));
P(3,4:Q) = T(1,1:(Q-3));
P(4,5:Q) = T(1,1:(Q-4));
P(5,6:Q) = T(1,1:(Q-5));
```

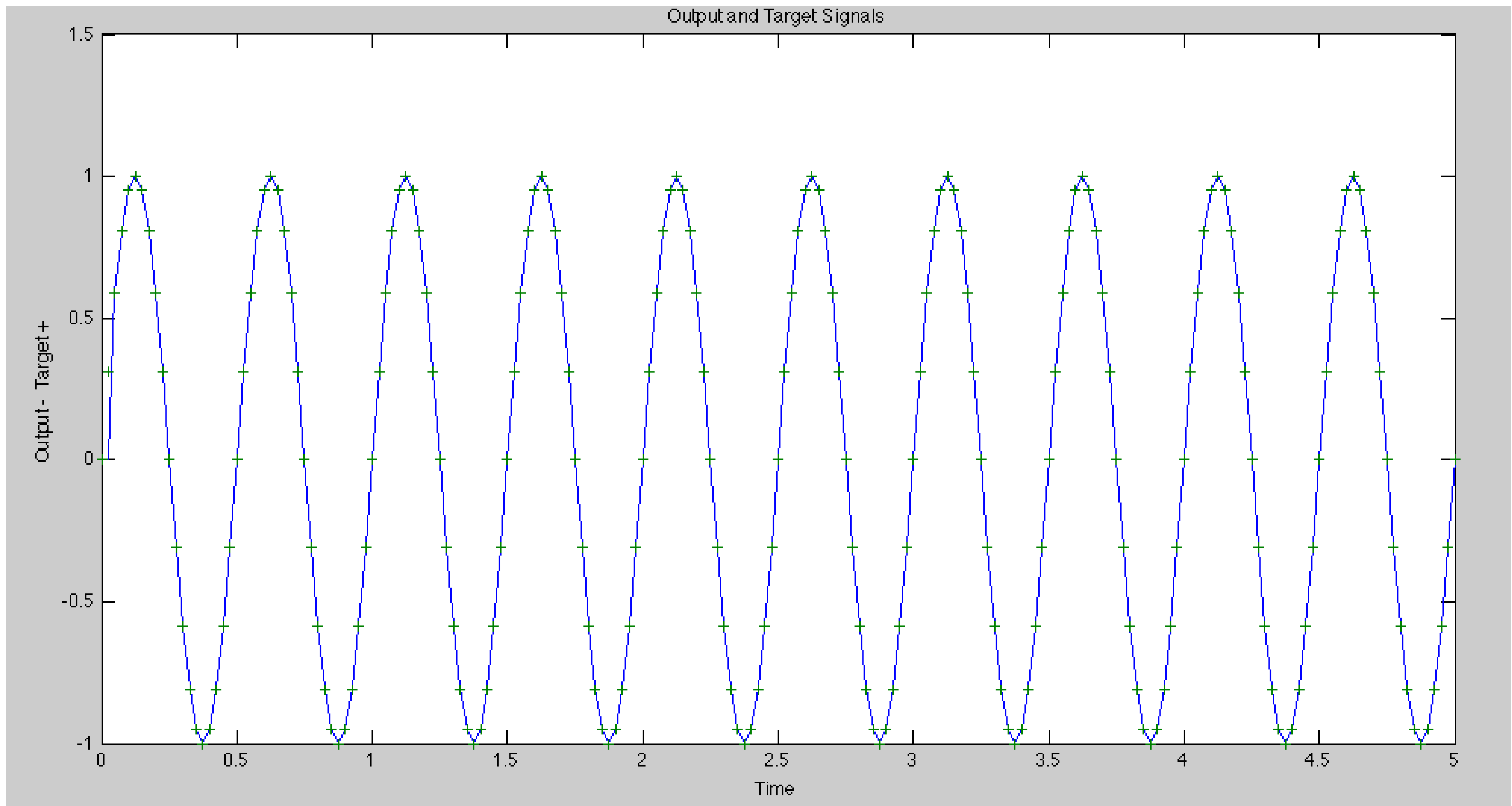
applin1 target signal to be predicted



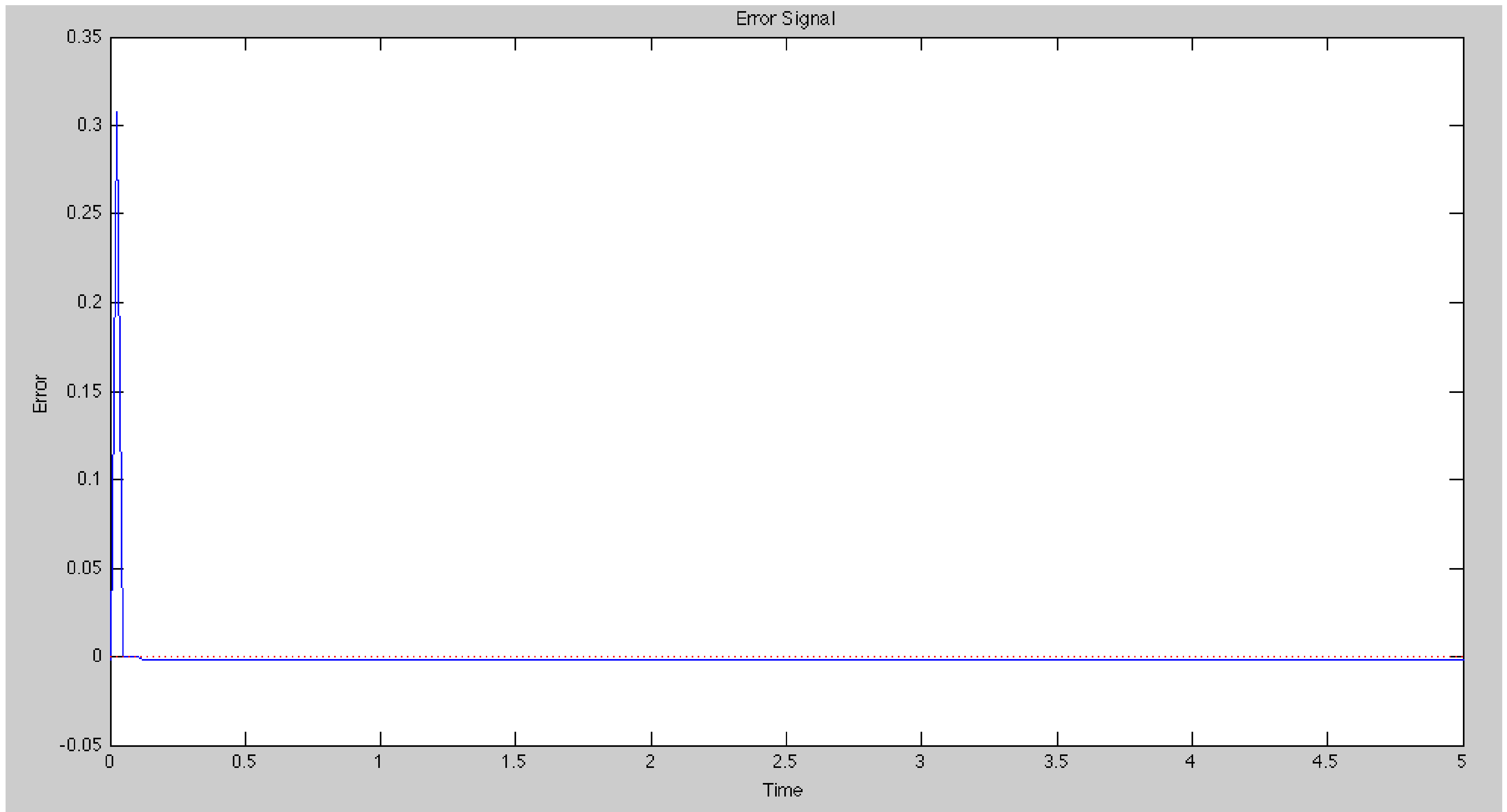
applin1 network design and test

```
%    NEWLIND solves for weights and biases which will let
%    the linear neuron model the system.

net = newlind(P,T);

%    TESTING THE PREDICTOR
%    ======================
%    SIM simulates the linear neuron which attempts
%    to predict the next value in the signal at each
%    timestep.

a = sim(net,P);

%    Error is the difference between output and target signals.

e = T - a;
```

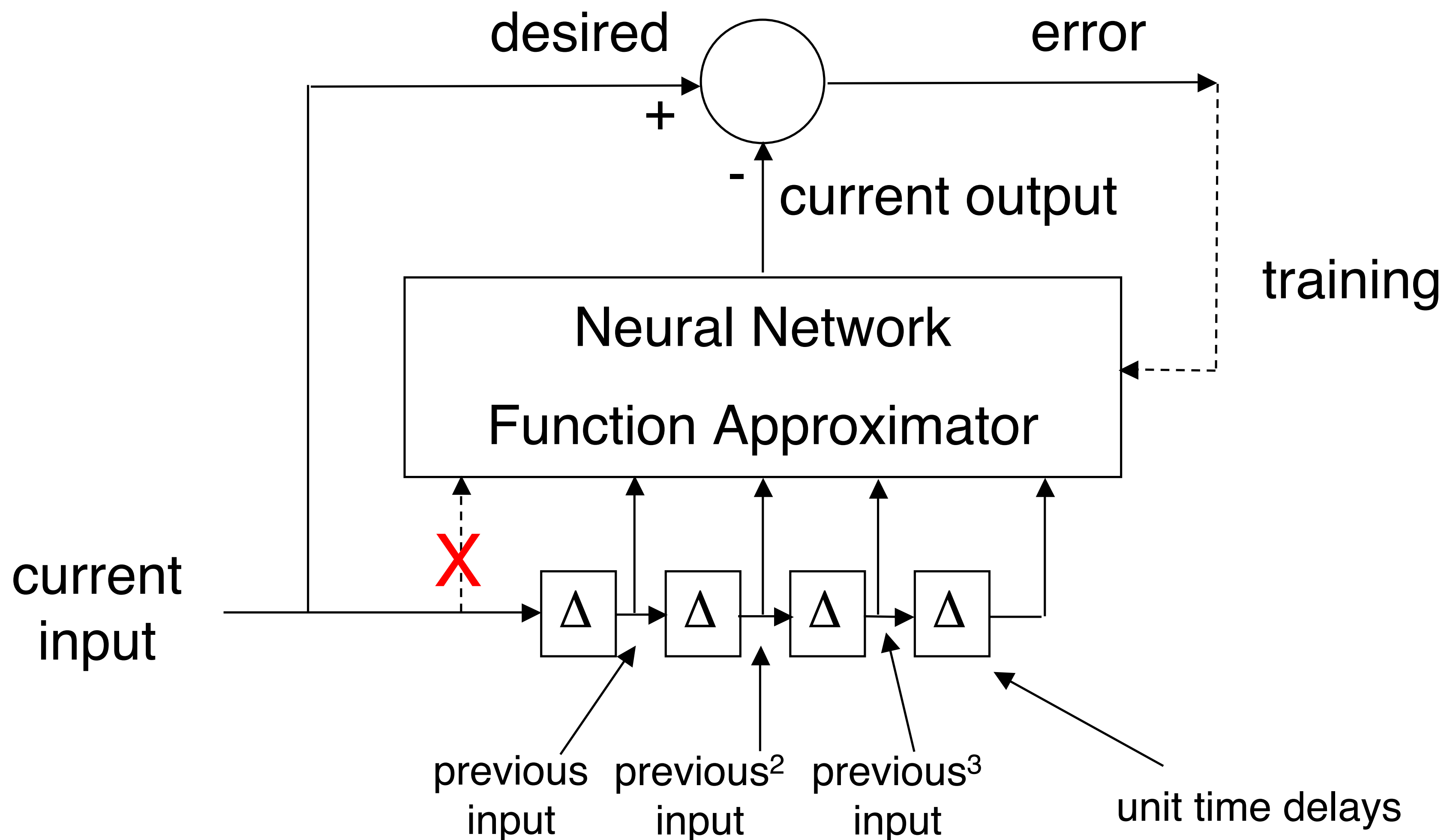
applin1 network output (-) vs. target (+)



applin1 error = target - predicted



Learning to Predict



Interesting Point

- The Adaline Predictor can be trained on-line **during operation**.
- At each time step, one set of weight modifications can be made.
- After a transient, the network learns to produce the desired behavior.

Demonstration applin2

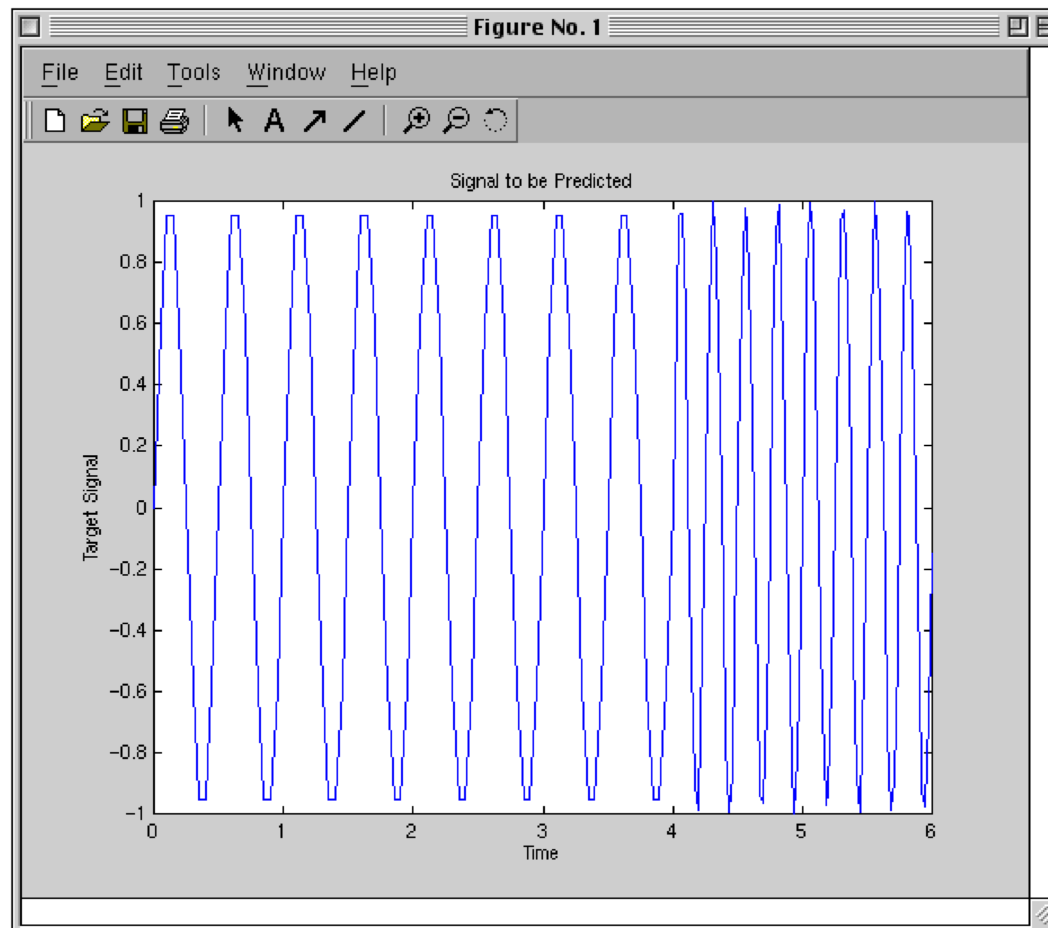
- Predicts the next input based on 5 previous input samples.

- The input is a sine wave, but the frequency doubles after awhile.

- It is desirable for the network to adapt its behavior to the new frequency.

applin2

signal to be predicted (2 sine waves of different frequencies)



applin2

% NEWLIN - Creates and initializes a linear layer.
% ADAPT - Trains a linear layer with Widrow-Hoff rule.

% ADAPTIVE LINEAR PREDICTION:

% Using the above functions a linear neuron is adaptively
% trained to predict the next value in a signal, given the
% last five values of the signal.

% The linear neuron is able to adapt to changes in the
% signal it is trying to predict.

applin2

```
%    DEFINING A WAVE FORM
%    TIME1 and TIME2 define two segments of time.

time1 = 0:0.05:4;       % from 0 to 4 seconds, steps of .05
time2 = 4.05:0.024:6;  % from 4 to 6 seconds, steps of .05

%    TIME defines all the time steps of this simulation.

time = [time1 time2];  % from 0 to 6 seconds

%    T defines a signal which changes frequency once:

T = con2seq([sin(time1*4*pi) sin(time2*8*pi)]);
```

% The input P to the network is the same as the target.

% **The network will use the last five**
% values of the target to predict the next value.

applin2

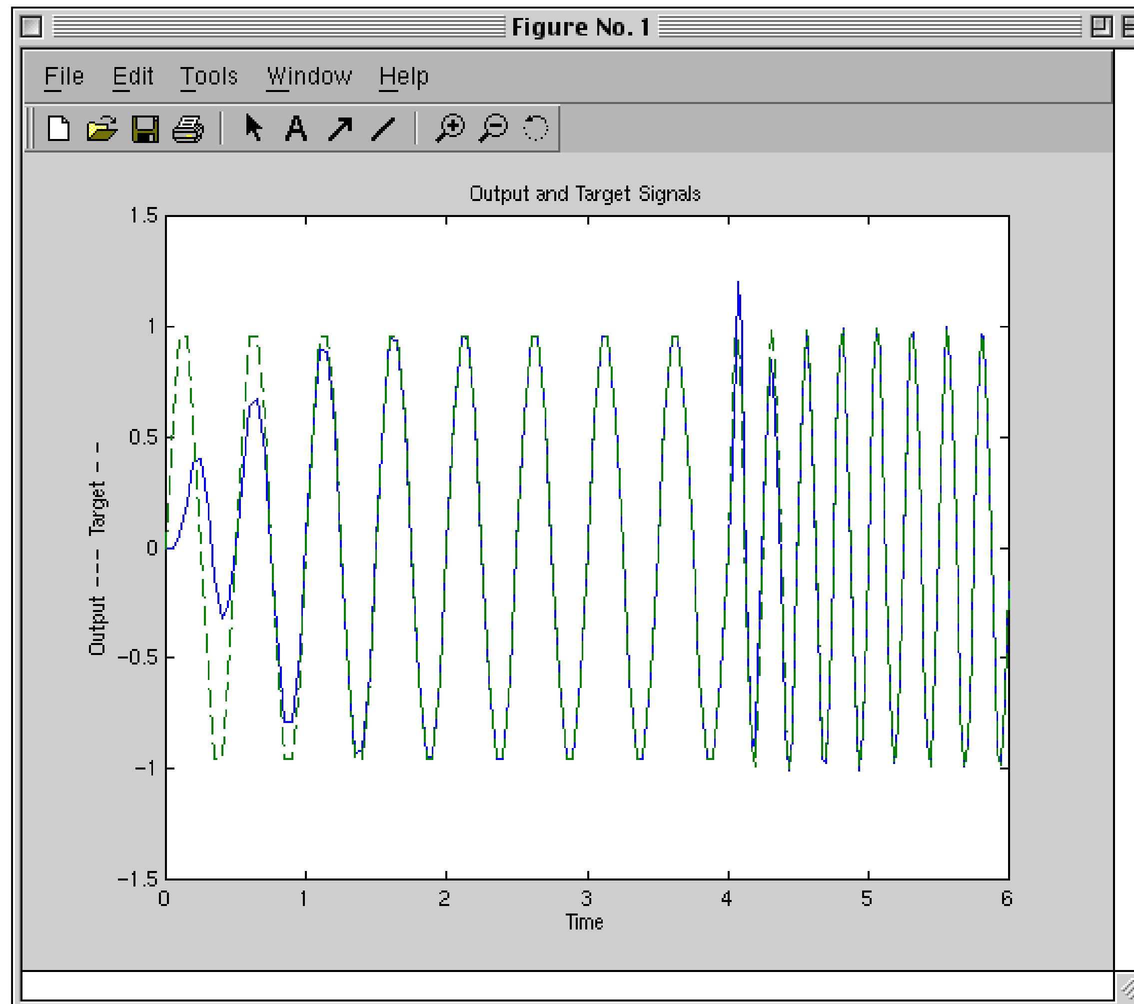
```
%   NEWLIN generates a linear network.

%   We will use a learning rate of 0.1, and five
%   delays in the input.  The resulting network
%   will predict the next value of the target signal
%   using the last five values of the target.

lr = 0.1;
delays = [1 2 3 4 5];
```

net = **newlin**(minmax(cat(2,P{:})),1,delays,lr);

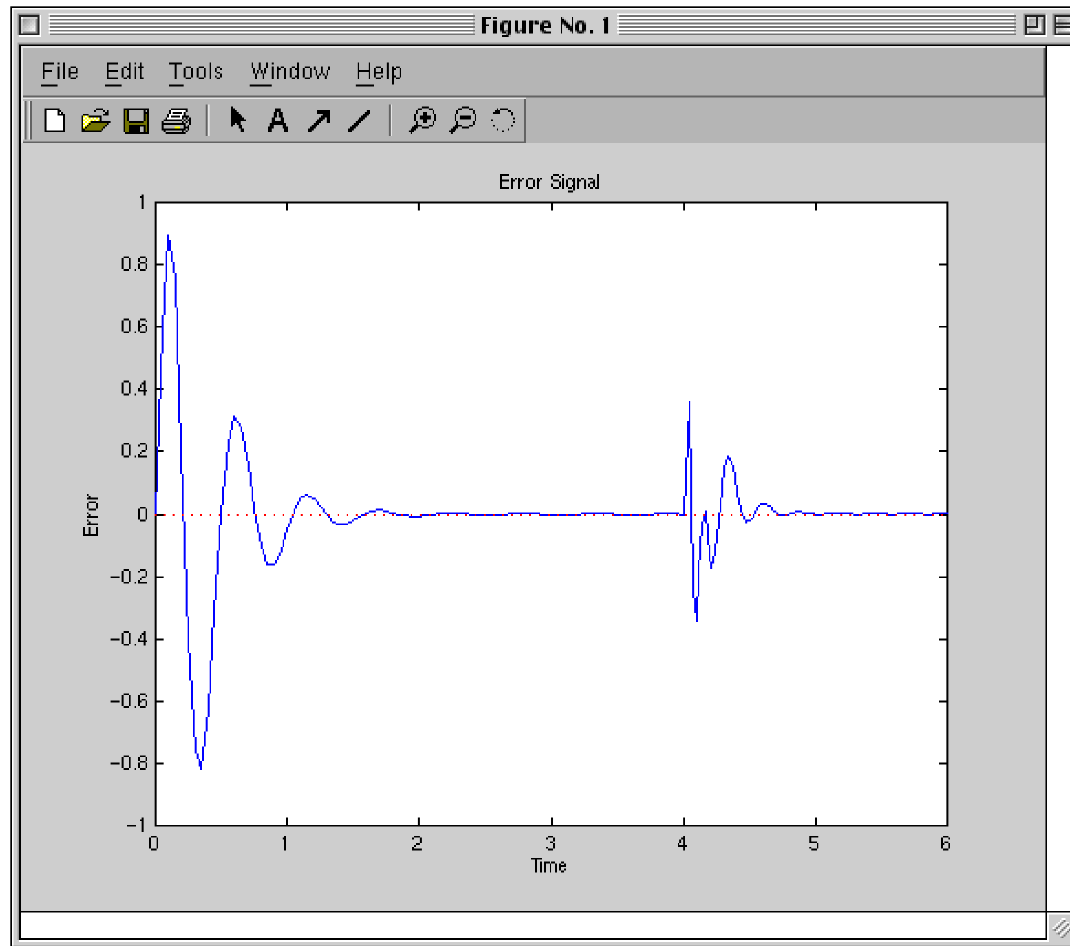
applin2

```
%    ADAPTING THE LINEAR NEURON
%    ===========================

%    ADAPT simulates adaptive linear neurons.  It takes the initial
%    network, an input signal, and a target signal,
%    and filters the signal adaptively.  The output signal and
%    the error signal are returned, along with new network.

[net, y, e]=adapt(net, P, T);
```

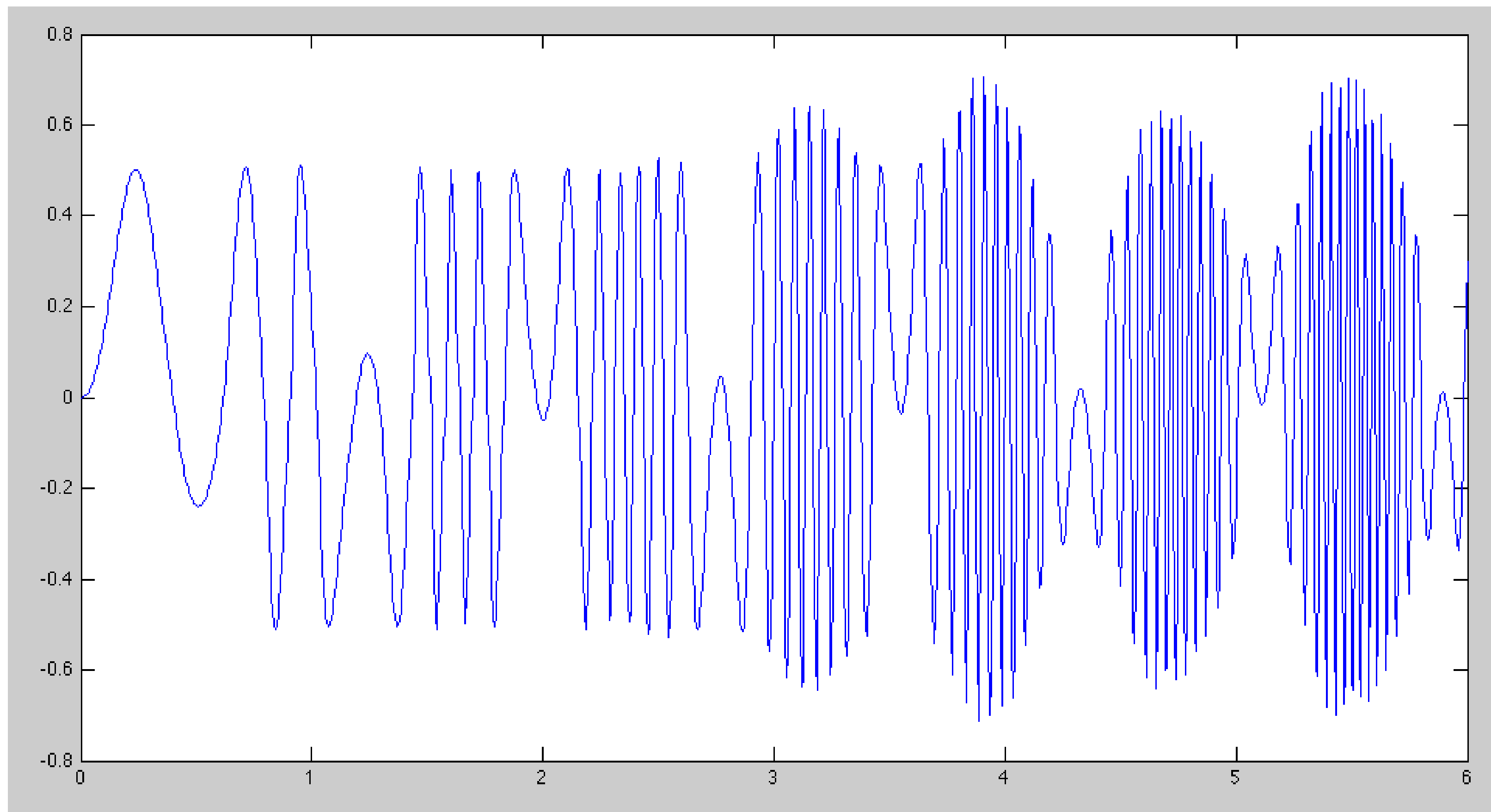
applin2: actual output vs. target



applin2: error = target - output

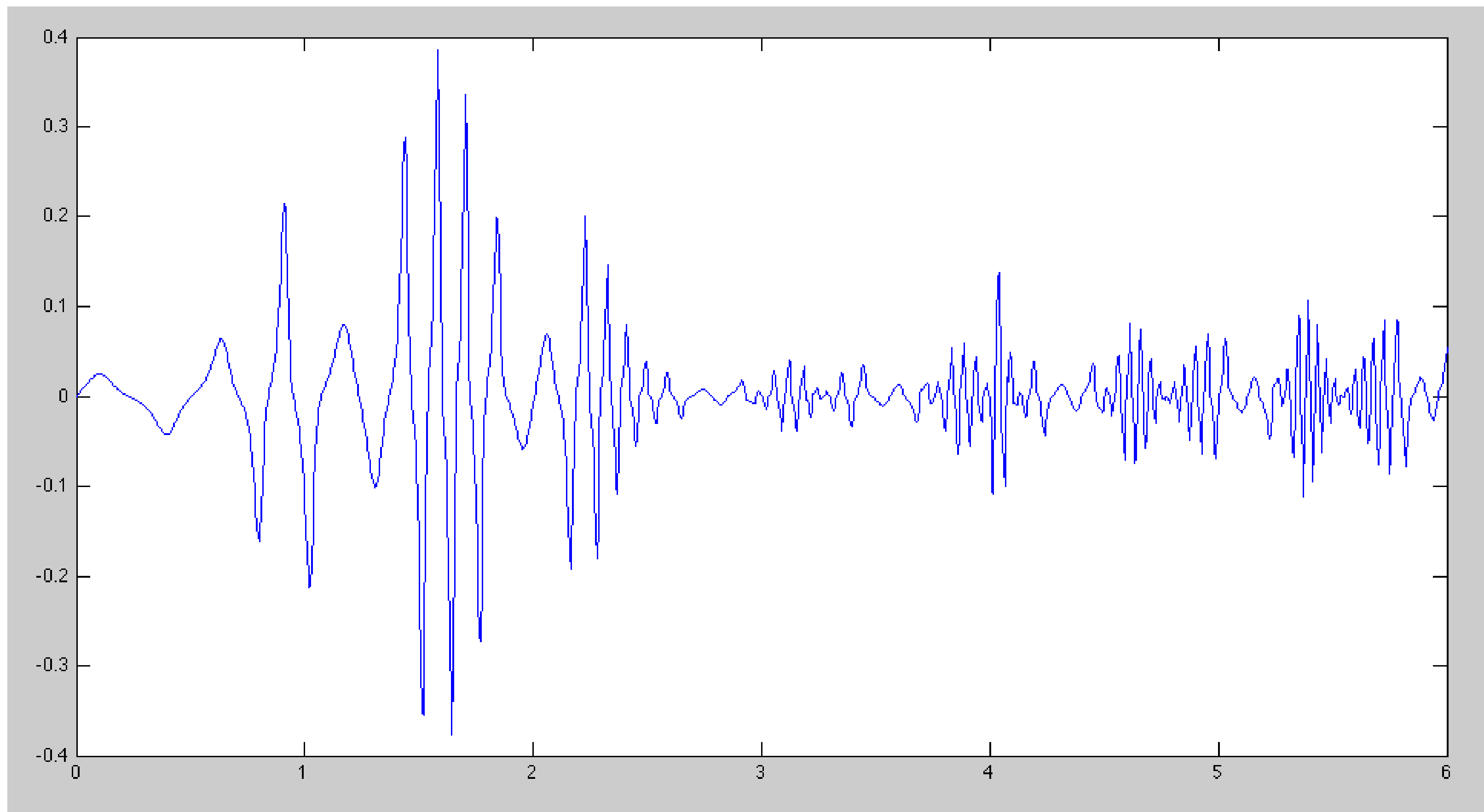


modified applin4 to predict:
using delays = [1 2] instead of [0 1]



modified applin4 to predict:

using delays = [1 2] instead of [0 1]



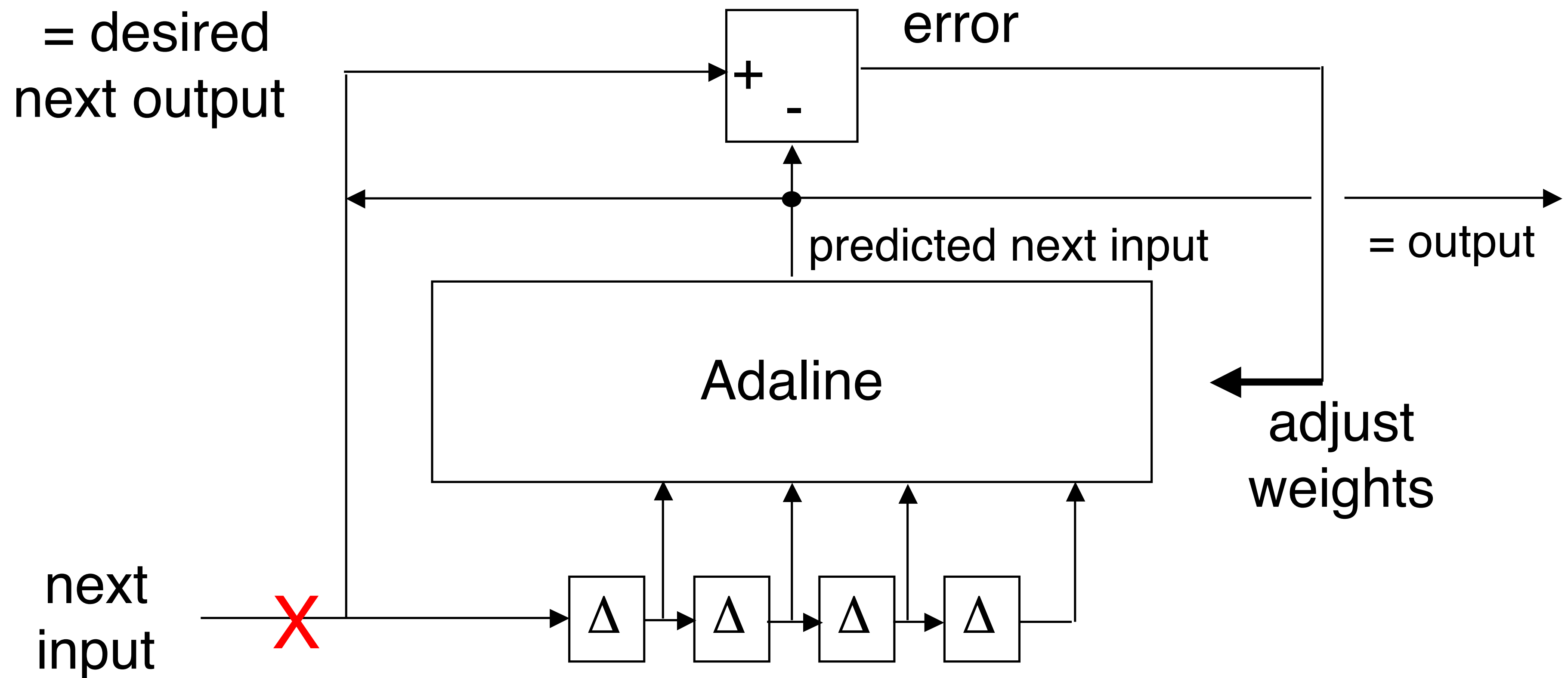
Once the Network has Been Trained

- it can use its *own* output as the next input.

- That is, it can "run free", predicting the full output **sequence.**

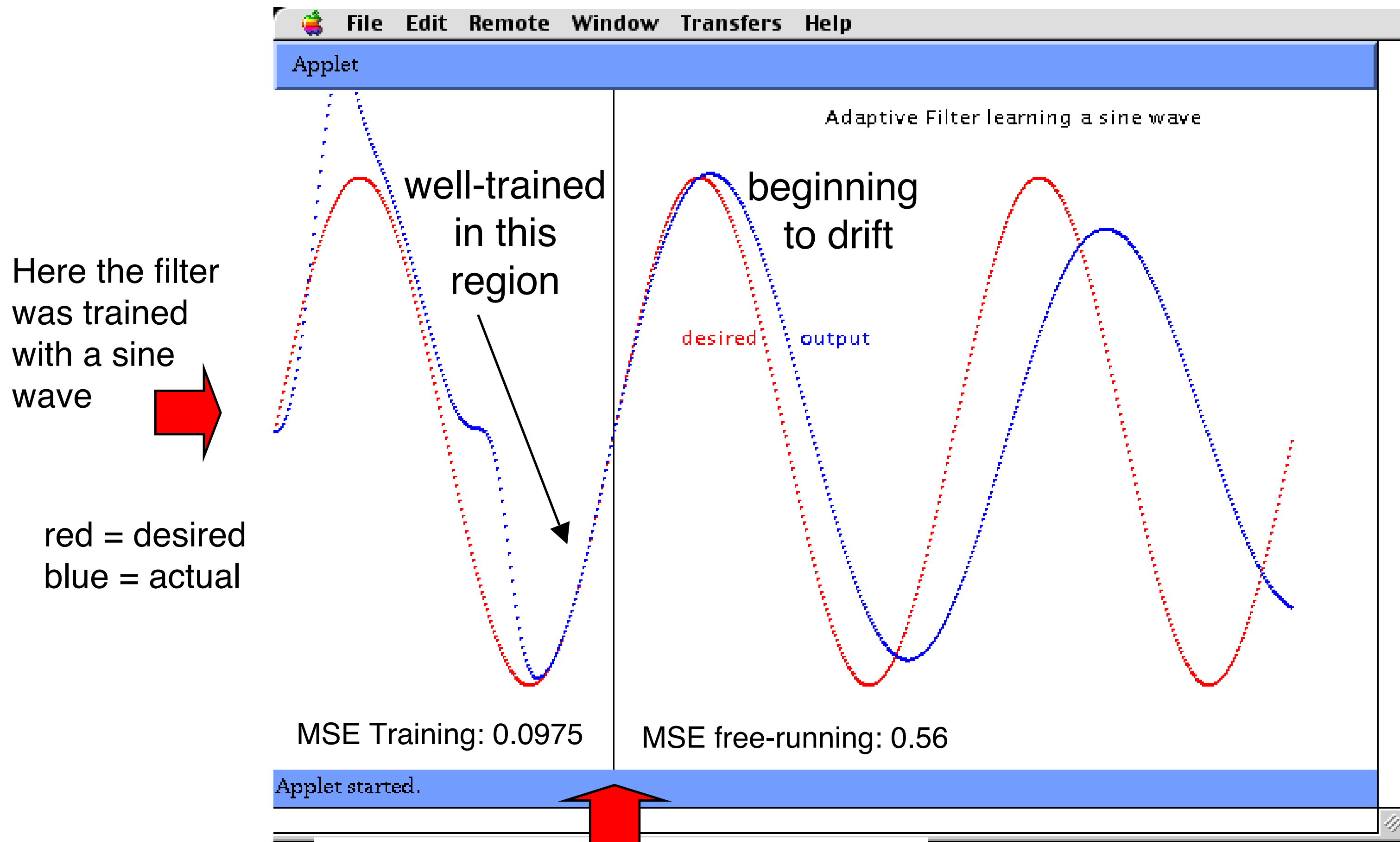
- Since the output was only an approximation, the accuracy of the predicted output will *deteriorate* with time.

Free-Running Mode



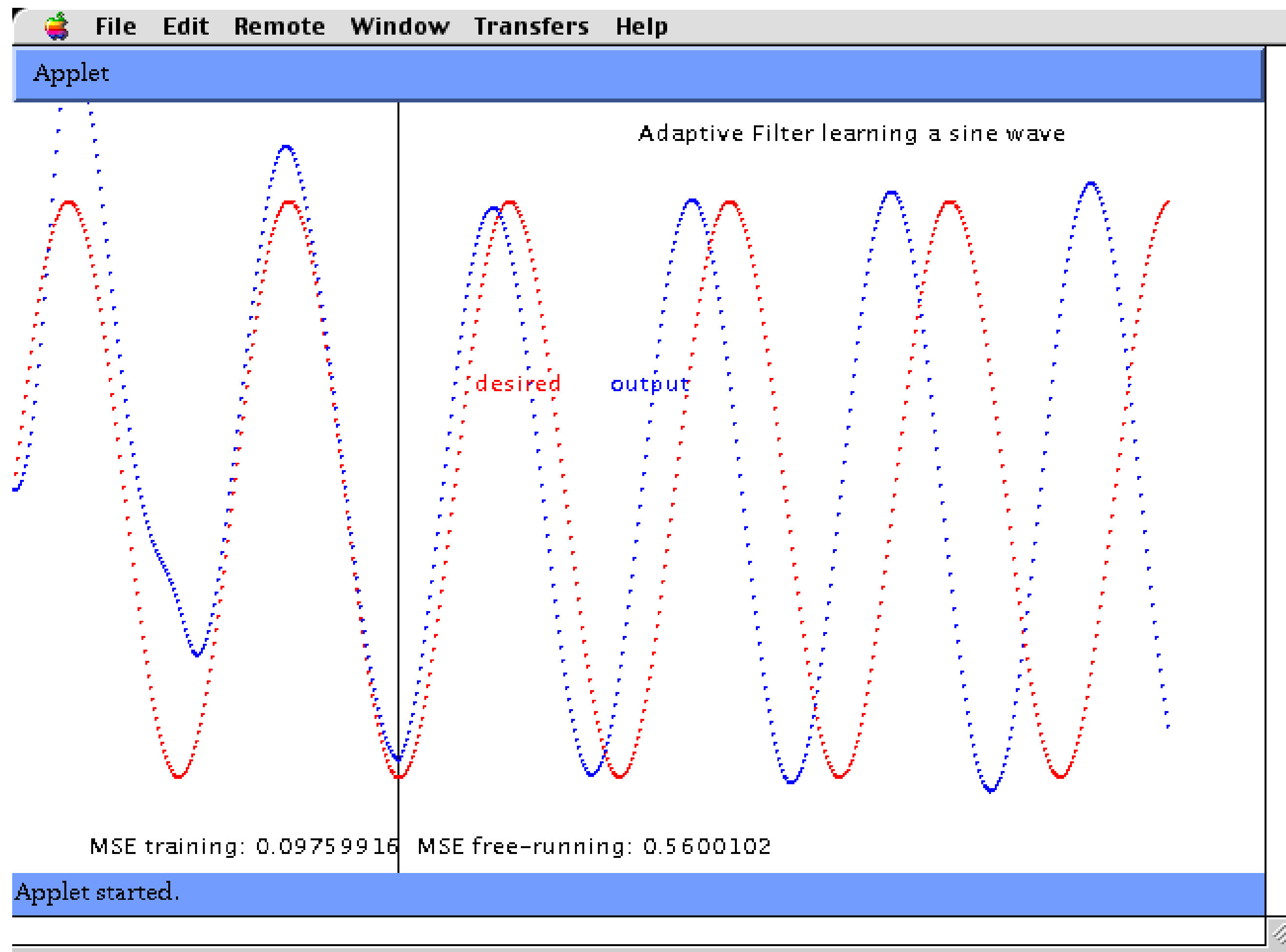
Free-Running after Training

(My applet on knuth: cd /cs/cs152/adaptive_filter; go)



Here the sine wave was removed and the output (prediction) fed back into the input

The same Filter at 1.75 x orig frequency



Comparison

- Classical filters are designed for one specific band.
- Adaptive filters can adapt to time-varying signal regimes.

Adaptive Noise Cancellation (ANC)

- An information signal could be **polluted with noise** from an uncorrelated source.
- If the noise contribution could be identified, it could be **subtracted** from the polluted signal, resulting in the information signal.
- Usually there is no opportunity to obtain the noise source directly.
- However, a modified version of the noise source can often be obtained, e.g. from a separate microphone.
- This modified version can be used to **learn the contribution of the noise** to the polluted signal.
- The learned contribution can be subtracted from the polluted signal, to give a purer signal.

Adaptive Noise Cancellation (ANC)

- Mixed = Signal + f(Noise)
- Noise Channel = g(Noise)
- Noise Channel g(Noise) is input to the network, which tries to learn f(Noise).
- The network output is thus subtracted from the Mixed Signal:

Signal = Mixed - f(Noise)

Network output = f(Noise) - Network error

Thus

Signal ≈ Mixed - Network output

Network error = f(Noise) - Network This should approach 0.

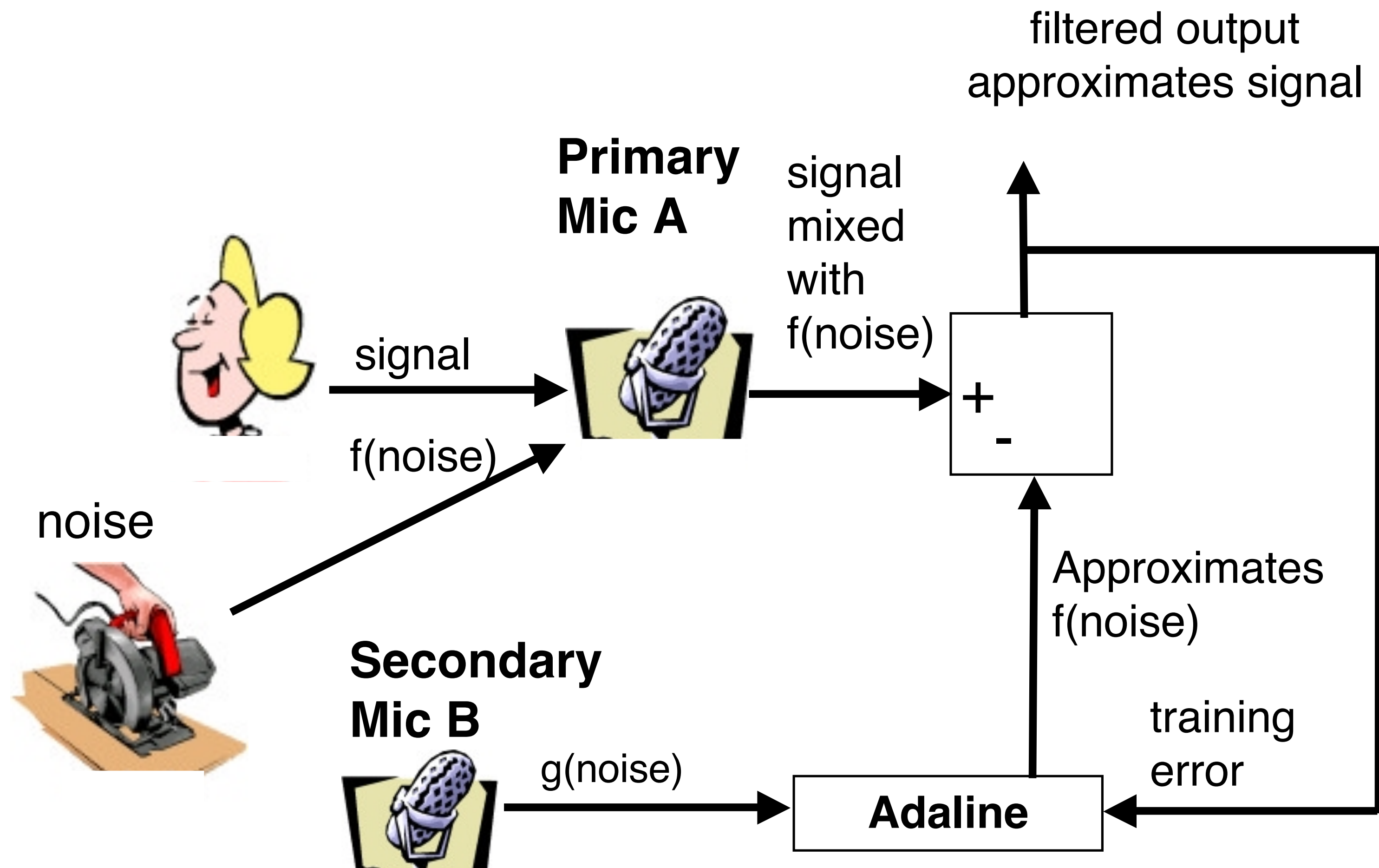
Filtered signal = Mixed - Network output ≈ Mixed - f(Noise) = Signal

Therefore

Filtered signal ≈ Signal

Noise-Reduction Scenario

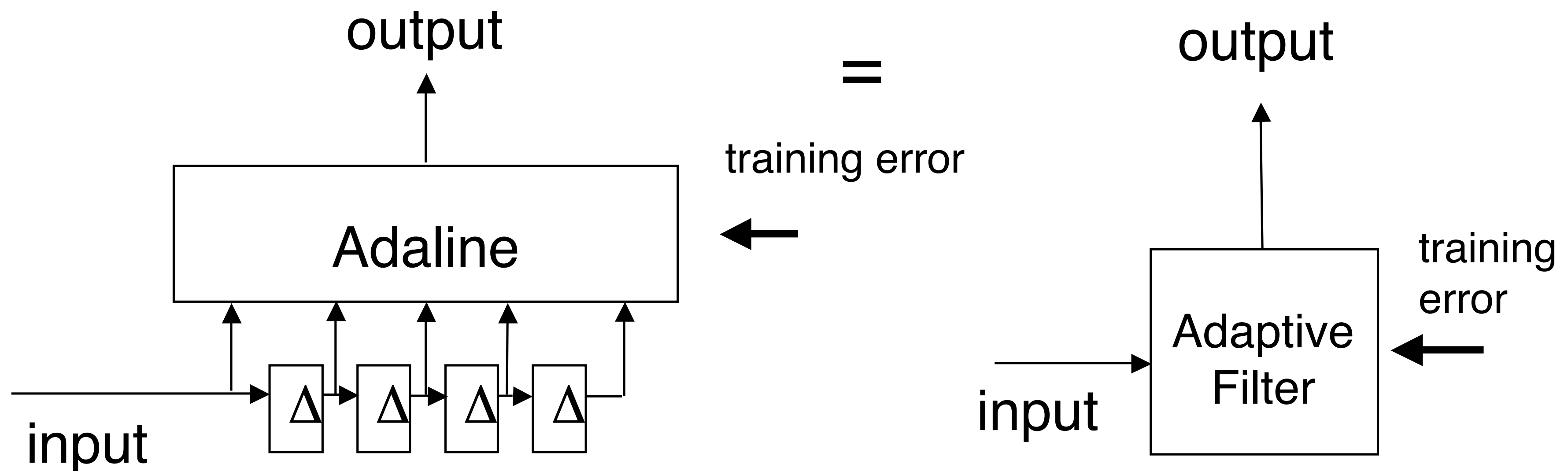
Filter “Learns the Noise”



(g(noise) is ***correlated*** *with noise*, but we don’t know the exact amplitude or phase)

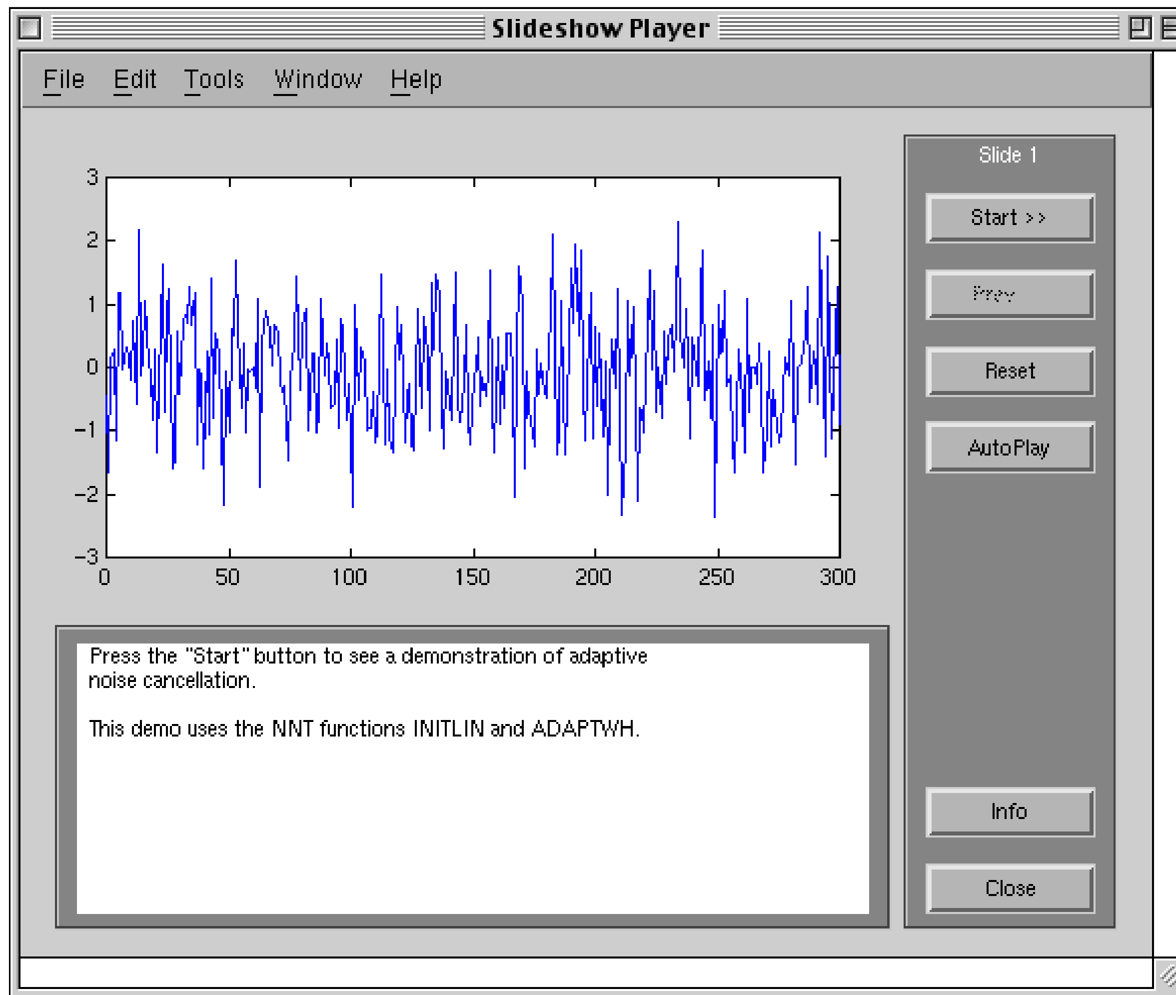
Adaptive Filter Component

- Adaptive filter component learns produce output from input as guided by training error signal



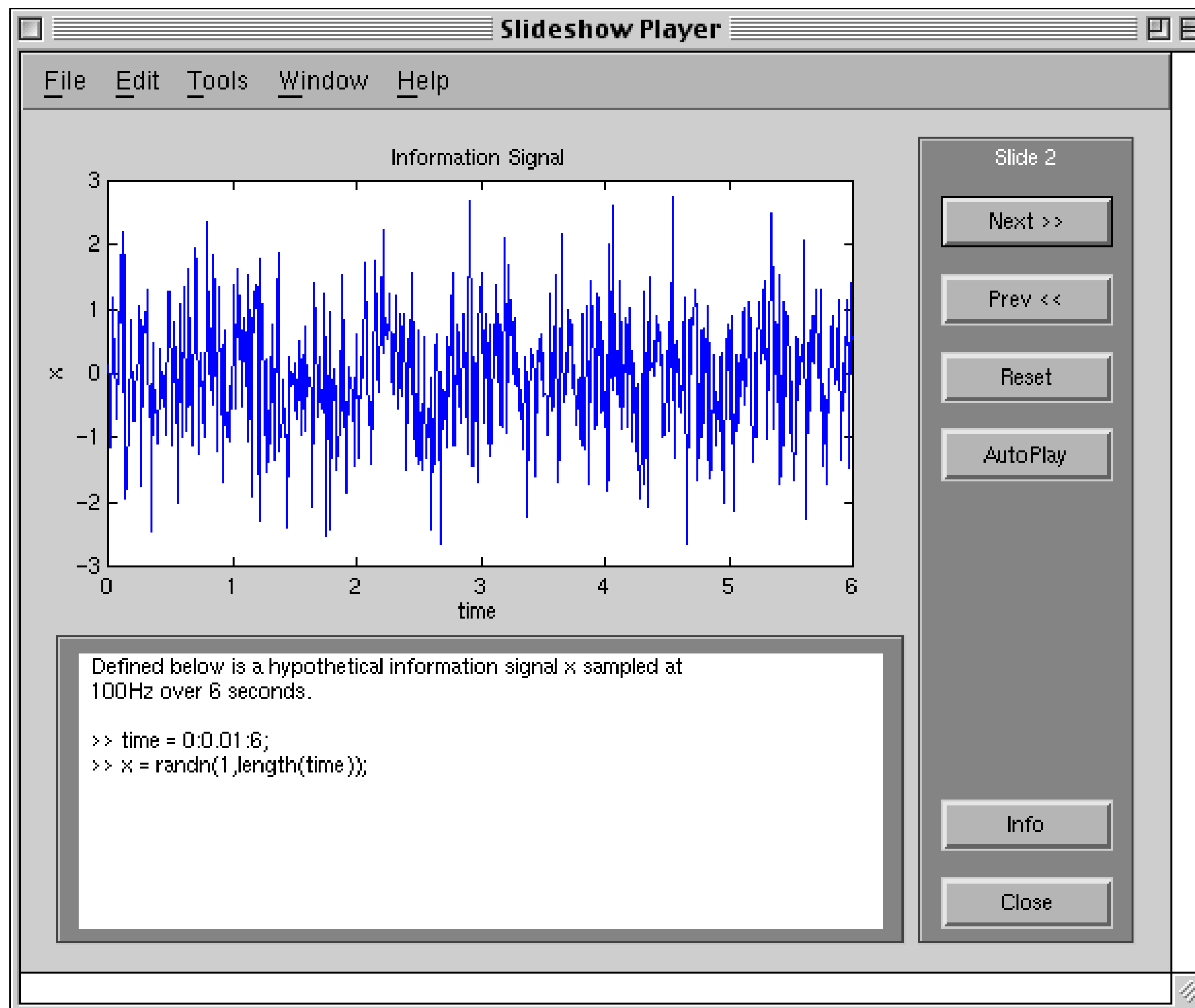
applin5 Demo

(no longer exists; needs rewrite for new NN functions)



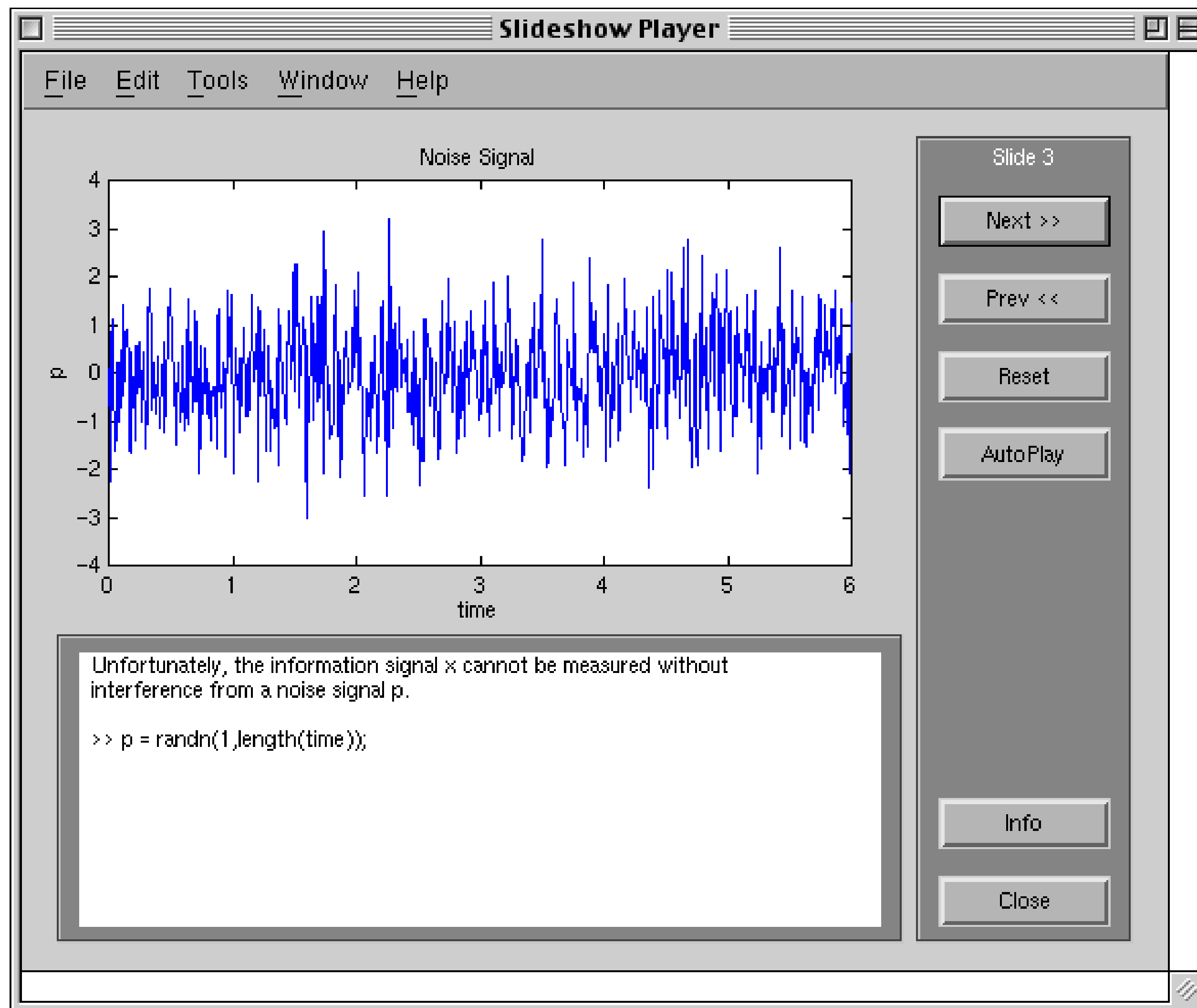
Information Signal

(without noise, not usually known, but we're creating it)

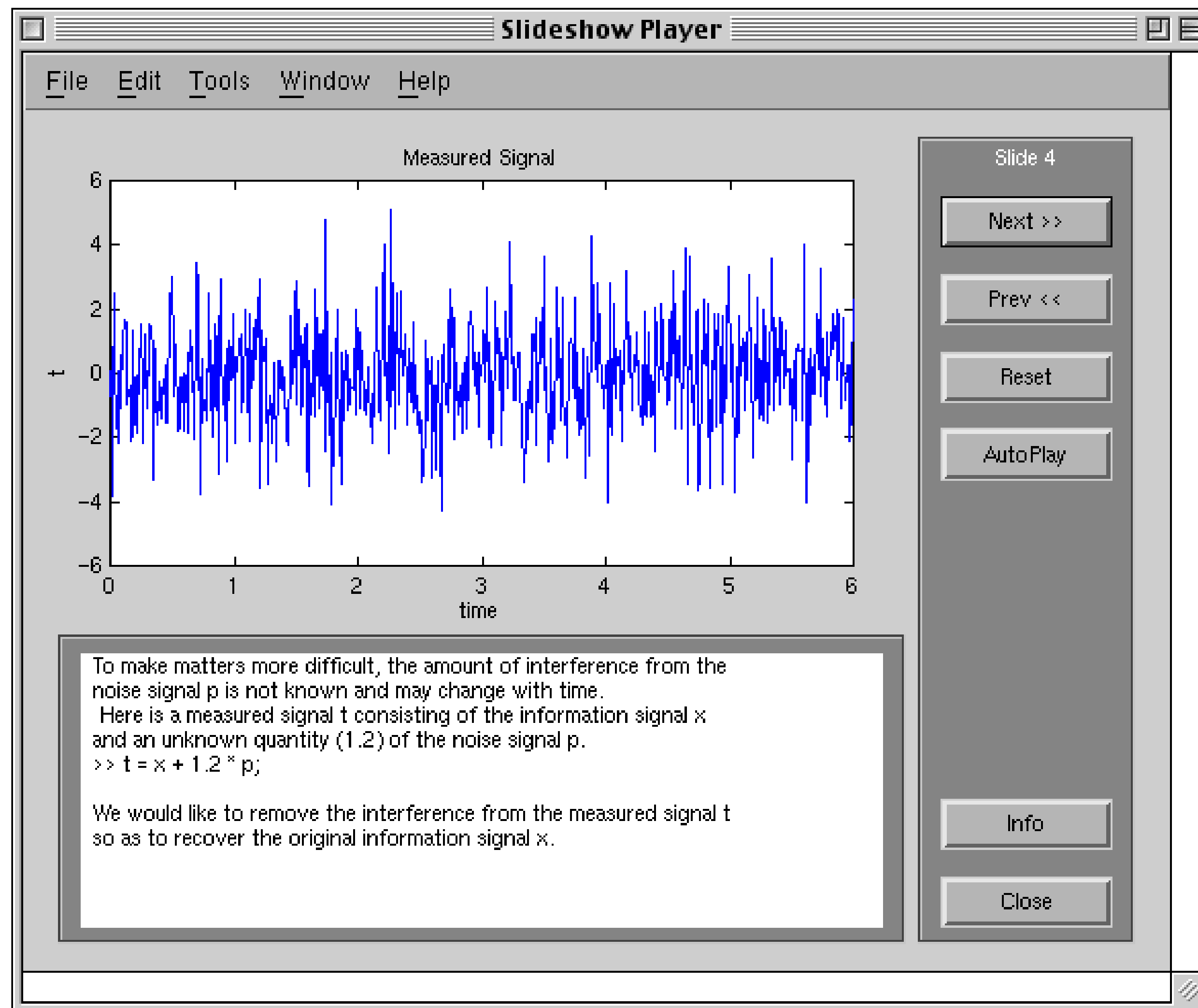


Noise Signal

(not usually known, trying to learn it)



Measured Signal (with noise)



Initializing Filter

Slideshow Player

File Edit Tools Window Help

Measured Signal

Slide 5

Next >>

Prev <<

Reset

AutoPlay

Info

Close

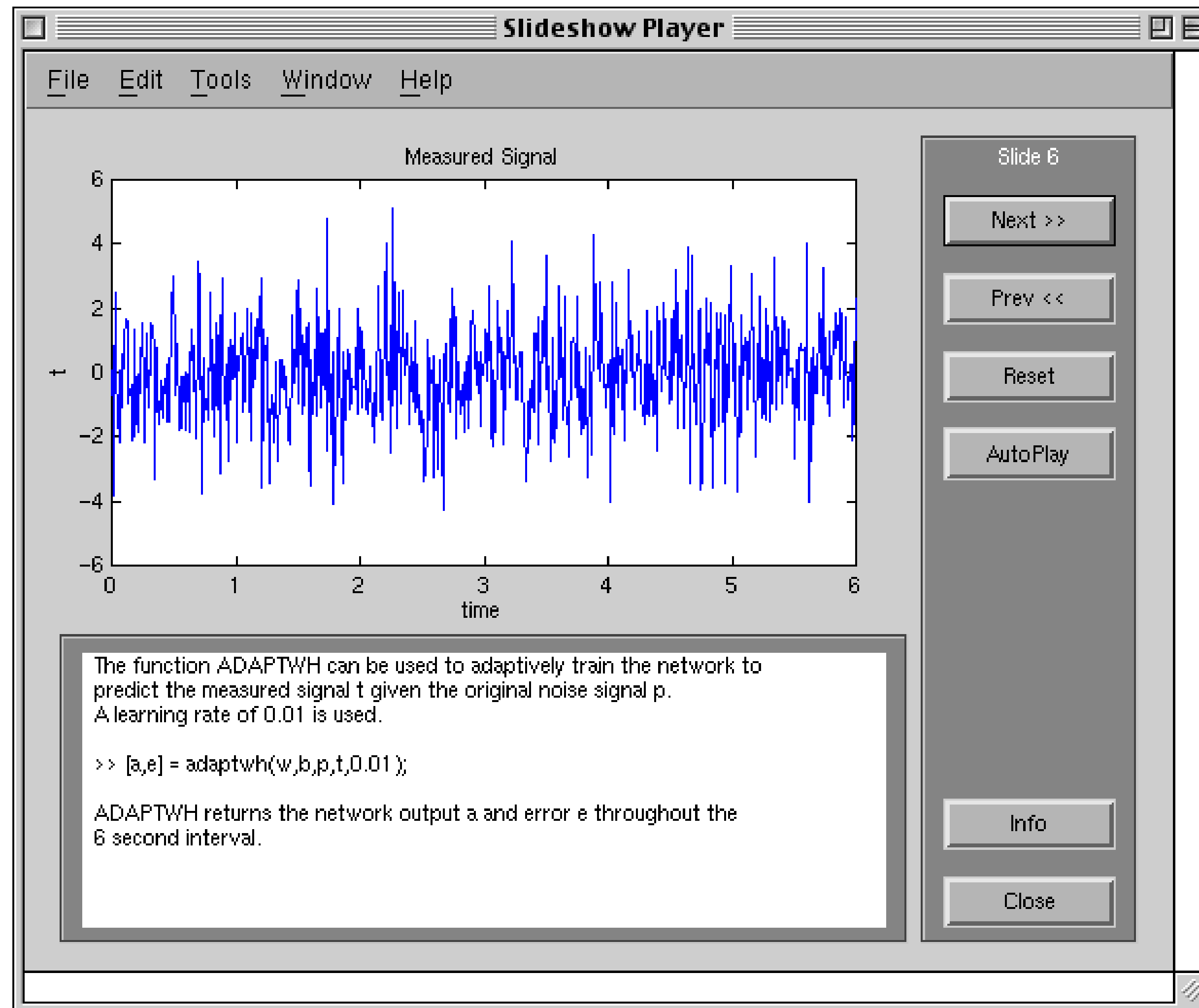
time

t

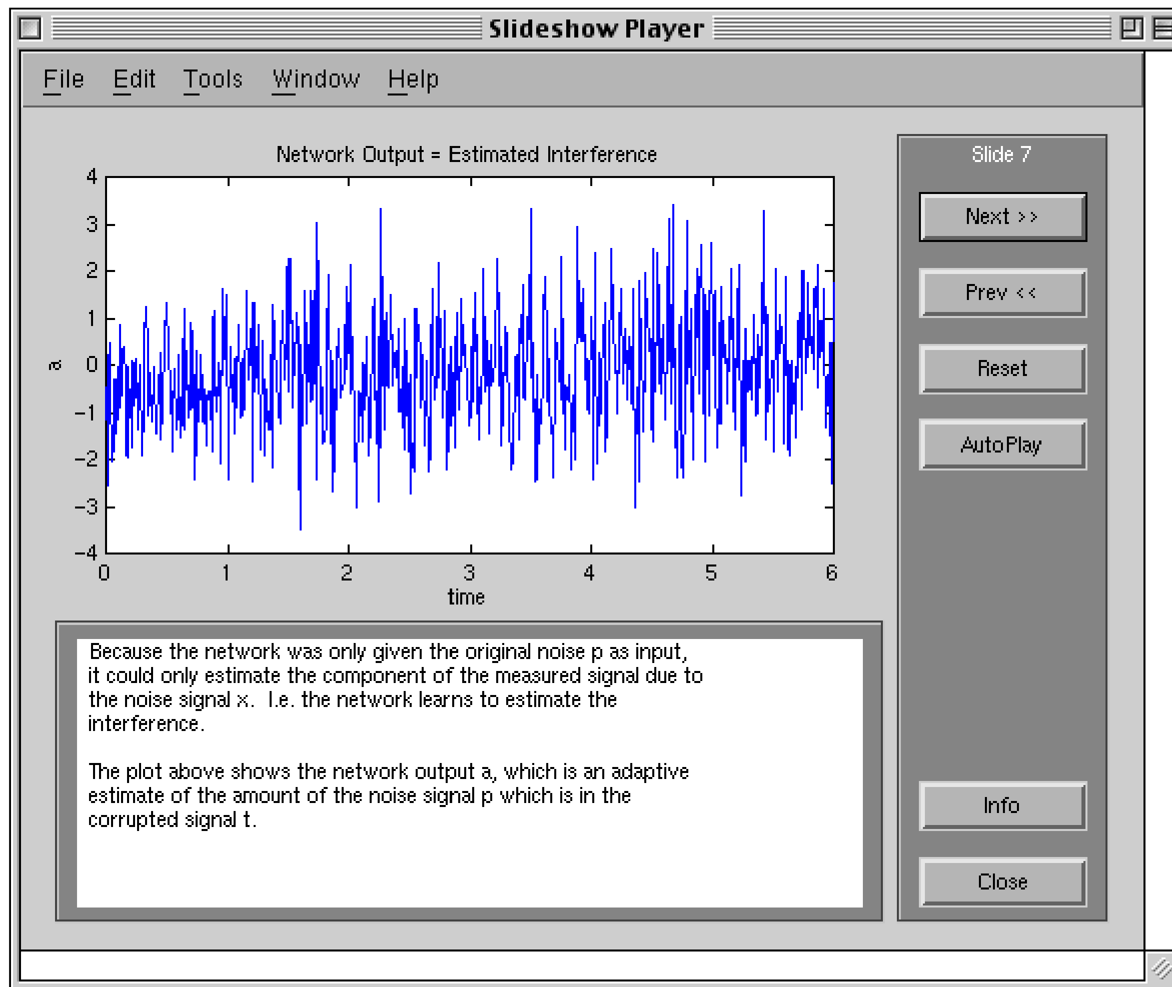
We will use the function INITLIN to create a linear network to discover the relationship between the noise p, and the interference it contributes to t.

```
>> [w,b] = initlin(p,t);
```

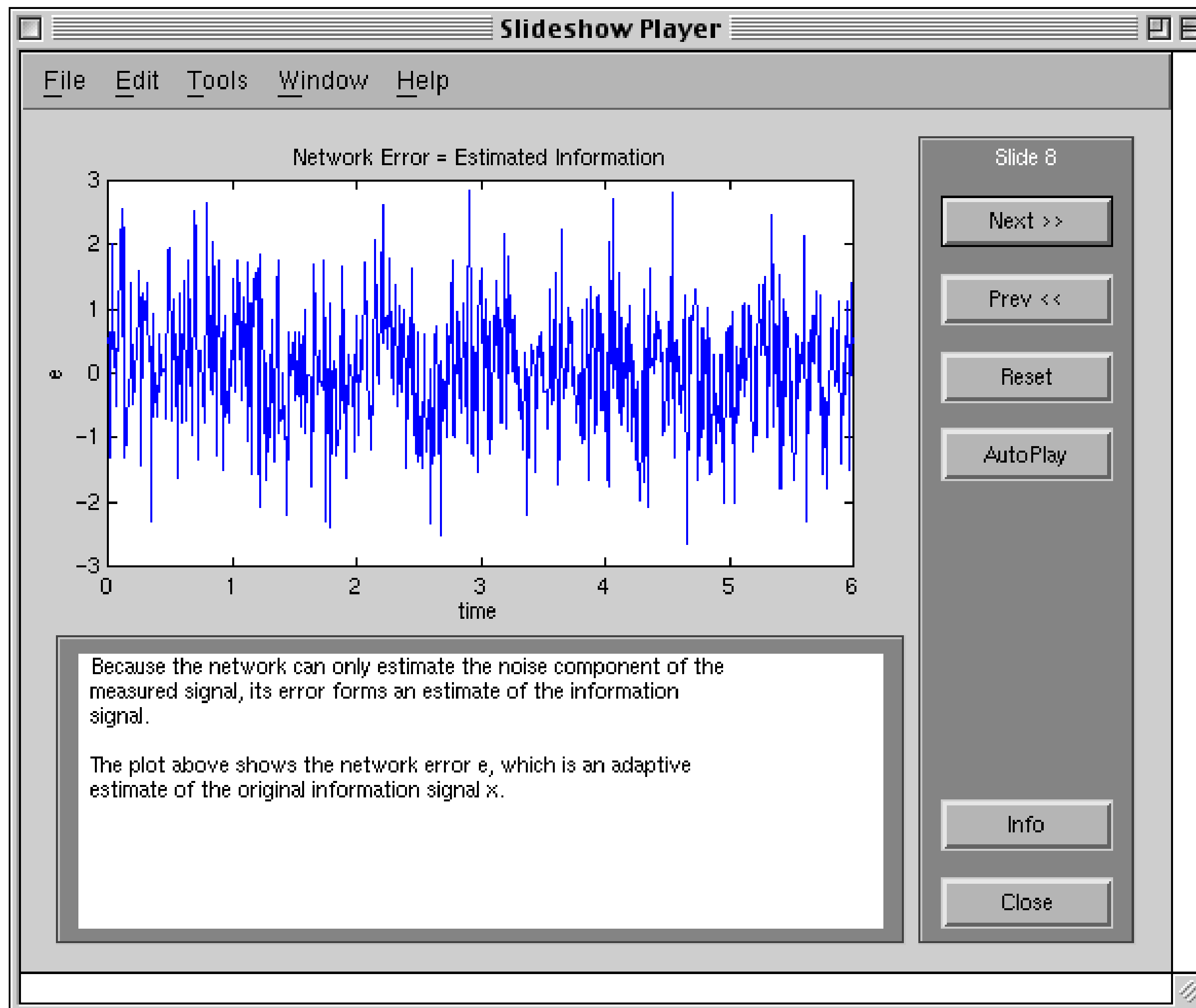
Adapting using Widrow-Hoff



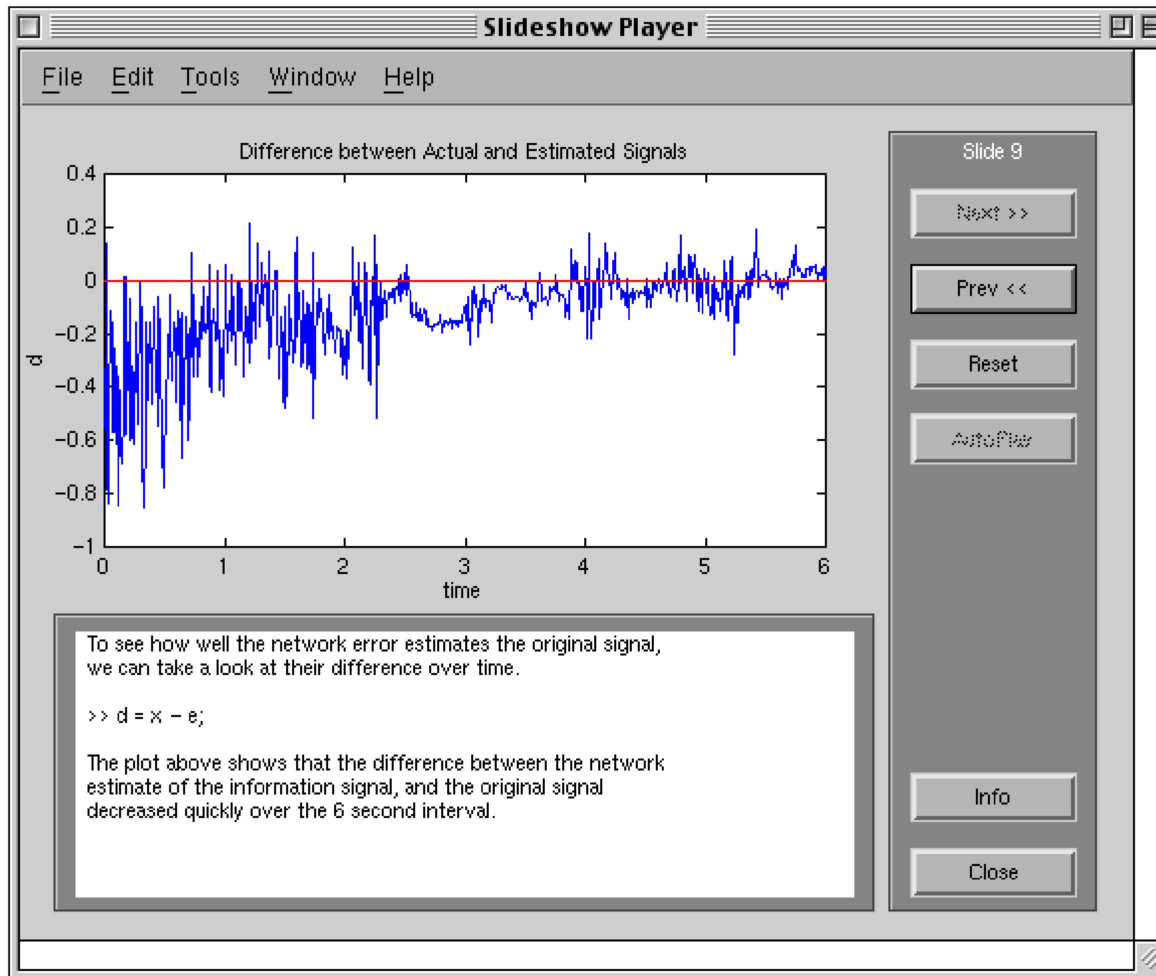
Noise Estimated by Filter



Estimated Information
(= Noisy Signal - Estimated Noise)



Error after Filtering (= Estimated Info - Actual Info)



Example: ANC (Active Noise Cancellation) Headphones Research

http://www.nalanda.nitc.ac.in/industry/appnotes/Texas/computing/spra160.pdf

THE FILTERED-X LMS ALGORITHM

The filtered-X LMS algorithm developed by Widrow [8] seeks the controller coefficients (weight vector) of $C(q^{-1},k)$, which minimize the mean-squared error, $\xi = E[e^2(k)]$. The mean-squared error is the average power of the error microphone signal. To accomplish this task, a gradient method is used. In the feedforward configuration, the component of $e(k)$ that is correlated with $x(k)$ is removed, leaving only $v(k)$. It is this feature that allows the selectivity property in an ANC system.

The controller weight vector, $\theta_C(k) = [c_0(k), c_1(k), \ldots, c_{n_C}(k)]^T$ is adjusted in the direction of the gradient

$$\nabla = \frac{\partial}{\partial \theta_C} E[e^2(k)] \quad . \qquad (1)$$

Because the exact gradient is unavailable, an estimate must be used. In the LMS algorithm, the instantaneous value of the error squared, $e^2(k)$, is used

Conclusions: ANC (Active Noise Cancellation) Headphones Research

http://www.nalanda.nitc.ac.in/industry/appnotes/Texas/computing/spra160.pdf

Commercially available active noise control headphones rely on fixed analog controllers to drive "anti-noise" loudspeakers. Our design uses an adaptive controller to optimally cancel unwanted acoustic noise. This headphone would be particularly useful for workers who operate or work near heavy machinery and engines because the noise is selectively eliminated. Desired sounds, such as speech and warning signals, are left to be heard clearly.

The filtered-X LMS adaptive control algorithm is implemented on a TI TMS320C30 DSP that drives a Sony CD550 headphone/microphone system. Our experiments indicate that adaptive control results in a dramatic improvement in performance over fixed control for narrow-band noise. This is achieved by the self-optimizing and tracking capabilities of the adaptive controller in response to different types of ambient noise. Further performance improvement for wide-band noise can be achieved by using more complex controller structures and faster converging algorithms.

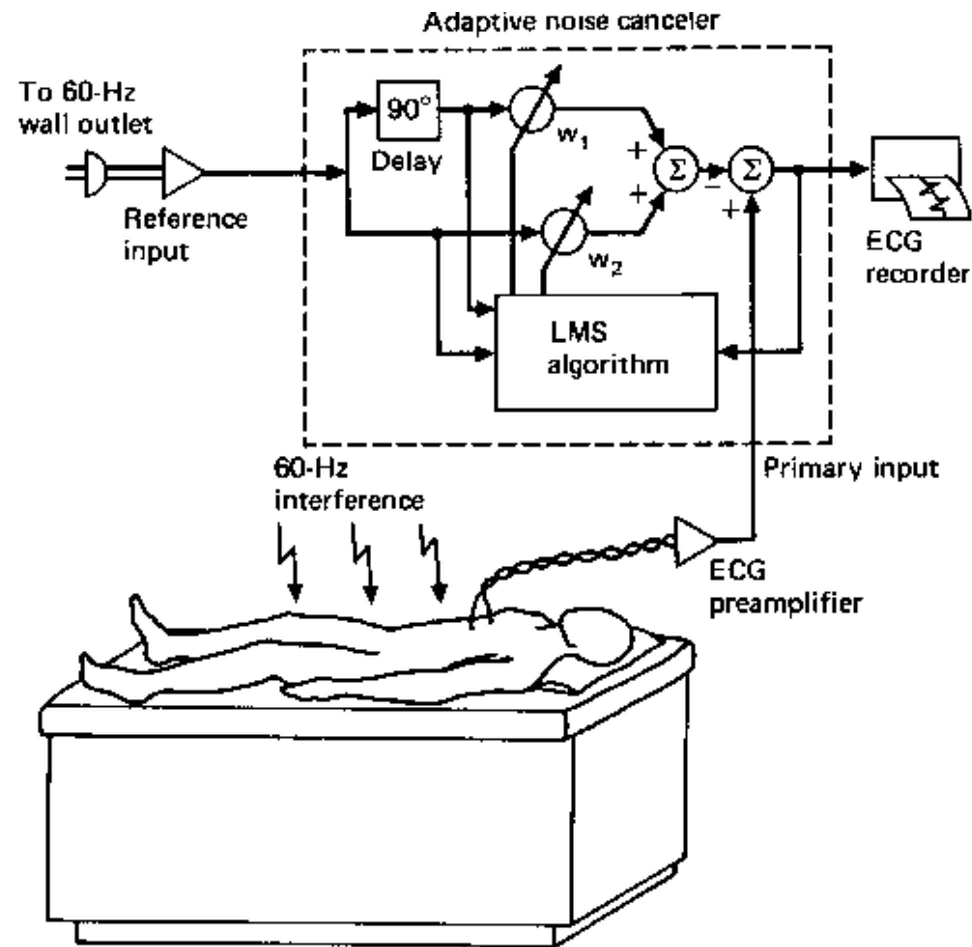
Contextual Nomenclature

- Classical filters don't adapt
  - (Lowpass / Highpass / Bandpass) filters
- Adaptive filters adapt
  - LMS filter (least-mean-squared)
  - RLS filter (recursive least squares, based on pseudo-inverse, not as stable)
  - Kalman filter (based on a stochastic state-space model)

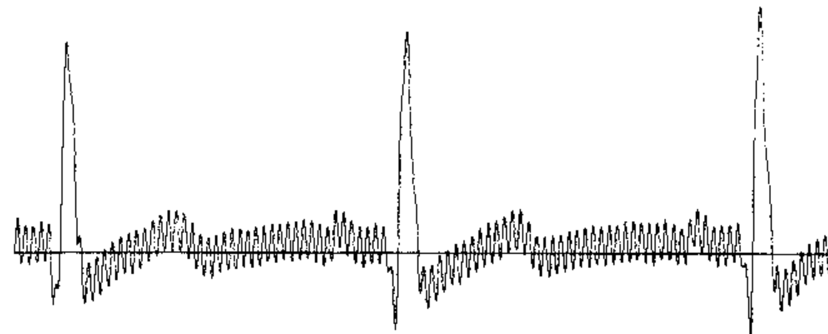
Other Applications

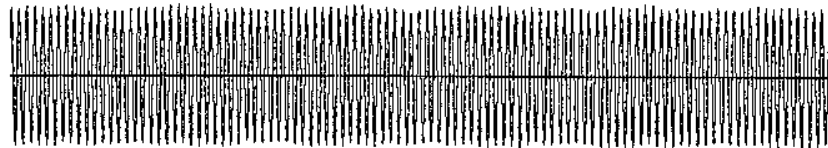
- EKG filtering (60 Hz noise)
- Fetal monitoring (baby’s heart - mother’s heart)
- Telephone echo cancellation
- Conference telephones

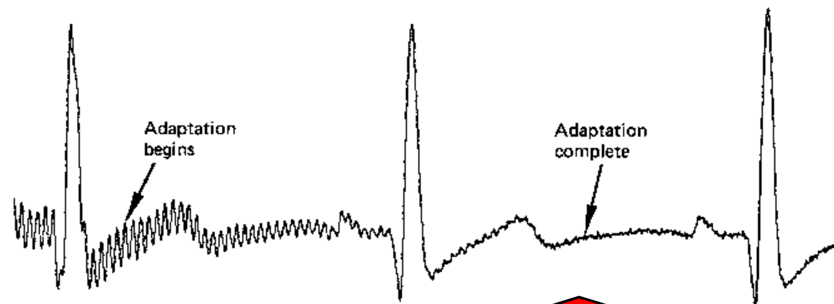
60 Hz Noise in EKG

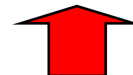


60 Hz Noise in EKG









Fetal Heartbeat Monitoring

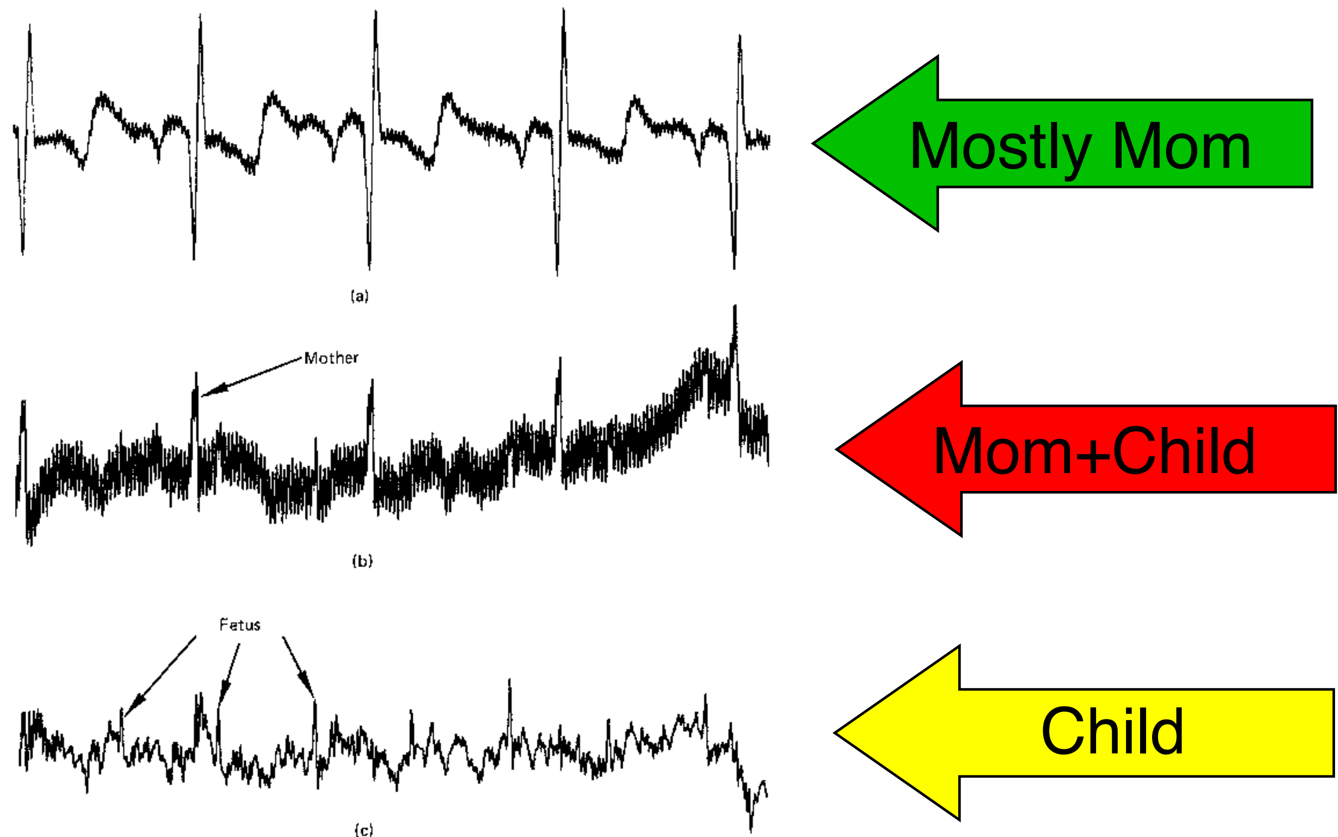
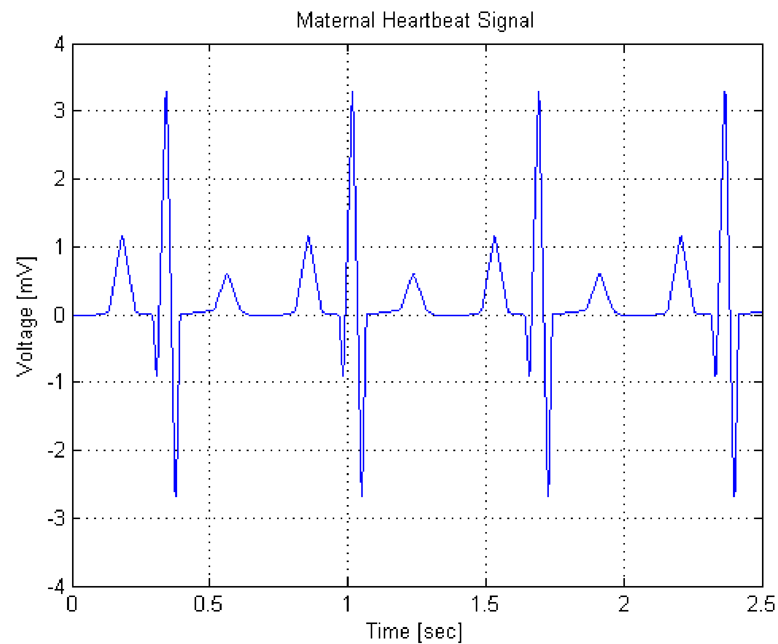


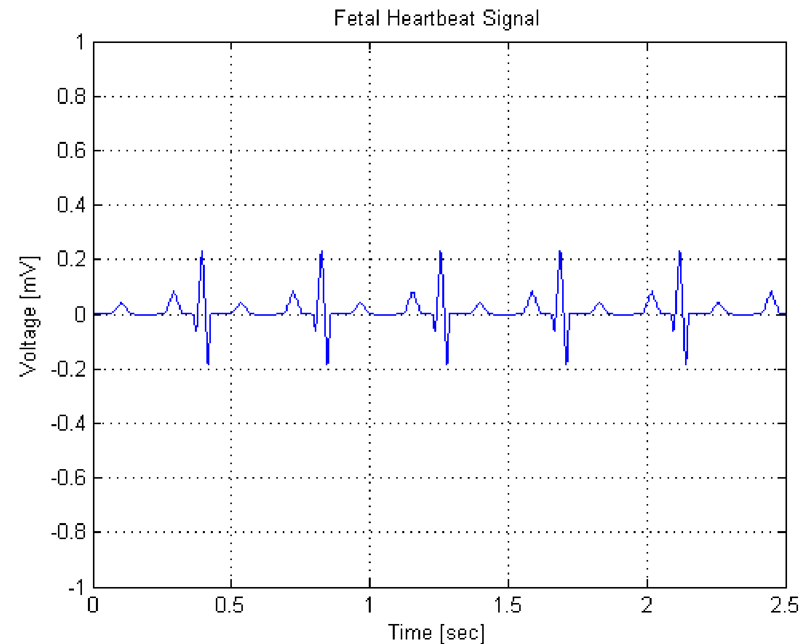
Figure 12.21 Result of wide-band fetal ECG experiment (bandwidth, 0.3–75 Hz; sampling rate, 512 Hz): (a) reference input (chest lead); (b) primary input (abdominal lead); (c) noise canceller output. From B. Widrow et al., *Adaptive Noise Canceling: Principles and Applications*, © December 1975, IEEE.

ANC Applied to Fetal Electrocardiography
Matlab Simulation Code

http://www.mathworks.com/products/filterdesign/demos.html?file=/products/demos/shipping/filterdesign/adaptncdemo.html

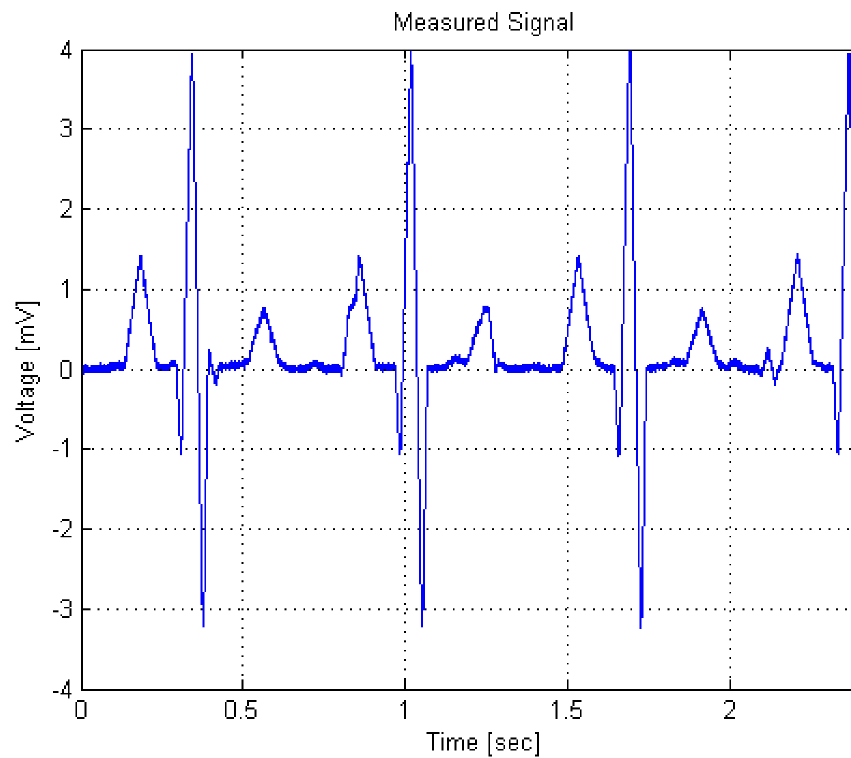
Note: Uses Filter Design Toolbox adaptfilt, rather than NN Toolbox



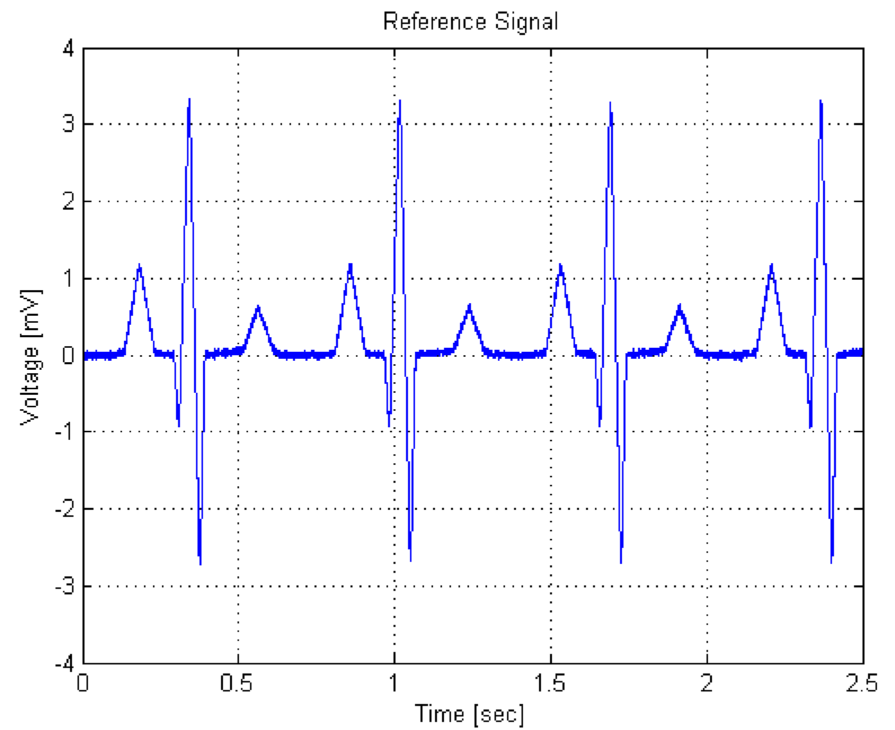


Microphone Inputs

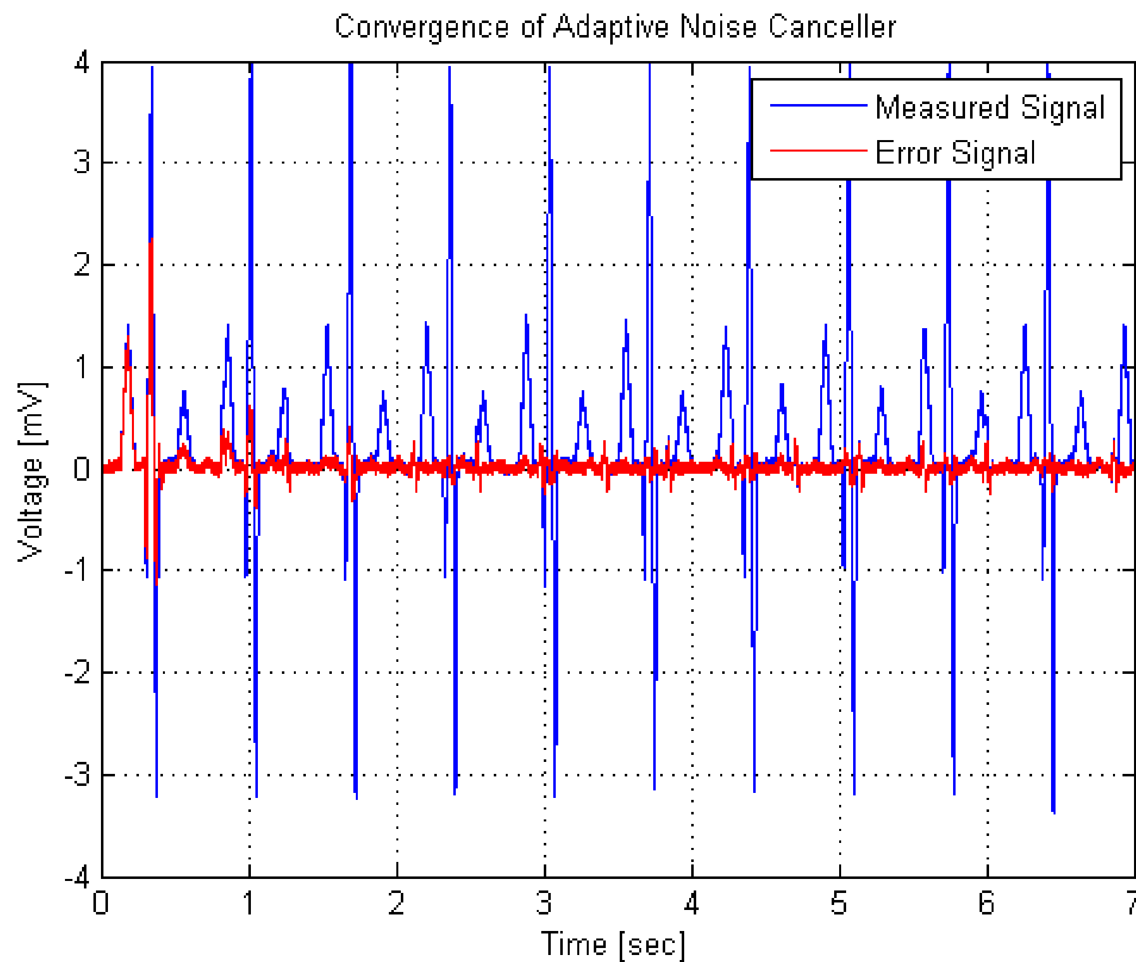
Mom+Child



Mostly Mom



ANC Convergence

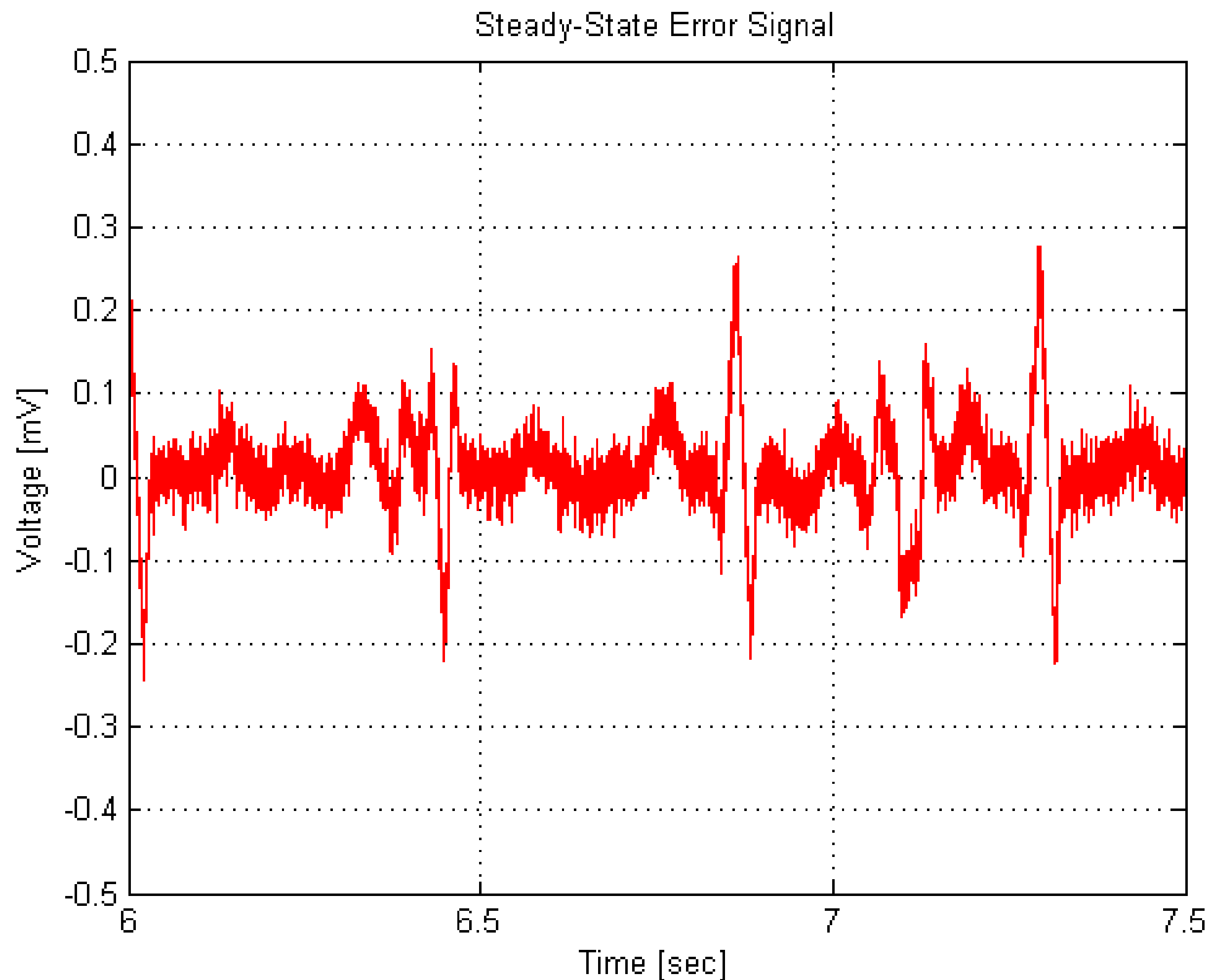


Mom+Child

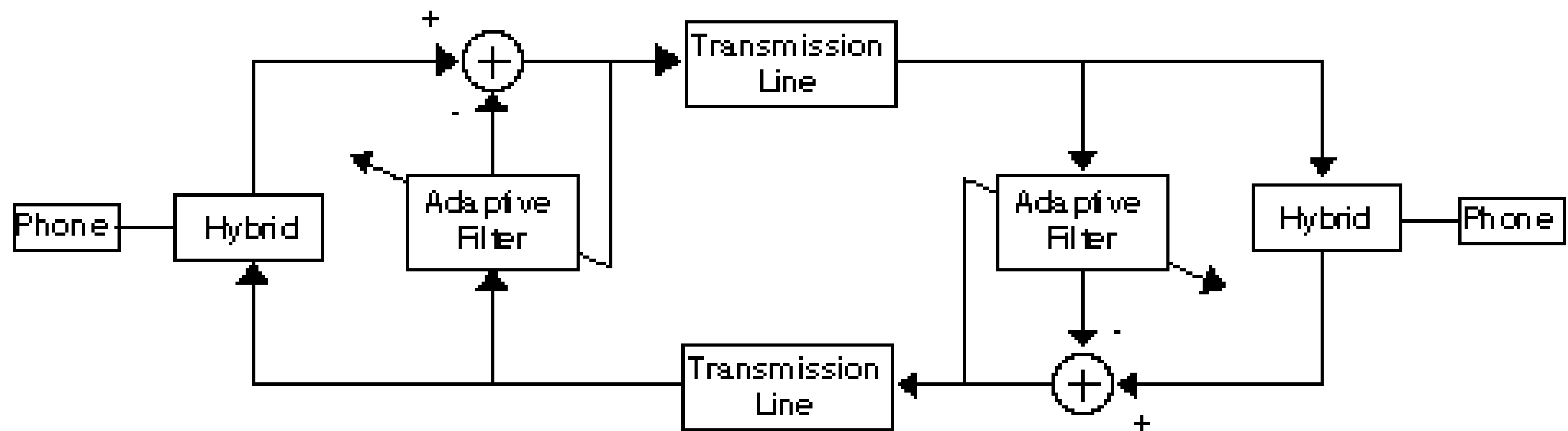
vs. Error

Learned Error estimates Fetal Heartbeat

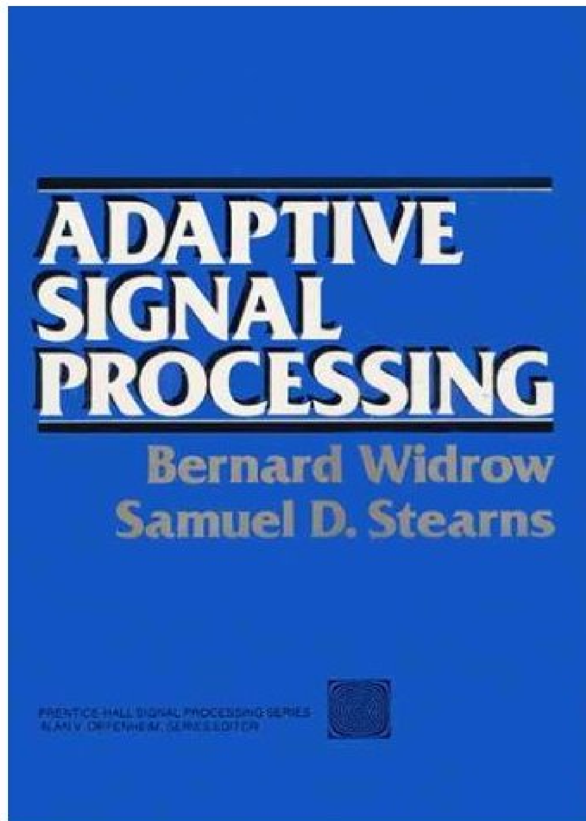
Estimated heartbeat rate of child?



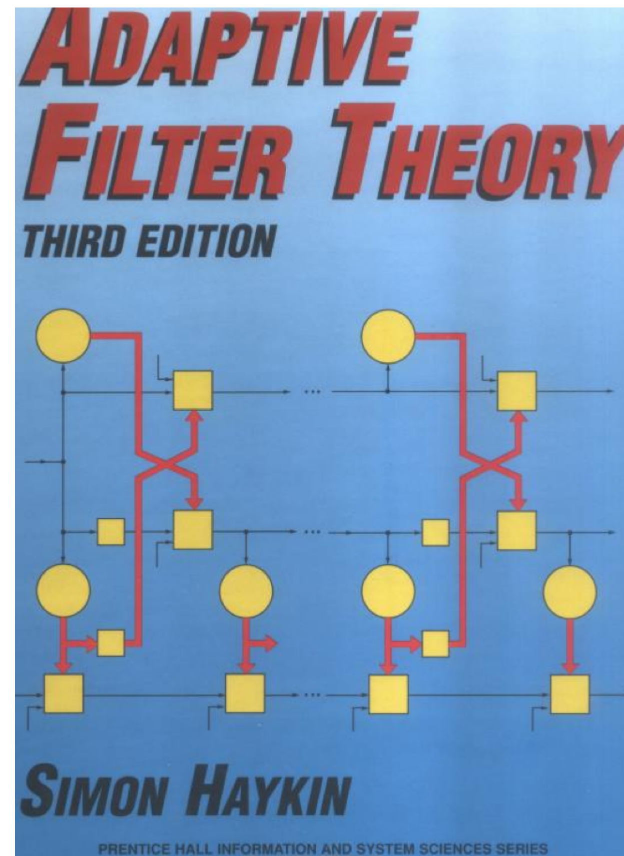
Telephone Echo Cancellation



References



Widrow & Stearns, 1985

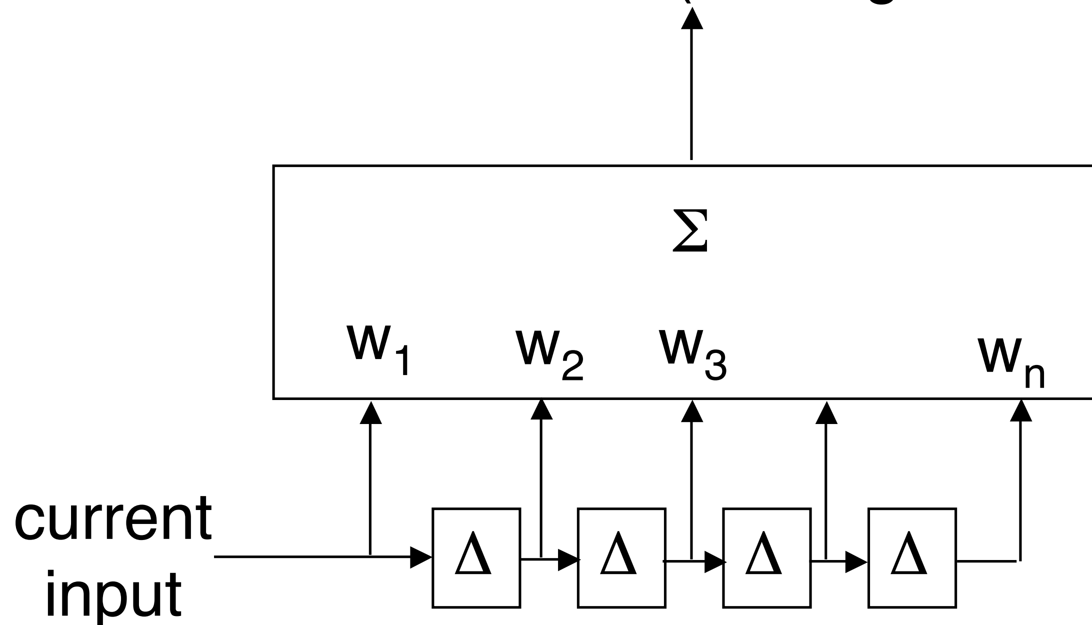


Haykin, 1995

Additional Nomenclature

Adaline form is eample of a **FIR** (**F**inite **I**mpluse **R**esponse) filter.

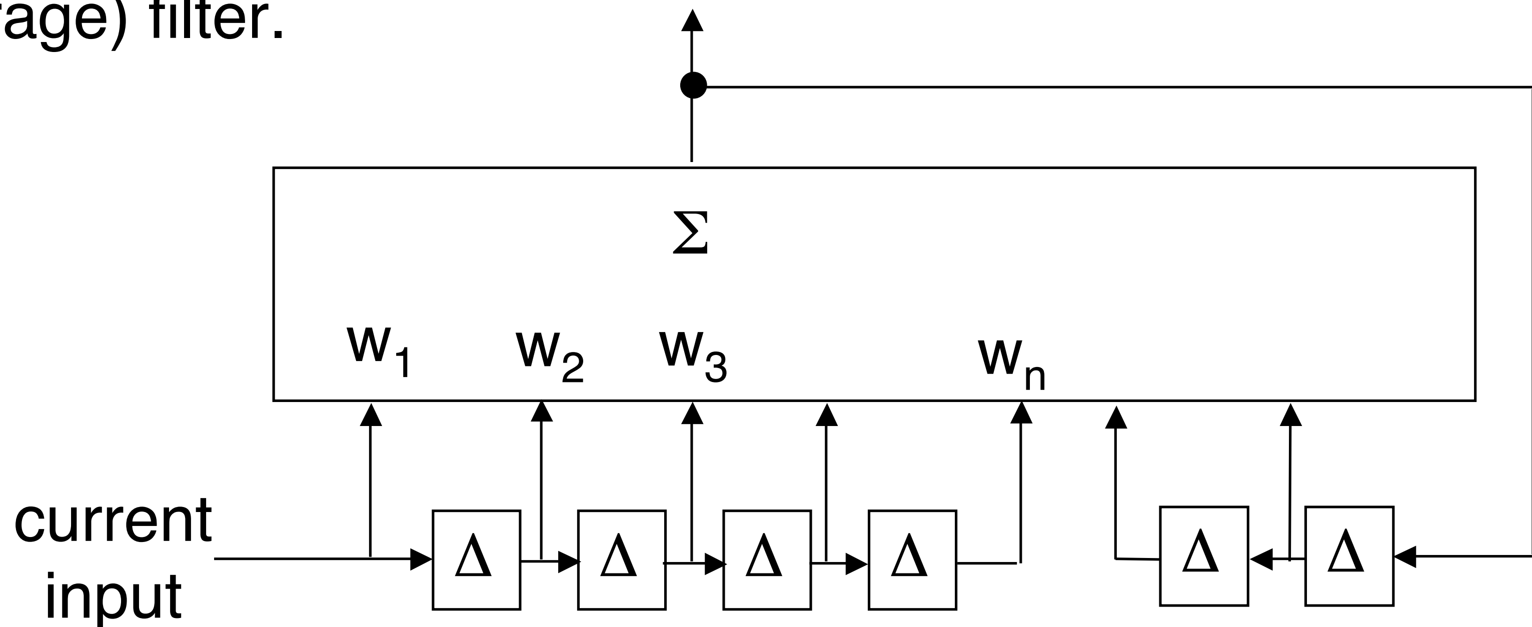
In statistics, it is called an **MA** (**M**oving **A**verage) filter.



Additional Nomenclature

If we add **feedback**, we have an **IIR** (**I**nfinite **I**mpluse **R**esponse) filter.

In statistics, it is called an AR**MA** (**A**uto**R**egressive **M**oving **A**verage) filter.

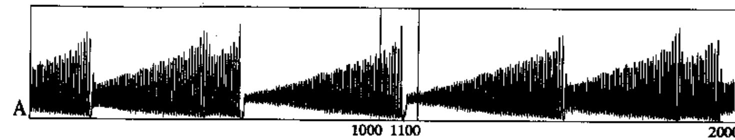


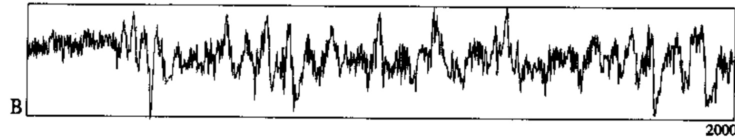
Classical Fitting of Time-Series

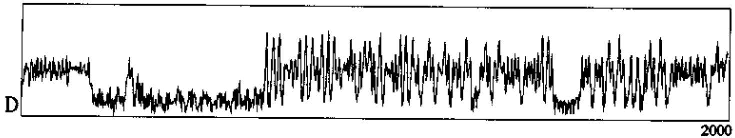
- A given ARMA's coefficients can be fit to generate ***approximately*** a given time series by using least-squares estimation on a set of simultaneous linear equations known as the "Yule-Walker equations".

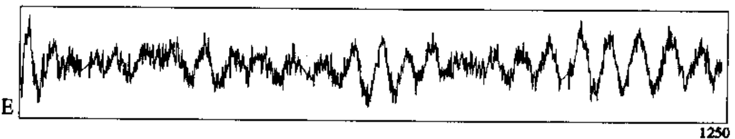
- Reference: Weigend and Gershenfeld (eds.), Time series prediction, Addison-Wesley, 1994.

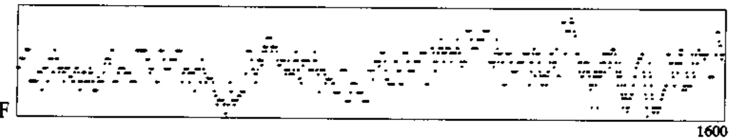
Sante Fe Institute Time Series Competition
6 unknown time series:
Challenge is to estimate what comes next











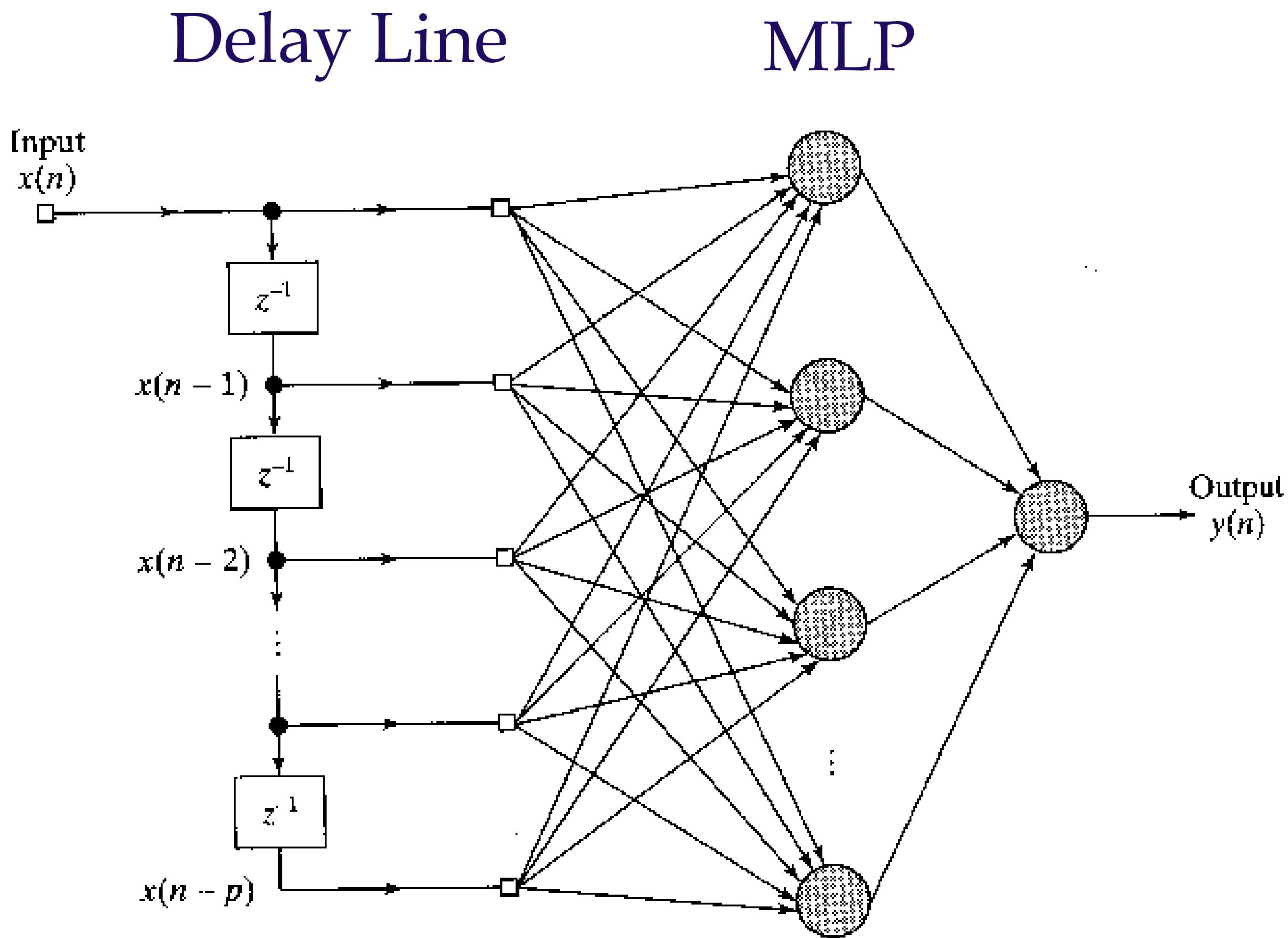
Other Models

- Time-Lagged Feed-Forward Networks, Time-Delay Neural Networks (TLFF, TDNN)
- FIR-Multi-layer networks (FIRNET)
- Backpropagation through time (BPTT)
- Real-Time Recurrent Learning (RTRL)
- Elman nets, Jordan nets
- Temporal difference method (TD(λ))

Time-Lagged Feed-Forward Networks (TLFF)

- An extension of the “Adaline” adaptive filter model
- Use an arbitrary feed-forward net (MLP) in place of the Adaline
- Train using ordinary backpropagation, analogous to LMS

Time-Lagged Feed-Forward Networks (TLFF)



Example TLFF Application

Bulletin of the Seismological Society of America, Vol. 98, No. 1, pp. 366–382, February 2008, doi: 10.1785/0120070002

PreSEIS: A Neural Network-Based Approach to Earthquake Early Warning for Finite Faults

by Maren Böse*, Friedemann Wenzel, and Mustafa Erdik

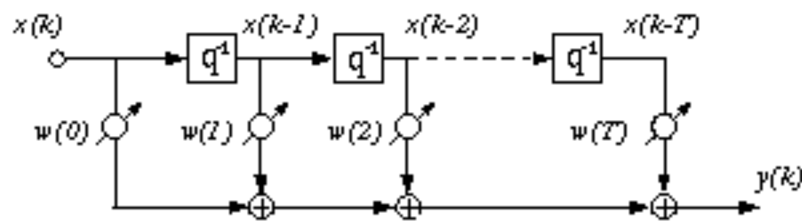
Discussion and Conclusions

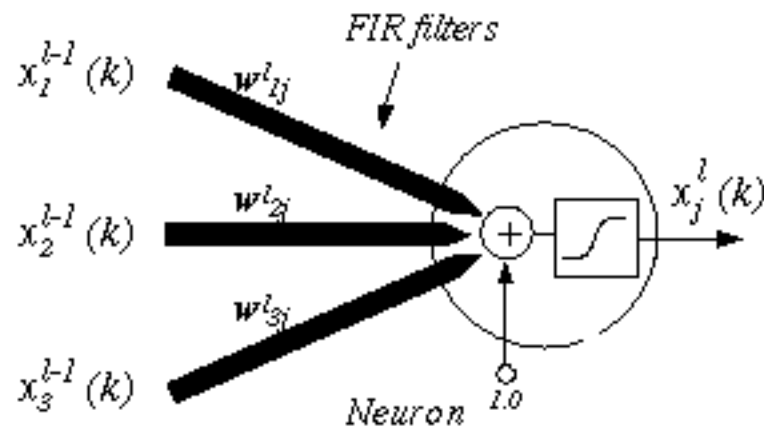
We have developed a new method for EEW that is based on two-layer feed-forward (TLFF) neural networks and that is applicable to finite fault. PreSEIS inverts time-dependent seismic attributes that are derived from ground-motion observations at different sensors in a seismic network. At regular timesteps after the triggering of the first EEW sensor by the propagating seismic *P* wave, PreSEIS estimates the most likely hypocenter location, moment magnitude, and rupture expansion of an earthquake in progress. Our studies demonstrate a clear and fast convergence of these estimates towards correct solutions with time. PreSEIS achieves a robust performance and, at the same time, issues first estimates only 0.5 sec after the triggering of the first sensors (i.e., is as fast as onsite warning approaches).

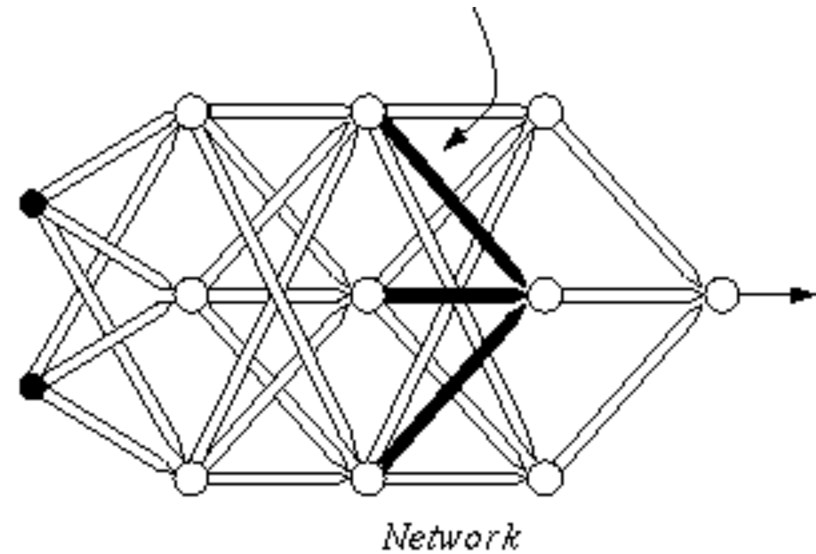
Another Approach: FIR Backpropagation

- Eric Wan (Stanford, OGI) came up with the idea of putting FIR filters **inside** a backprop network.
- **In place of each single weight** there is an entire FIR filter.
- Wan developed the training algorithm for such networks.

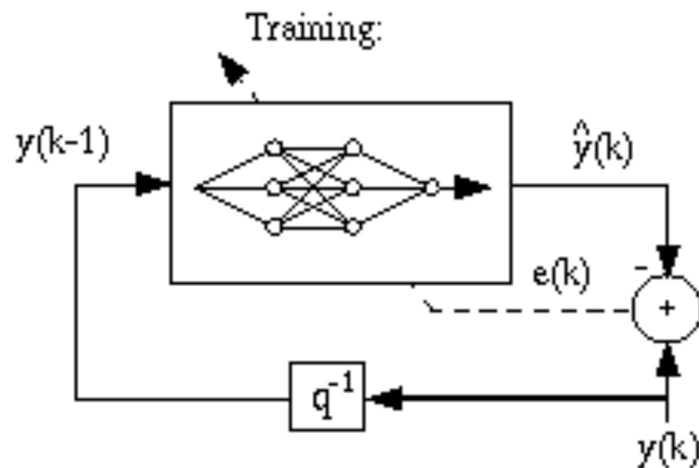
Wan's FIR Backprop Net



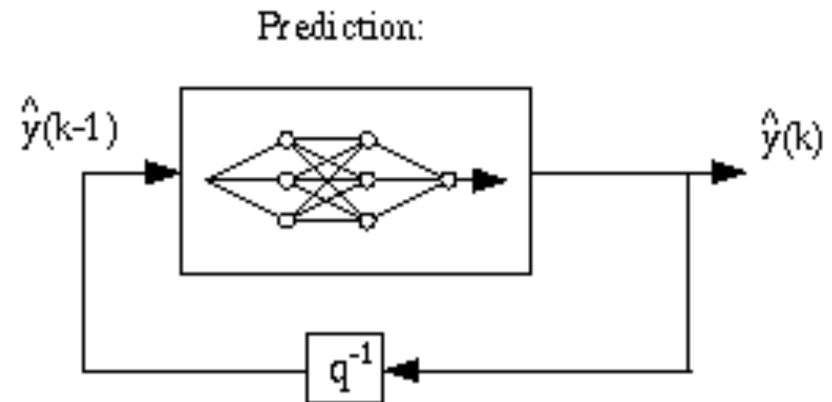




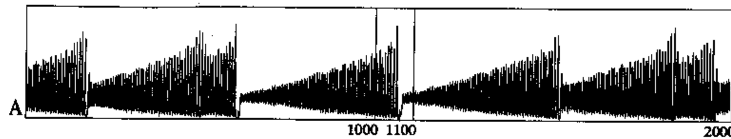
Recall: Training vs. Prediction

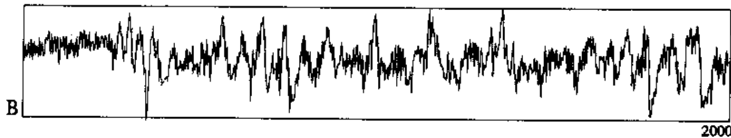


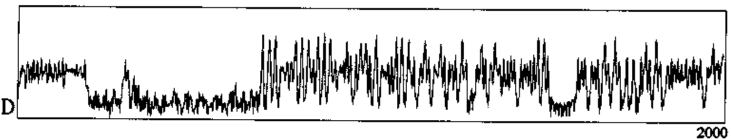
just a way of saying “delay”

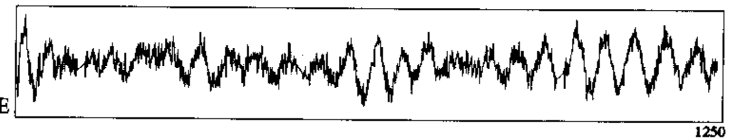


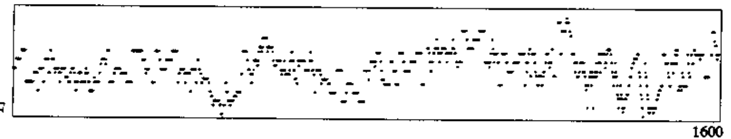
Sante Fe Institute Competition
6 unknown series: sources











A A clean physics experiment. 1000 points of the **fluctuations in a far-infrared laser**, approximately described by three coupled non-linear DE's.

B Physiological data from a patient with sleep apnea. 34,000 points of heart rate, chest volume, blood oxygen concentration, and EEG state of a sleeping patient.

C High-frequency **currency exchange rate** data. Ten segments of 3,000 points each of the exchange rate between the Swiss franc and the U.S. dollar, 1-2 minutes apart.

D Numerically generated series. A driven particle in a 4-dimensional nonlinear multiple-well potential (9 degrees of freedom) with a small nonstationary drift in the depths.

E Astrophysical data from a **variable white dwarf** star. 27,704 points in 17 segments of the time variation of the intensity.

F J.S. Bach's final **(unfinished) fugue** from *The Art of the Fugue*.

Wan's Entry in Sante Fe Institute Competition

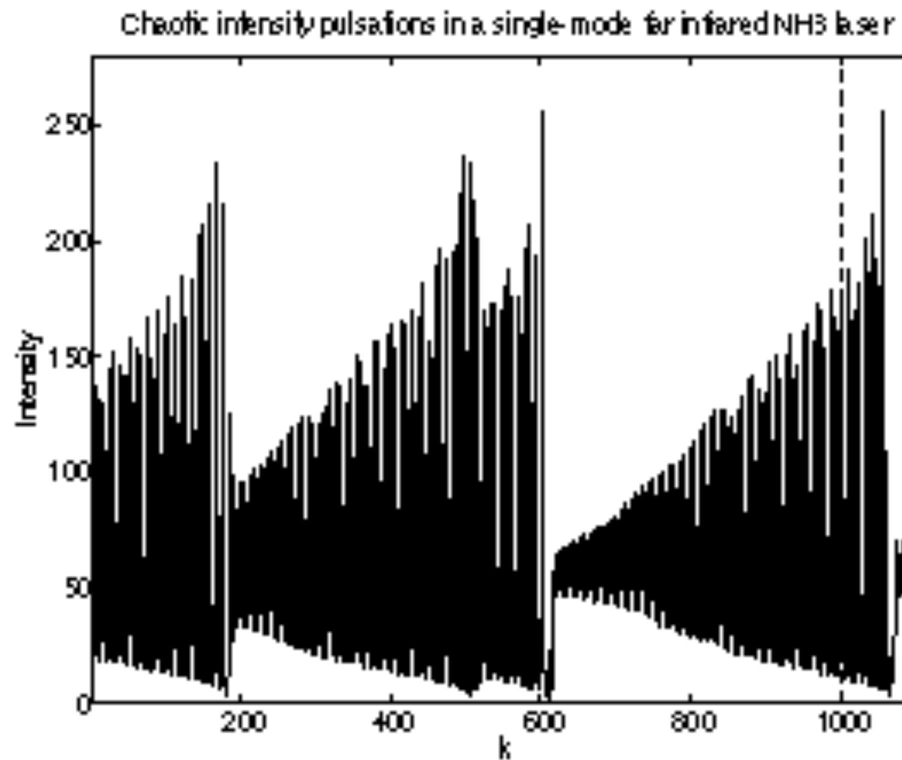
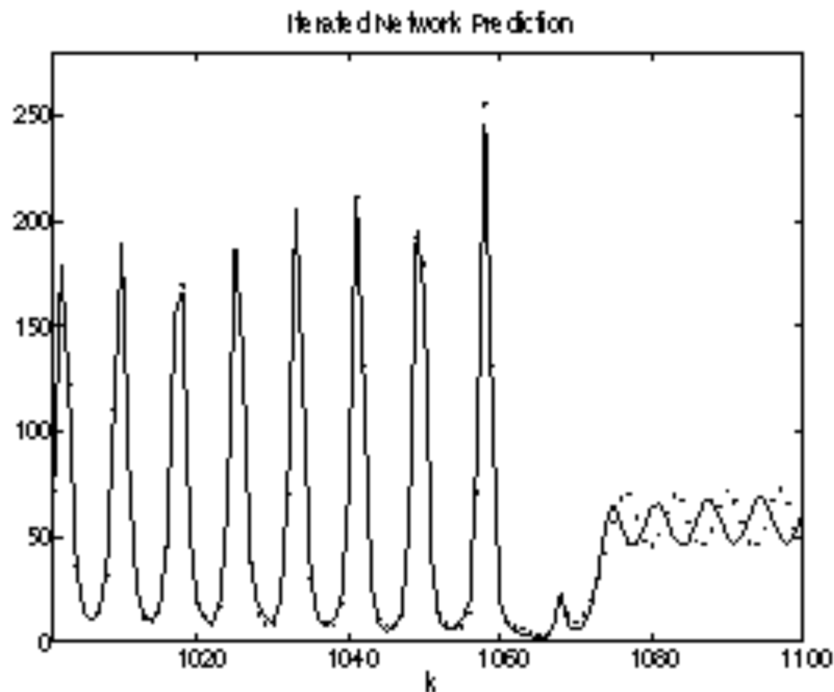
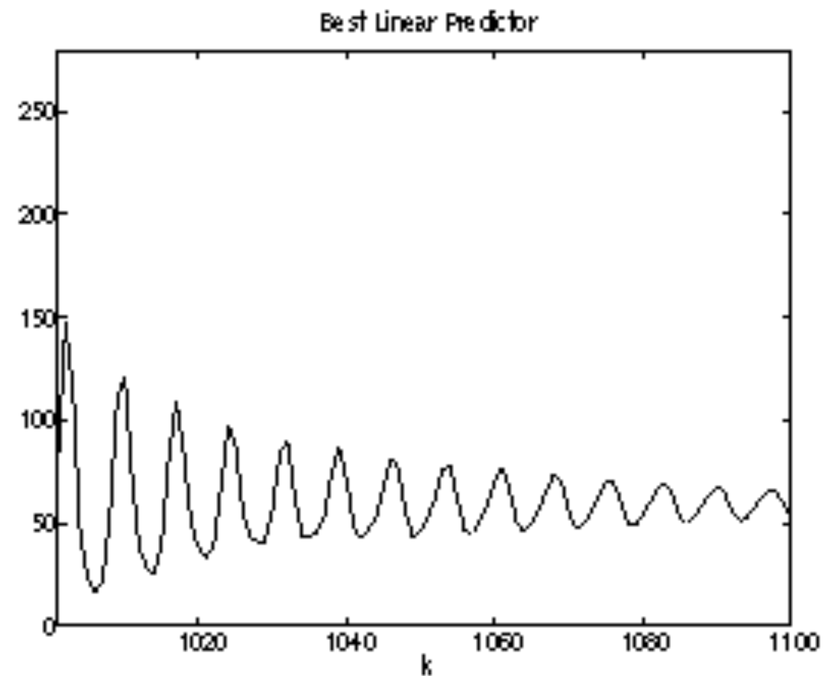


Fig. 3: 1000 points of laser data.

Wan's Prediction Expanded in Sante Fe Institute Competition





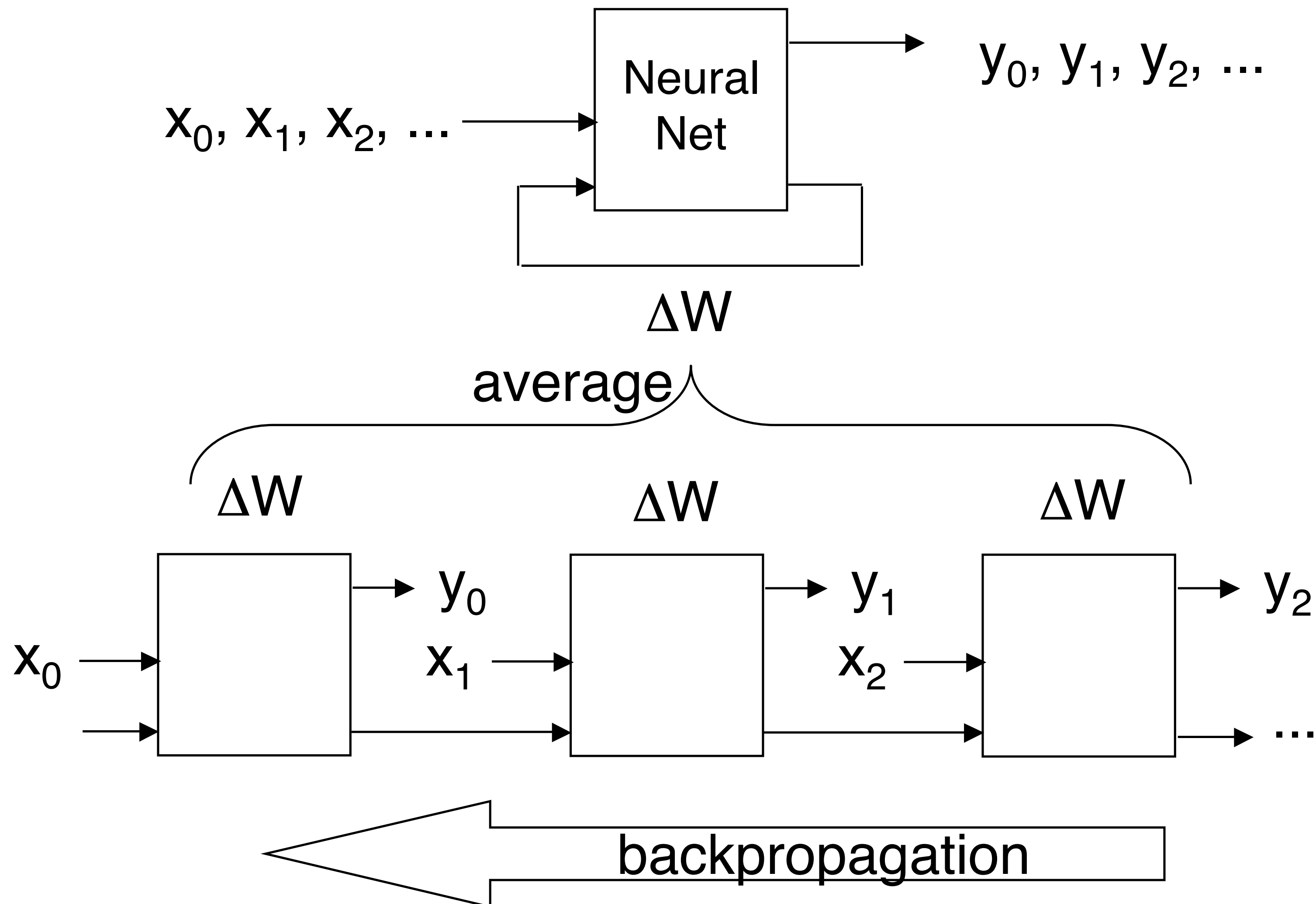
Solid line is actual

dashed line is predicted.

Backpropagation through time (BPTT)

- Unlike TLFF (Time-Lagged Feed-Forward), input samples are not kept in explicit delay lines.
- Instead, input fed sequentially into network (also used in FIR backprop).
- Training is **as if** the network were **unrolled** to accommodate the entire sequence of input samples.
- Only one set of weights is actually used in operation; the weight changes are **averaged** across stages to get the actual weight change

Backpropagation through time (BPTT)



BPTT Application

- The Truck Backer-Upper, D. Nguyen and B. Widrow
- reprinted in Miller, Sutton, and Werbos (eds.), *Neural Networks for Control*, MIT Press, 1990.
- Problem: Back up a truck so that $(x_{trailer}, y_{trailer}) = (x_{dock}, y_{dock})$, given initial values for $(x_{trailer}, y_{trailer}, x_{cab}, y_{cab}, \theta_{trailer}, \theta_{cab})$

Truck-Backer Problem

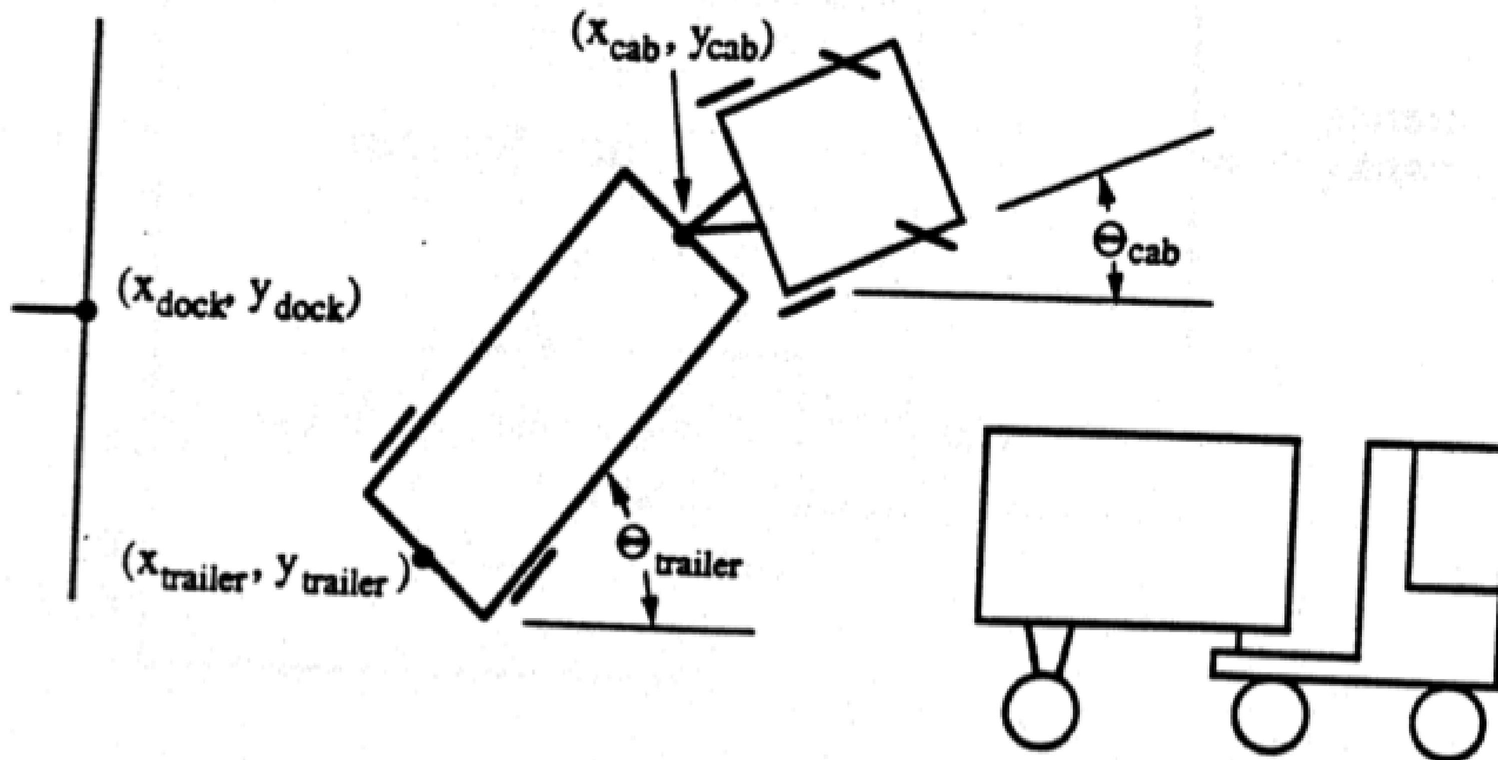


Figure 1: The truck, the trailer, and the loading dock

Training the Truck-Backer

- The truck moves in small time increments Δ
- A neural net is first trained to mimic the truck backing using **real truck dynamics**.
- Given the current state at a time t (which includes the steering angle), the network learns to determine the next state (at time $t + \Delta$)
- This is done by starting the truck in a random state, observing the error between what the network does and the dynamic model, and adjusting the weights.

Kinematics as a Neural Net

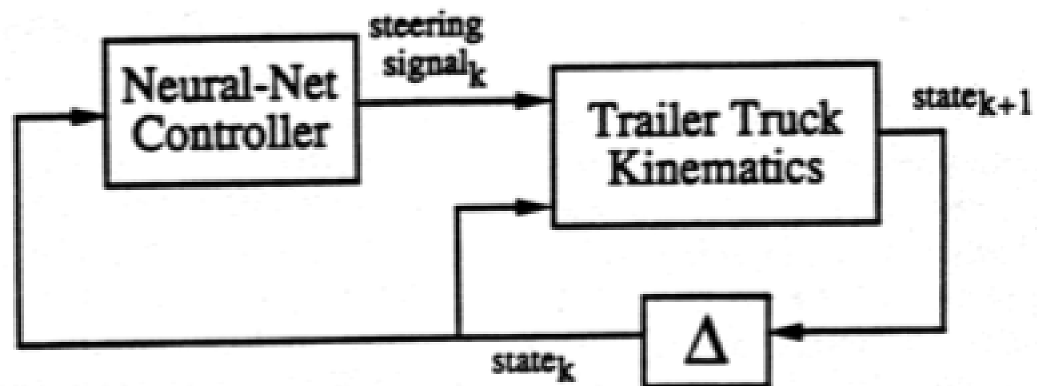


Figure 2: Overview Diagram

Training Kinematics Network

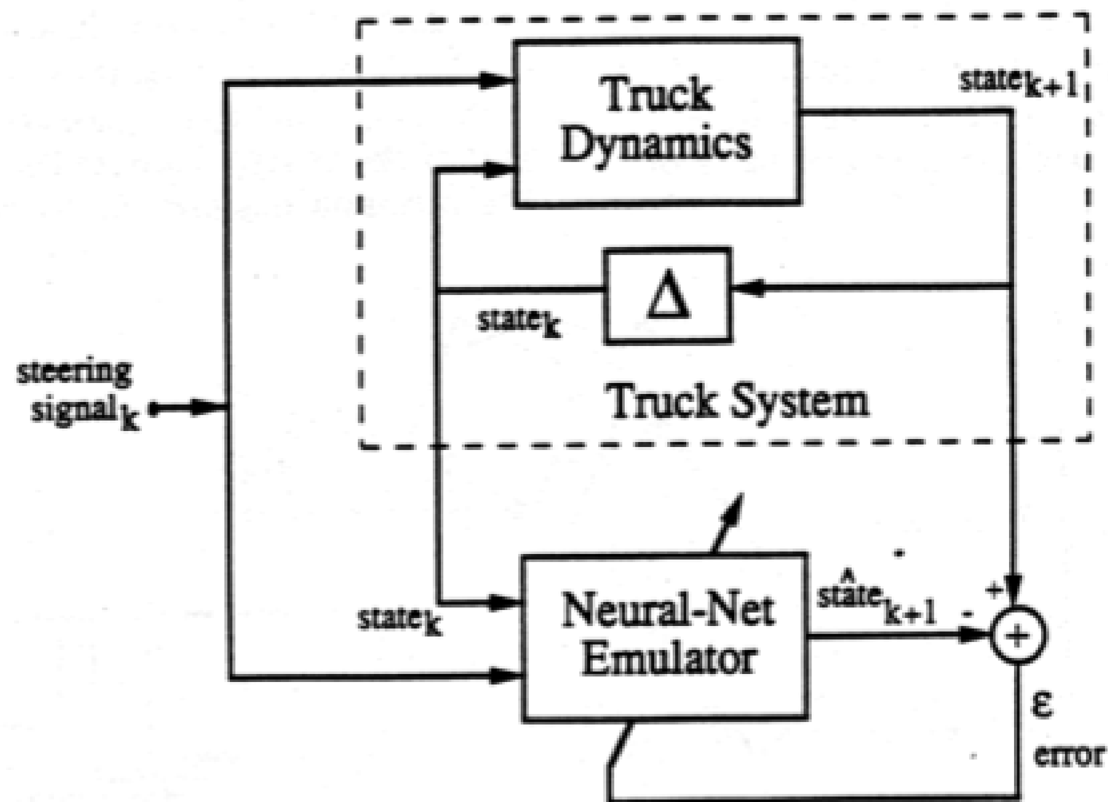
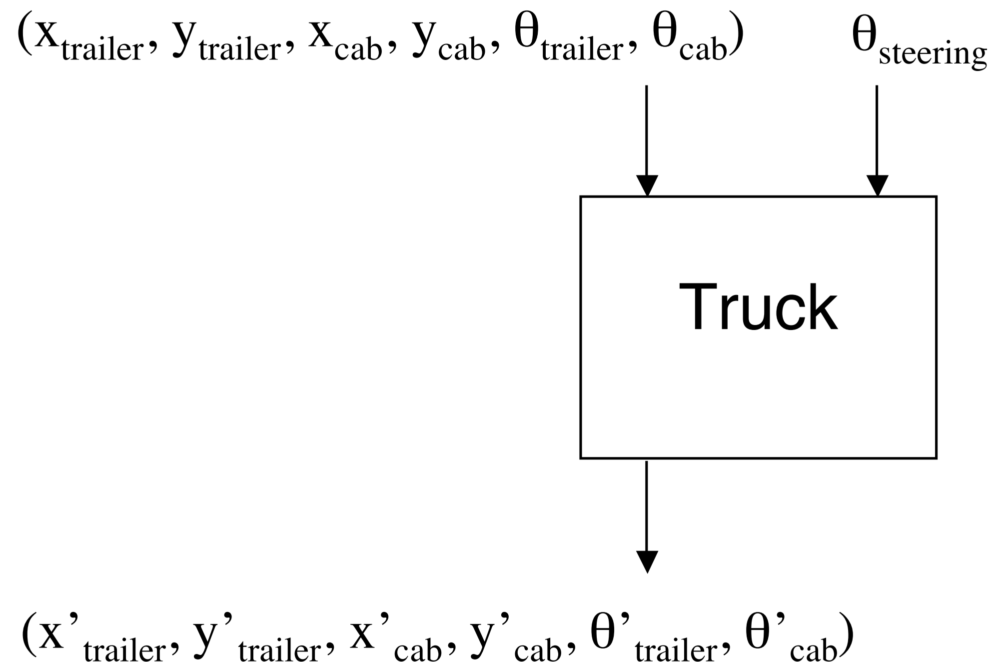
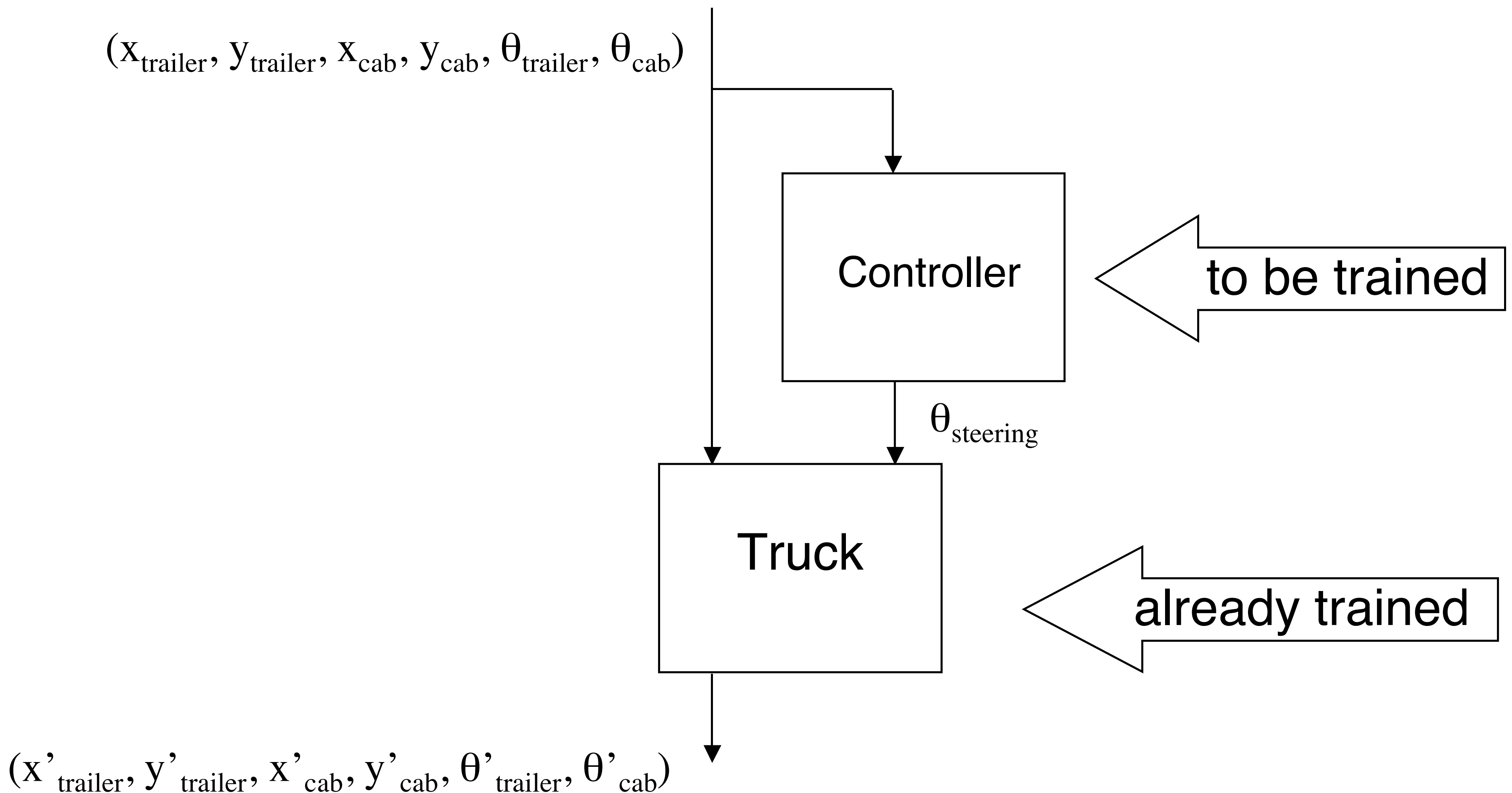


Figure 3: Training the neural-net truck emulator

The function being learned



Truck-Controller Combo

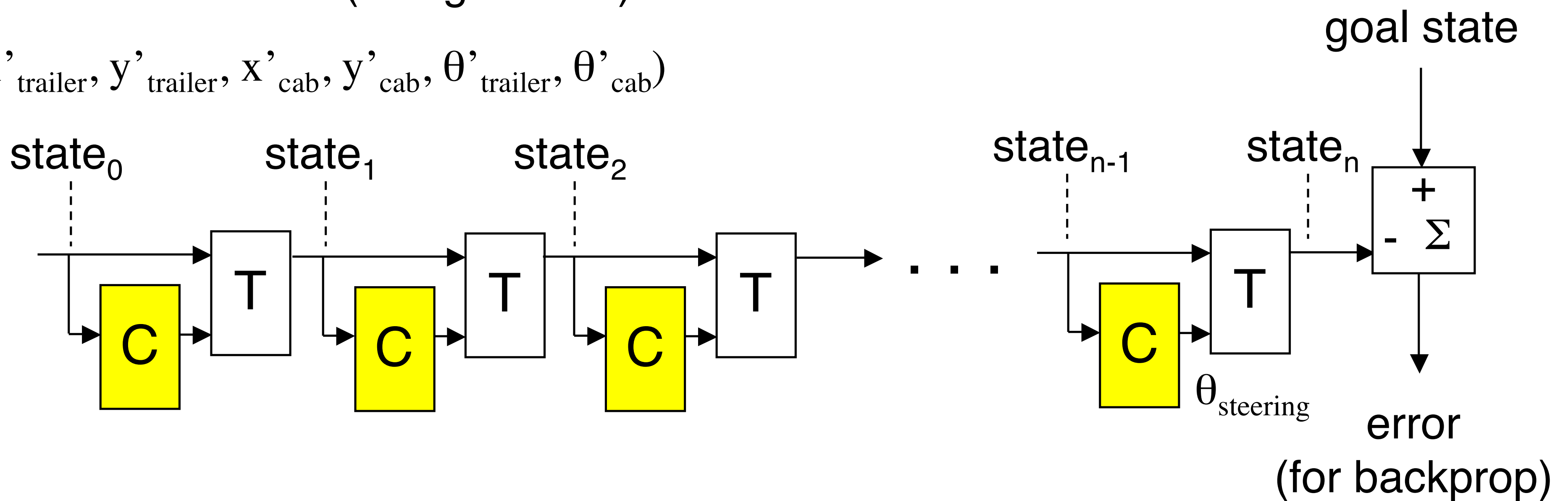


Training the Truck-Backer

- Starting from a random position, the controller backs up the truck one step at a time, until the goal is reached, or an obstacle (such as a side wall) is hit.
- An error value is produced by comparing the desired final state with the goal.
- The error value is backpropagated through the controller-truck combination to adjust the controller's weights, using BPTT.

BPTT for truck training

T = Truck (already trained, weights fixed),
C = Controller (being trained)



Network Statistics

- Truck Emulator:
  - 6-45-6 tansig-tansig network
- Controller
  - 6-25-1 tansig-tansig network

Controller + Emulator

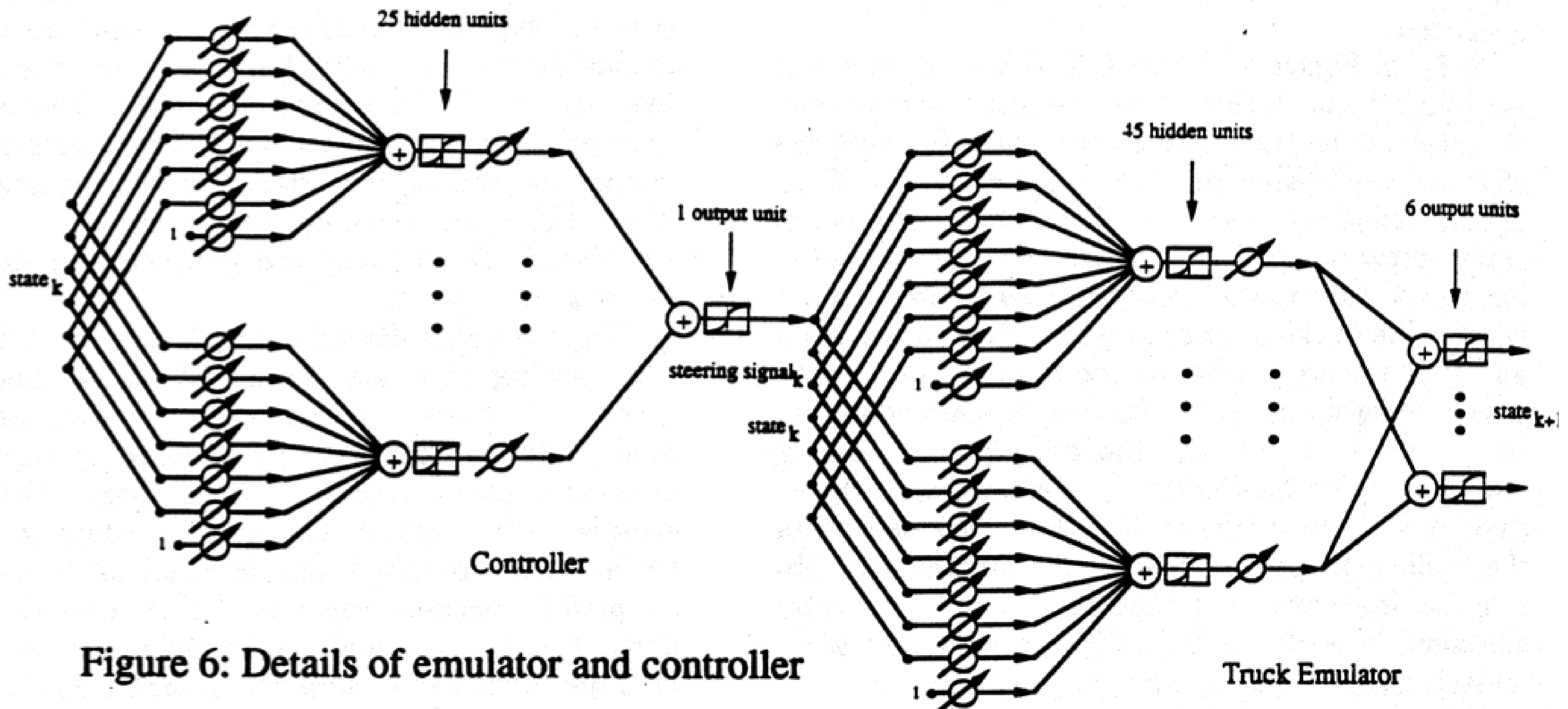


Figure 6: Details of emulator and controller

Training

- 20,000 trials required to train
  - 16 lessons of 1000-2000 each
- Initially truck positioned very close to dock and in a nearly-correct position, so controller could **learn easy tasks first**.
- Final MSE was 3% of truck length, angle 7 degrees

Simulations

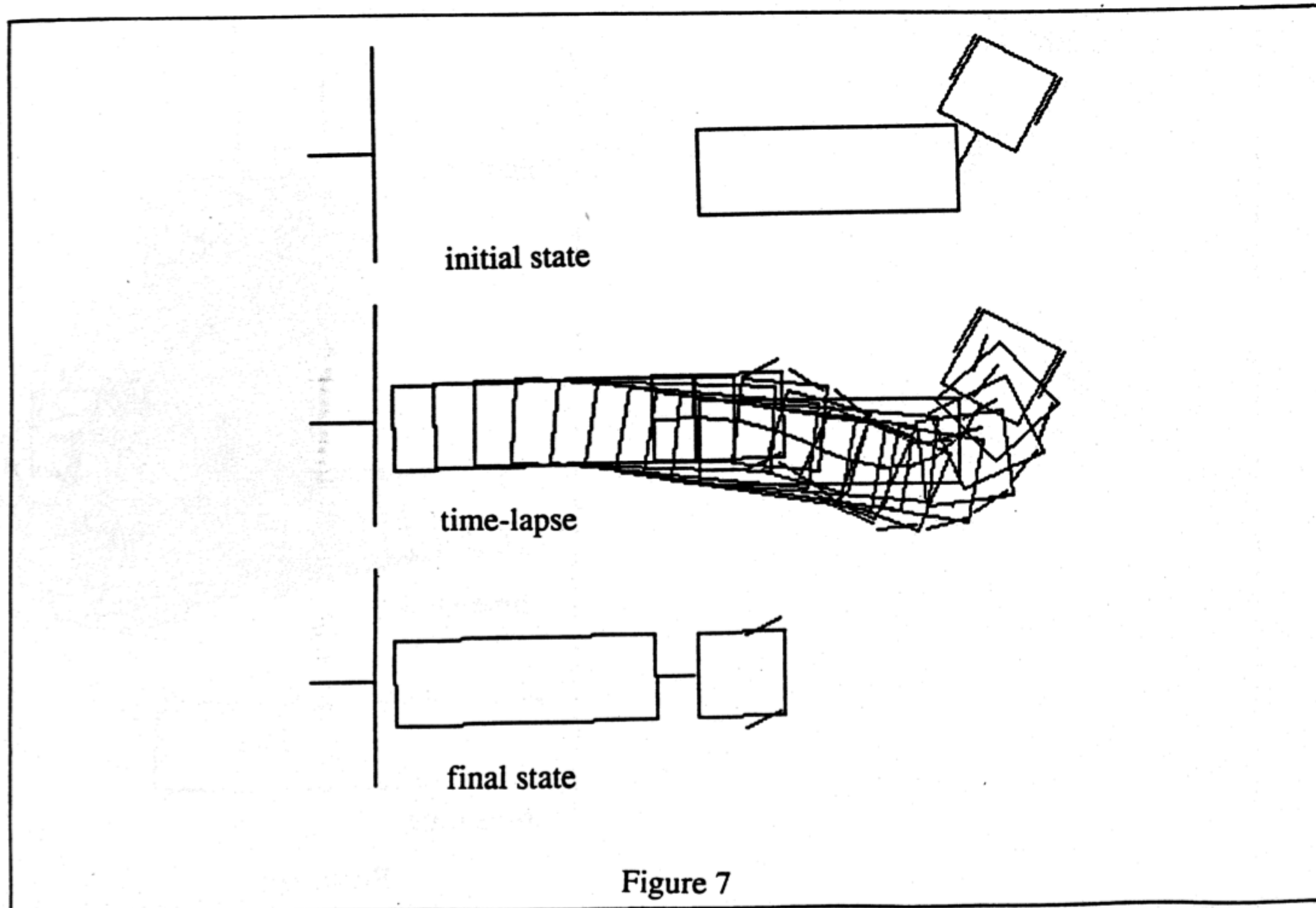
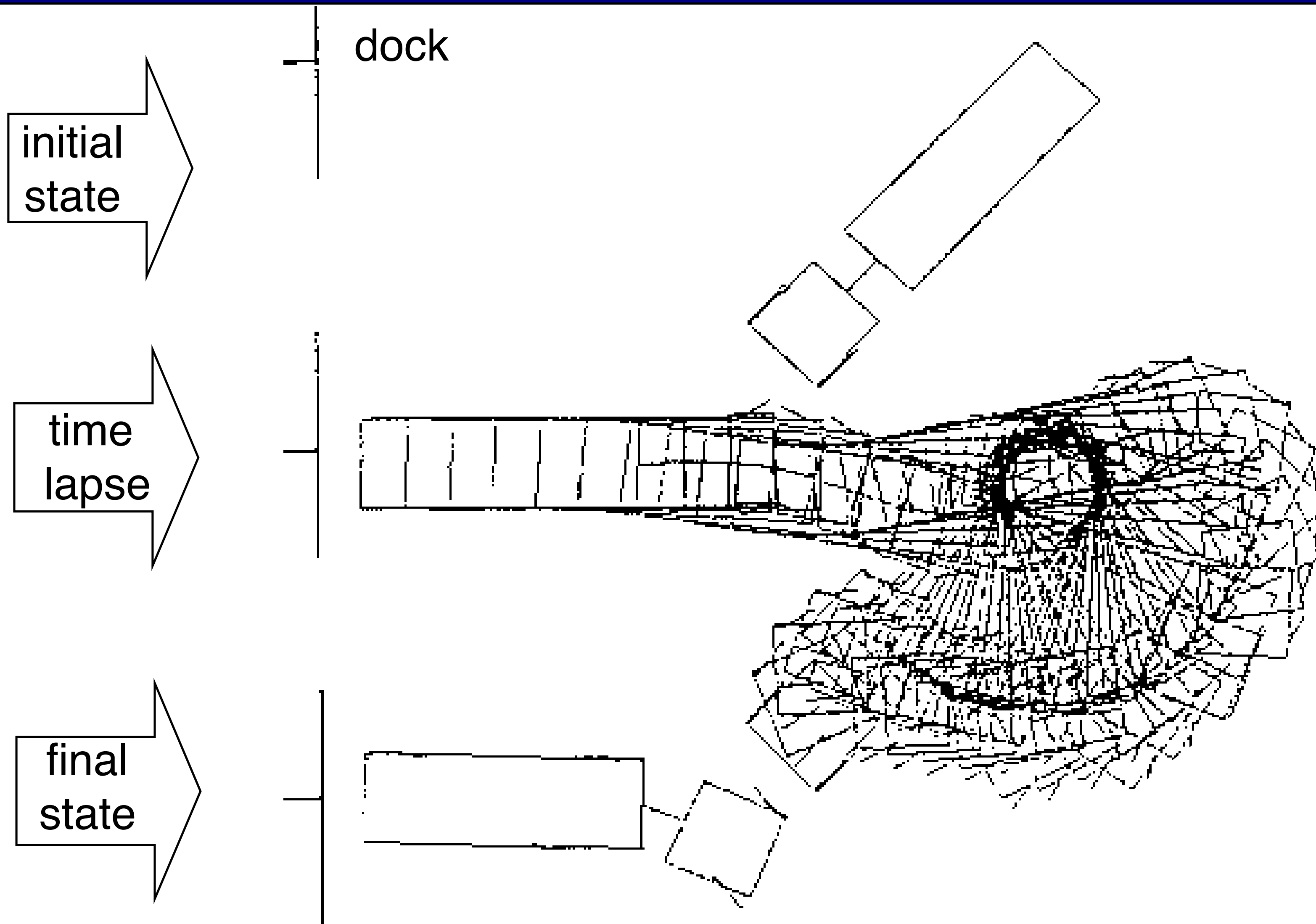
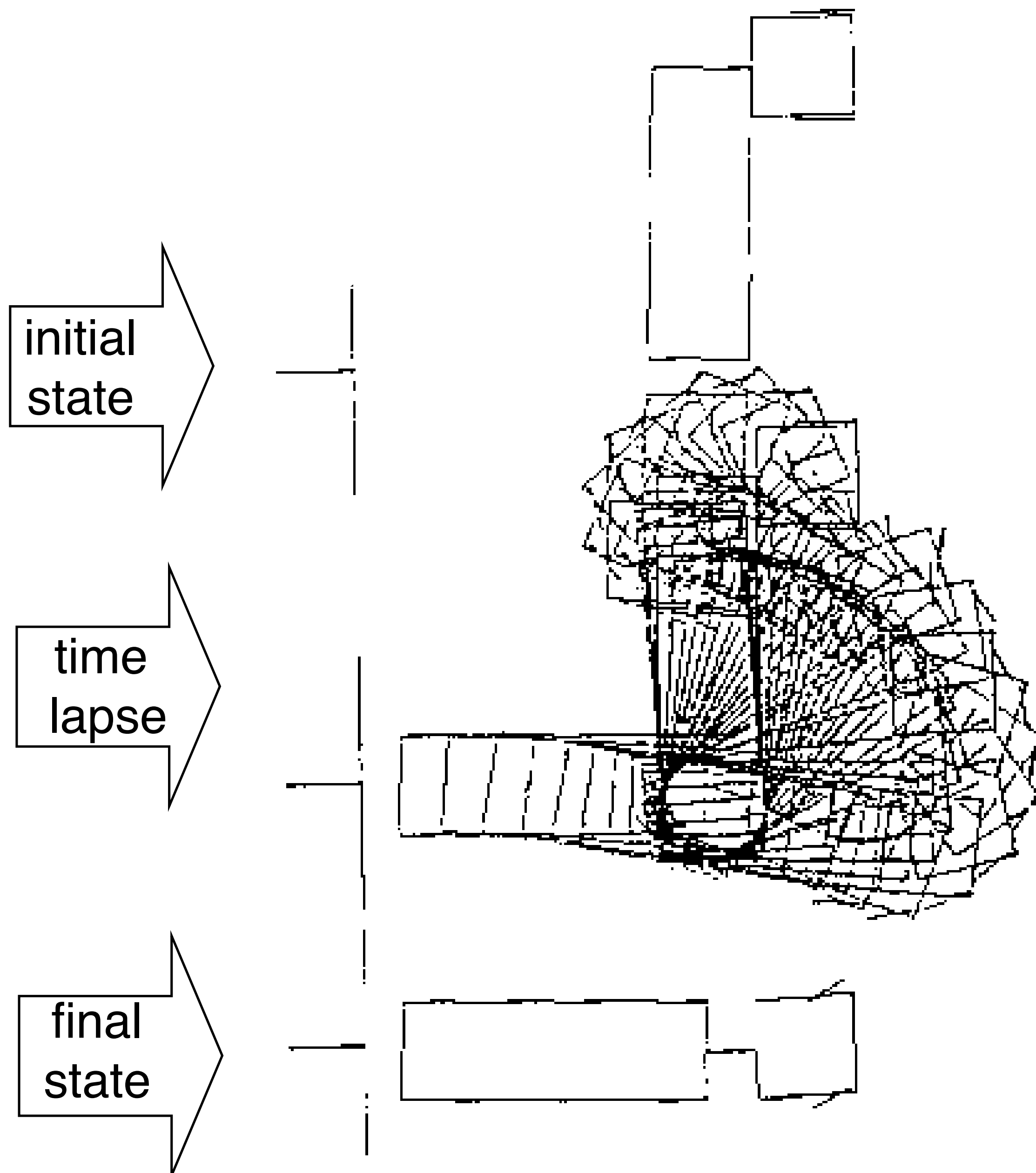


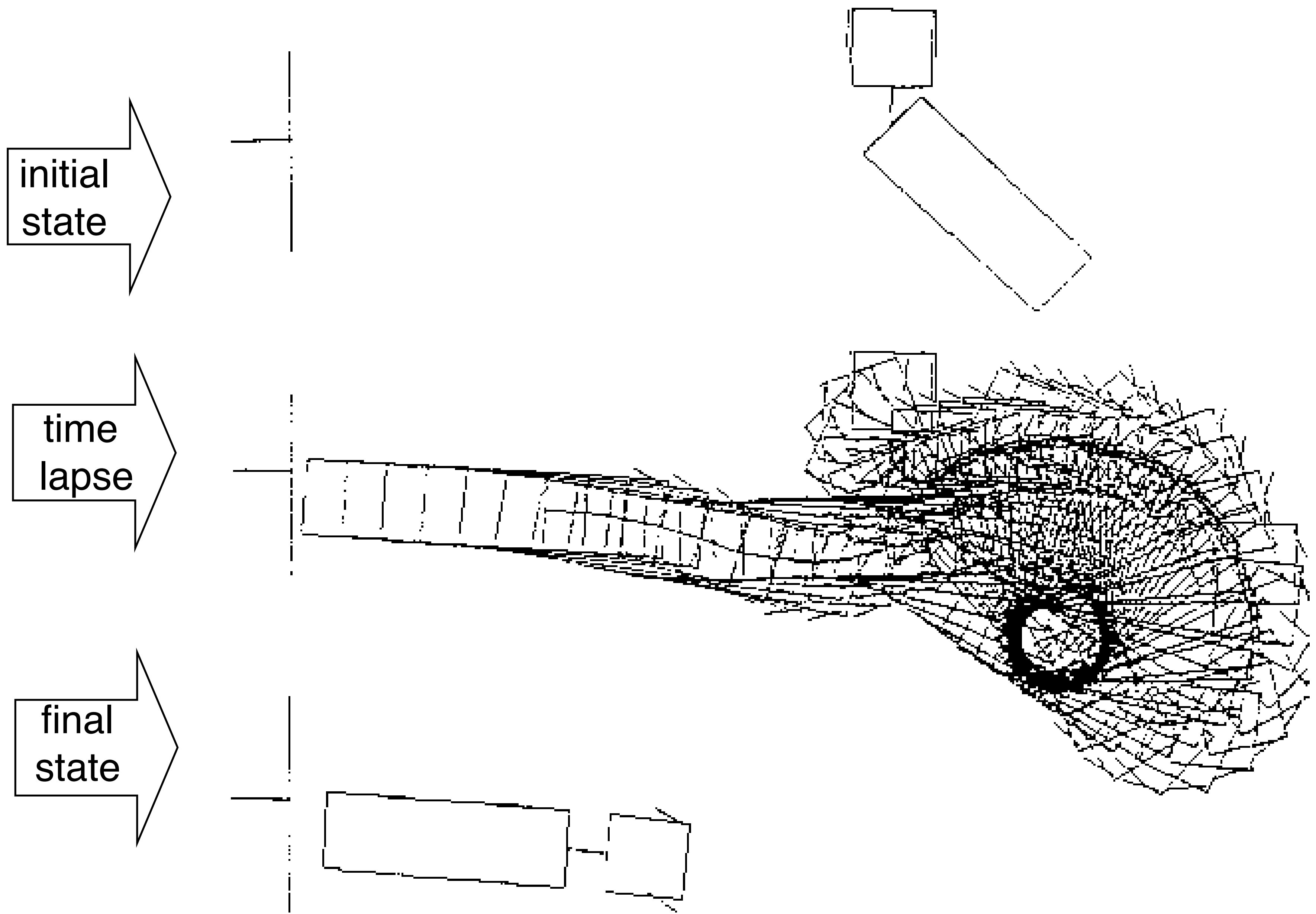
Figure 7

Simulations





initial
state
time
lapse
final
state



initial state
time lapse
final state