A New Technique for Interactive Simulation of Recurrent Neural Networks

by Granino A. Korn, ECE Department, University of Arizona
sites.google.com/site/gatmkorn   *gatmkorn@aol.com*

Abstract

We present new techniques for modeling the feedback loops of recurrent networks, including networks that incorporate tapped delay lines or gamma delay lines. Very fast simplified programs result. Examples of applications include signal prediction and dynamic-model matching. We also suggest interesting future research on improved programs for time-series recognition and classification.

1. Introduction

This article describes much-simplified computer programs for interactive simulation of recurrent neural networks. Sections 2 to 5 briefly review dynamic-system simulation and our open-source software for Windows and Linux. (Korn, 1995, 1998, 2007), We employ a compact, human- and machine-readable vector notation, including very powerful vector index-shift operations for modeling delay lines and filters. The remainder of this report applies these techniques to neural-network simulation.

Section 6 presents a simple backpropagation model representing each neuron layer by a one-line vector assignment. Section 7 then describes a significant innovation: *a technique for programming the time-delayed feedback in recurrent networks without the complication of special context layers*. Sections 8 to 10 next apply our simple vector index-shift notation to neural networks with input and feedback delay lines or gamma delay lines.

Finally, Sections 11 and 12 discuss applications to model matching and time-history prediction and suggest other applications for future research.

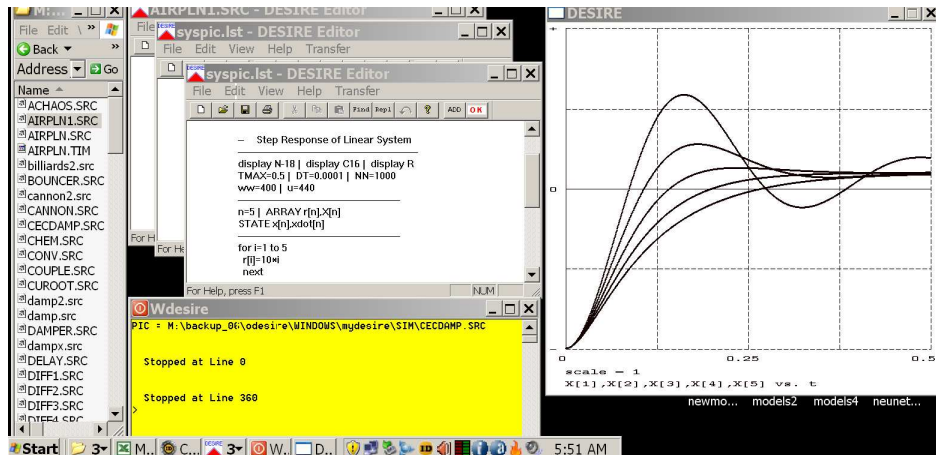## 2. A Simulation Language for Interactive Dynamic-system Modeling

Desire simulation programs (Korn, 1995, 1998) model dynamic systems using a natural mathematical notation for successive difference-equation assignments like

$$x = x + a * sin(c * t)$$
$$y = x \tag{1}$$

and/or differential-equation-system assignments like

$$u = alpha* sin(w * t + beta) + c$$
$$d/dt\ x = xdot$$
$$d/dt\ xdot = - a * x - b * xdot \tag{2}$$

Such model definitions are screen-edited into a ***DYNAMIC program segment*** (Fig. 1). Simulation studies are controlled by typed interactive commands and/or by an ***experiment-protocol script***. Experiment-control commands set or change parameters and initial conditions and then call ***simulation runs*** that produce time-history displays. Each simulation run exercises the model by calling the DYNAMIC program segment for **NN** successive time steps, as in

Figure 1. Desire with a file manager, command window, and three screen-editor windows. Programs in different editor windows can be run in turn to compare models. The original curves were in color.

> **t0 = 0 | t = t0 | NN = 2000 | TMAX = 100**
> **a = - 5.00 | x = 17.1**
> **drun**

| is a statement delimiter. When the experiment protocol encounters the first **drun** statement the DYNAMIC segment is compiled with a fast runtime compiler and runs immediately to produce time-history displays (Fig. 1). More elaborate experiment protocols can call multiple simulation runs with modified parameters and different DYNAMIC segments (Korn, 1998, 2007).

3. Fast, Human- and Machine-readable Vector Operations[1]

Desire experiment-control scripts can declare *vectors* like **x ≡ (x[1], x[2], …, x[n])** and *matrices* like **W ≡ (W[1,1], W[1,2], …, W[n, m])** with single or multiple **ARRAY** statements such as

> **ARRAY x[n], a[m], b[n], c[n], y[m], W[m, n], u[n], v[n], …**

DYNAMIC program segments can then use the vectors and matrices in *vector assignments*, and *vector differential equation* , say

> **Vector x = a + alpha * b * c**
> **Vector y = tanh(W * x)**
> **Vectr d/dt x = beta * cos (t + c)**

which automatically compile into multiple scalar operations

$$x[i] = a[i] + alpha * b[i] * c[i] \qquad (i = 1, 2, …, n)$$

$$y[i] = tanh(\sum_{k=1}^{m} W[i, k] * x[i]) \qquad (i = 1, 2, …, n)$$

$$d/dt\ x[i] = alpha * cos (t + c[i]) \qquad (i = 1, 2, …, n)$$

---

[1] References 1 (due for a new edition) and Reference 2 (out of print) refer to an early version of Desire that lacked the new vectorizing compiler (Korn, 2007) we employ here.

MATRIX assignments similarly compile into multiple assignments to matrix elements **W[I,k]**.[2]. All these compiler operations unroll program loops, so that the resulting binary code is fast.

We can also compute vector-component sums and inner products like

$$p = \sum_{k=1}^{n} u[k]\, v[k] \qquad\qquad p = \sum_{k=1}^{n} u[k]\, v[k]$$

with *inner-product assignments* **DOT p = u * 1** and **DOT p = u * v**, again without program-loop overhead.

Desire vector operations permit very fast vectorized Monte Carlo simulation of engineering and biological systems and can model fuzzy-logic controllers and partial differential equations as well as the neural-networks we shall discuss here (Korn, 2007).

4. Vector Index-shifting, Delay Lines, and Filters

Given an **n**-dimensional vector **x ≡ (x[1], x[2], …, x[n])** and an integer **k**, the *index-shifted* vector **x{k}** is the **n**-dimensional vector **(x[1+k], x[2+k], …, x[n+k])**, with components referring to indices less than 1 or greater than **n** set to 0. Significantly, the assignments

$$\textbf{Vector x = x\{-1\} \quad | \quad x[1] = input} \tag{3}$$

compile into

$$\textbf{x[i] = x[i − 1]} \quad (\textbf{i} = 1, 2, …, \textbf{n}) \qquad\qquad \textbf{x[1] = input}$$
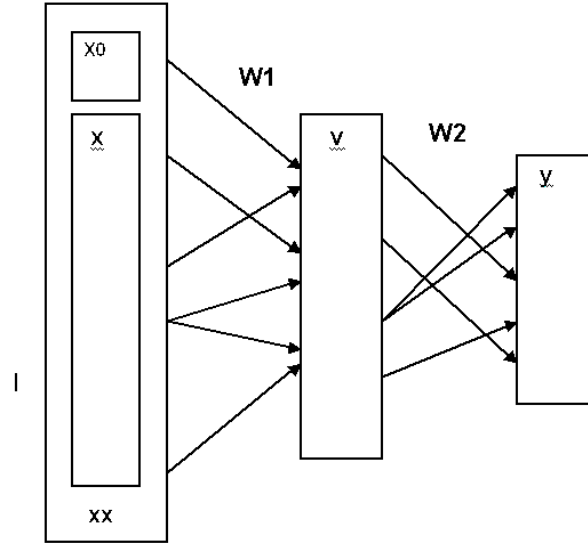
This neatly models shifting successive samples of a function **u(t)** into a *tapped delay line* with tap outputs **x[1] = input, x[2], …, x[n].** Note that the assignment **x[1] = input** overwrites the **Vector** operation's assignment **x[1] = 0** at each step.

Assignments like (3) can, for instance, model a complete **n**th-order digital filter *with only two program lines* (Appendix A). Sections 8 to 10 will describe neural networks incorporating tapped delay lines and also gamma delay lines (Principe, 2000) modeled with a similar index-shift operation.

5. Neural-network Models

DYNAMIC program segments (1) that include differential equations compute state-variable derivatives. An integration routine selected by the experiment-control script then combines derivative values from successive time steps to update differential-equation state variables(Korn, 1998).

Desire can model biological neurons with differential equations (e.g. pulsed integrate-and-fire neurons) (Korn, 2007). but the neural-network models we discuss here are much simpler. For DYNAMIC program segments without differential equations, the simulation time **t** automatically steps through **t = 0, 1, 2, …, NN** by default (users can, if desired, specify different starting times and/or time increments). Neuron activations and connection weights are represented by real numbers that roughly model neuron pulse rates and synapse chemistry. Both are updated with simple difference equations in successive time steps. Appendix B shows how we handle problems that combine differential-equation models and neural networks, as in sampled-data control systems.

Figure 2. A simple backpropagation network.

6. A Simple Backpropagation Network

Figure 2 shows a simple three-layer neural network. Desire's interpreted experiment-protocol script declares the three neuron layers in turn with

ARRAY x[nx] + x0[1] = xx v[nv], y[n]
x0[1] = 1

and two connection-weight matrices **W1** and **W2** with

ARRAY W1[nv, nx + 1], W2[ny, nv]

Desire array declarations like **ARRAY x[nx] + x0[1] = xx** act like Fortran equivalence statements: **xx[3]** is identical with **x[3]**, and **xx[nx + 1]** is identical with **x0[1]**. As is customary, the input layer **xx** adjoins a one-dimensional bias vector **x0** to the normal nx-dimensional network input **x**. With **x0[1]** set to 1, we can then conveniently represent input biases as **nv** extra connection weights **W1[i, 1].**

The runtime-compiled DYNAMIC program segment defines the network dynamics with

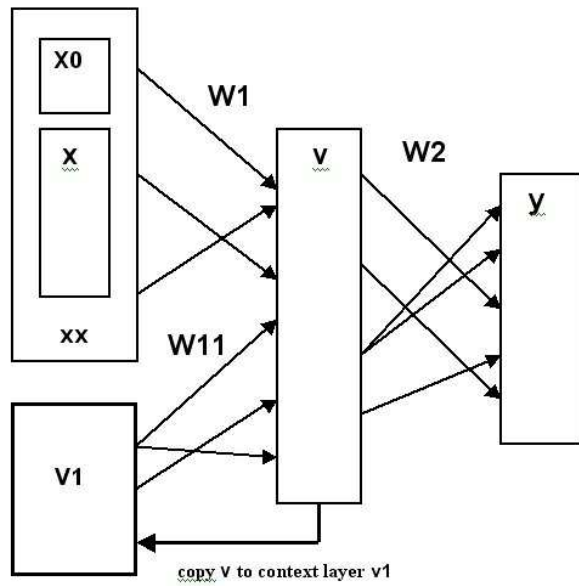**Vector v = tanh(W1 * xx)**
Vector y = W2 * v

if we use a tanh activation function for the nonlinear hidden layer. To produce simple backpropagation updating, we declare **target**, **error**, and error-propagation vectors with
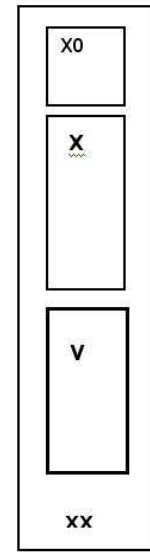
ARRAY target[ny], error[ny], vdelta[nv]

and program

Vector error = target – y
Vector vdelta = W2% * error * (1 – v^2)
DELTA W1 = lrate1 * vdelta * xx
DELTA W2 = lrate2 * error * v   (4)

Figure 3. A simple Elman recurrent network.   **Figure 4.** Modified input layer .

Here **W2%** denotes the transpose of the connection-weight matrix **W2**, and

DELTA W = *matrix expression*   is equivalent to   **MATRIX W = W +** *matrix expression*

These assignments update vectors and matrices with data computed earlier, starting with given initial values. Desire **ARRAY** declarations initialize all subscripted variables to the default value zero. That is fine for the vectors; but the experiment-protocol script must initialize the connection weights **W1[i, k]** and **W2[i, k]** with small random values.

In addition to declaring and initializing neuron-layer arrays, the experiment-protocol script for a neural-network experiment must set parameters and initial values of scalar state variables (if any) and then schedule training and test simulation runs with **drun** statements.. The script also selects integration rules (if any) and the display scale and colors. For simplicity, our text omits these housekeeping operations.

7. Simplified Recurrent-network Programming

An Elman recurrent network (Fig. 3) (Principe, 2000, Elman, 1990) copies all or some of the hidden network layer **v** to a *context layer* **v1** that is fed back to **v** together with the input **xx**. The experiment-protocol script declares the original 3 neuron layers **xx**, **v**, and **y** and the connection weight matrices **W1** and **W2** as before,

    ARRAY x[nx] + x[0[1] = xx, v[nv], y[ny], W1[nv, nx + 1], W2[ny, nv]
    x0[1] = 1

and adds the context layer **v1** and a new connection-weight matrix **W11**:

    ARRAY v1[nv], W11[nv, nv].

The network dynamics in the DYNAMIC program segment become

> **Vector v1 = v**
> **Vector v = tanh(W1 * xx) + tanh(W11 * v1)**
> **Vector y = W2 * v**   (5)

To update **W11** as well as **W1** and **W2** by backpropagation now requires two error-propagation vectors **v1delta** and **v2delta**, and the updating program becomes more complicated. But *there is a much better way!*

Just as we concatenated the input layer **x** and its bias layer **x0**, we can declare a single new input layer **xx** that combines our hidden layer **v** with **x** and **x0**:

> **ARRAY x[nx] + x0[1] + v[nv] = xx | x0[1] = 1**   (6)

(Fig. 4). The two connection-weight matrices **W1** and **W11** of the Elman network in Fig. 3 can now be replaced with a single connection-weight matrix **W1**,

> **ARRAY W1[nv, nx + 1 + nv]**

W1 feeds **xx** to the hidden layer **v** just as in Fig. 2 - but **xx** now includes the hidden-layer acti-vations **v** computed in the preceding iteration. *The simple backpropagation-updating assign-ments (4) for the static network of Fig. 2 then work without change for the recurrent neural network* in Fig. 3. Only the array dimensions have changed.

It is just as easy to implement time-delayed feedback from the output layer **y** (Jordan recurrent network), or from both **v** and **y**. Backpropagation updating remains *exactly the same*. This simplified implementation of recurrent-network feedback is by no means restricted to back-propagation networks. This technique serves equally well for two-layer linear and nonlinear net-works, for softmax pattern recognizers, and for radial-basis-function networks, which are all easy to program in the Desire language (Korn, 2007). In each case we simply reuse the unchanged program for a static neural network.

8. Networks with Input Delay Lines

The earliest neural network with time-history memory was Widrow's adaptive filter (Principe, 2000) In Fig. 5, successive values of a single time-series **input** enter a delay line whose taps feed a static neural network trained to filter, recognize, or predict time-series patterns. Desire's compact index-shift operation (3) is exactly what is needed for modeling such networks.

Widrow's original network, for example, combined a delay line with a simple linear network layer

> **Vector x = x{-1} | x[1] = input**
> **Vector y = W * x**

Widrow's network had a single output **y[1]** and thus implemented a linear filter that could be trained with his new LMS algorithm to match a target time series. In our notation this succes-sive-approximation rule would be

> **DELTA W = lrate * (target – y) * x**

Improved designs incorporate a nonlinear multilayer network, say the backpropagation network of Sec. 6:

> **Vector x = x{-1} | x[1] = input**
> **Vector v = tanh(W1 * x)**
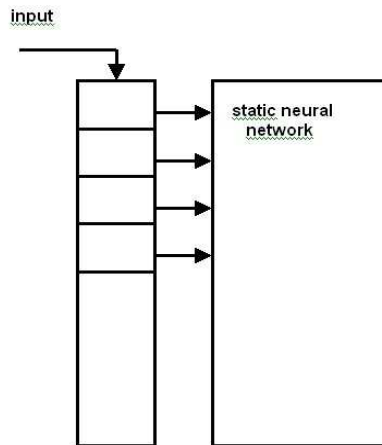> **Vector y = W2 * v**

Figure 5. A static neural network
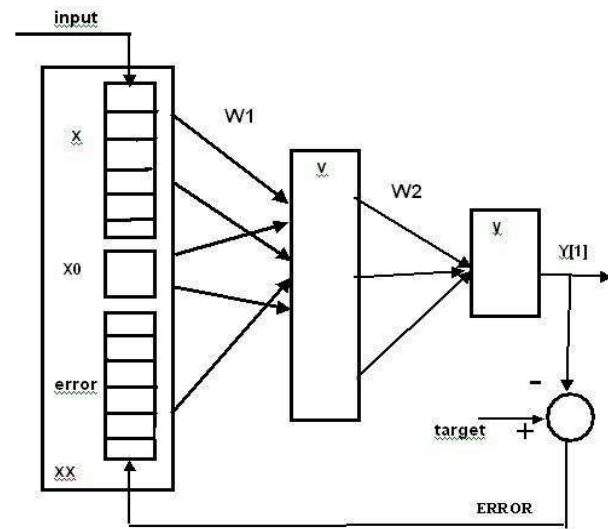   fed by an input delay line.

Figure 6. A NARMAX network.

or other types of static networks. All need only ordinary static-network training.

9. NARMAX[2] Networks use Delay-line Feedback

The recurrent network in Fig. 6 has a single input **input** to a delay-line layer **x** of length **nx** as before. The output layer **y** has only a single output. **y[1]**. The (scalar) error in this network output is

ERROR = target - y[1]

where **target** is a desired output time series. Successively delayed samples of **ERROR** are produced by a second delay-line layer **error** of length **ne**. The delayed error samples are fed back to the neural network.

Referring to Fig. 6, we again concatenate all input-layer vectors, in this case the two delay lines **x** and **error** and the bias **x0**, into a single input layer **xx**:

ARRAY x[nx] + x0[1] + error[ne] = xx

xx feeds the hidden layer **v** of an ordinary backpropagation network.

The DYNAMIC program segment models each delay line with the vector index-shift operations introduced in Sec. 4. The complete NARMAX network is thus programmed with

```
ARRAY x1[nx] + x0[1] + error1[ne] = xx  |  x0[1] = 1
ARRAY v[nv], y[1], error[ne], vdelta[nv]
ARRAY W1[nv, nx + ne + 1], W2[1, nv]
    . . . . . . . . . . . . . . .
DYNAMIC
Vector x1 = x1{-1}   |   x1[1] = input          input delay l;ine
```
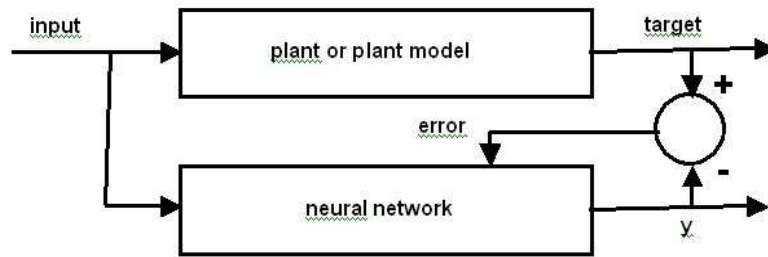
---

2 NARMAX stands for Nonlinear Auto-Regressive Moving Average with eXogenous inputs.

Figure 7. Matching a neural network to a plant or plant model. **input**, **target**, **y**, and **error** can be scalars or vector functions of the time **t**.

Vector v = tanh(W1 * xx)	hidden layer
Vector y = W2 * v	output layer
ERROR = target - y	output error
Vector error = error{1} | error[1] = ERROR	feedback delay line
Vector vdelta=W2%*error*(1-v^2)	backpropagation
DELTA W1= lrate1 * vdelta * xx	
DELTA W2= late2* error * v	

Programmers must specify the **input** and **target** time series for different applications.

Once again *the backpropagation program is exactly the same as in Sec. 6.* One can also substitute different types of neural networks for the backpropagation layers in Fig. 6.

10. Networks with Gamma Delay Lines

A simple tapped delay line of length n "remembers" its input for only **n** time steps. Principe's *gamma delay line* (Principe, 2000) replaces each delay-line element with a simple first-order filter. That effectively gives neural-network input and feedback delay lines a much longer memory, so that the networks tend to perform better or use fewer neurons. Our vector index-shift notation models a gamma delay line with

Vector x = x + beta * (x{-1} - x)   \| x[1] = input

which automatically compiles into

x[i] = x[i] + beta * (x[i – 1] – x[i])   (i = 1, 2, …, n)   x[1] = input

beta is a scalar filter parameter set by the experiment-protocol script; we have compactly programmed n difference equations for **n** identical first-order filters.[3]   We normally prefer such gamma delay lines for NARMAX networks.
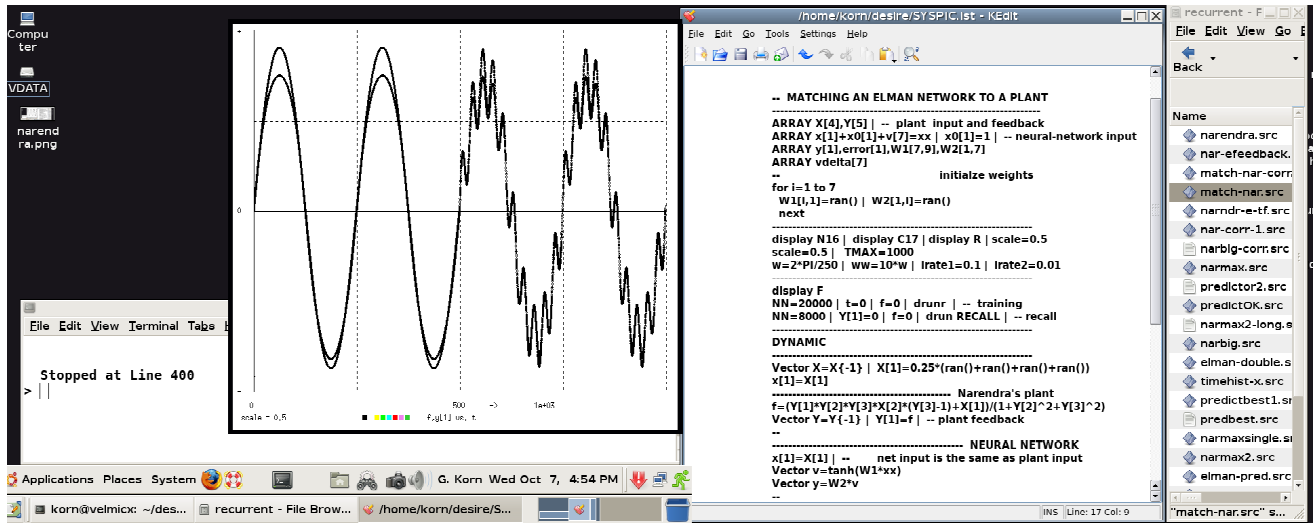
11. Applications

The most common applications of recurrent networks are
- model matching (e.g. plant models for control-system design)
- time-series prediction
- recognition or classification of time-series patterns

---

[3] It is convenient to program **Vector x = x + beta * (x{-1} - x)** as **Vectr delta x = mu*(x{-1} - x)** .

Figure 8. Dual-screen display showing our Elman network matching Narendra's plant model (6). Both plant and network were fed Narendra's test function (7). The graphs of **target = f** and **y** essentially reproduce the results Narendra obtained with his own 4-layer NARNAX network (Narendra, 1990). The original graphs were in color. The complete program is included in the open-source Desire distribution package.

Figure 8 demonstrates a model-matching experiment. The program can be screen-edited and rerun immediately for truly interactive modeling. We programmed Elman networks with 2 and 3 hidden layers and a NARMAX network to match one of Narendra's difference-equation plant models (Narendra, 1990) described by

f = [Y(k)*Y(k-1)*Y(k-2)*input(k-1)*(Y(k-2)-1)+input(k)]/[1+Y(k-1)^2 + Y(k-2)^2)]
target(k) = Y(k)

$$(k = 0, 1, 2, \ldots) \qquad (6)$$

The networks were trained with random-noise input and tested with Narendra's test function.

s = 0.5 * ((1 - 0.2 * swtch(t - 500)) * sin(w * t) + 0.2 * swtch(t - 500) * sin(ww * t)) (7)

Training typically converged in 8 out of 10 simulation runs. All three recurrent networks then matched the plant equally well (Fig. 8).
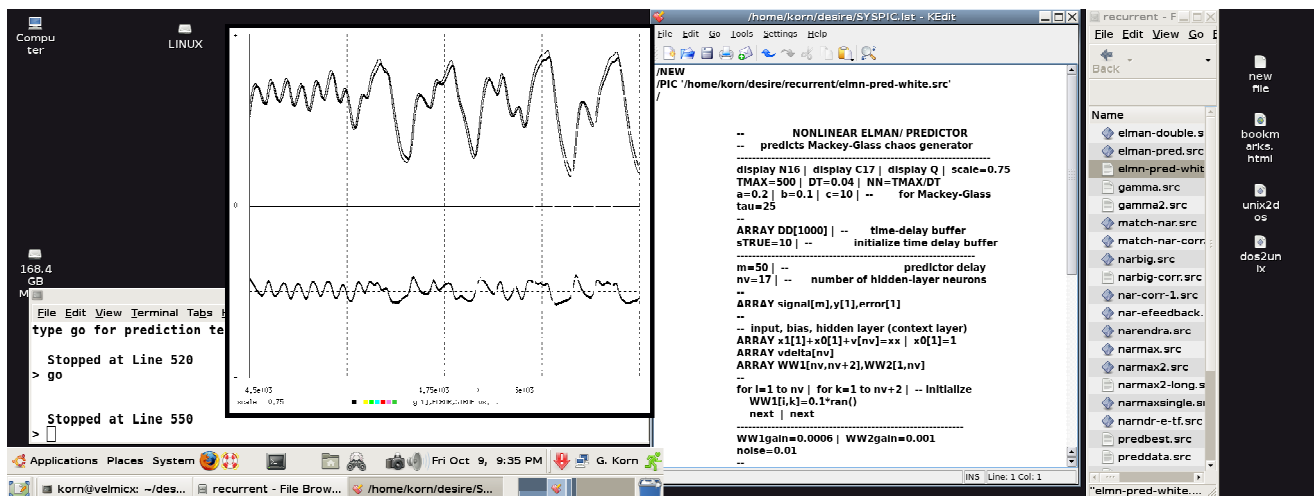
For modeling a predictor the "present" neural-network input is a delayed version of a specified "future" time series **target**:

ARRAY buffer[m]
Vector buffer = buffer{-1} | buffer[1] = target | input = buffer[m]

The neural network output **y** is then trained to match **target**. We programmed a textbook problem (Principe, 2000) predicting the chaotic Lorenz (Korn, 1998) and Mackey-Glass (Principe, 2000) time series.[4] Our Elman and

---

[4] Desire models Mackey-Glass with only two program lines

> **tdelay S=D(signal, tau**
> **d/dt signal = a * S/(1 + Sd^c) – b * signal**

Figure 9. This display shows **target**, **y**, and **ERROR** for an Elman network predicting the Mackey-Glass chaotic time series. The original graphs were in color.

NARMAX networks predicted this time series within a few percent for 50 time steps ahead (Fig. 9). As expected, gamma delay lines worked better than simple delay lines of the same length. Prediction was still successful when we removed the feedback delay line from the NARMAX network, resulting in the simpler model of Fig. 5. Readers interested in the details of these studies – or in repeating our experiments – will find the compact Desire programs for 20 model-matching and prediction experiments included in the open-source Desire distribution file.

12. Conclusions and Future Research

The essential contribution of this article is the novel application of the Desire language's array declaration (6) in Sec. 7. Acting much like a Fortran equivalence statement, ***this programming trick effectivey eliminates entire neuron layers and greatly simplifies recurrent-network updating algorithms.*** The resulting neural-network models are smaller, run faster, and are easier to understand.

On a 3.15 GHz 2-CPU Penryn-class personal computer, the screen-edited, runtime-compiled programs exhibited in this report all compiled and produced time-history displays within 25 msec. This compilation delay is not noticeable, so that truly interactive modeling is possible. The recurrent-network programs in Figs. 8 and 9 converged within 1 to 3 seconds.
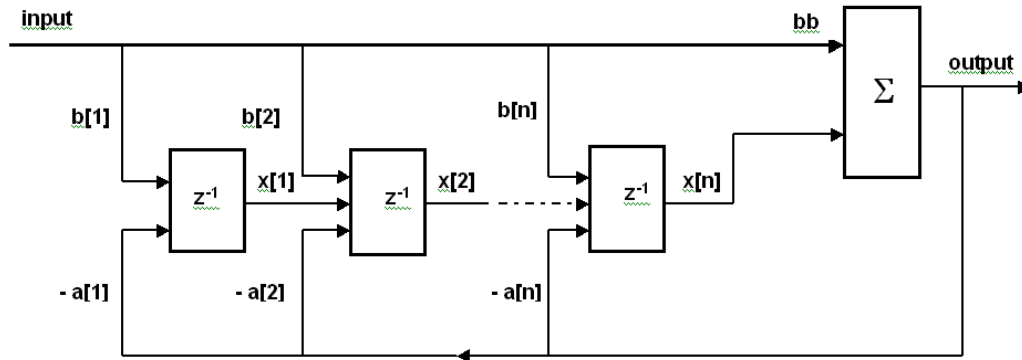
We demonstrated simple applications to Elman, Widrow, and NARMAX networks to model matching and time-series prediction. Time-series pattern recognition (pattern classify-cation) will be the first interesting topic for future work. Neuron layers implementing various softmax classifiers (Korn, 2007). will replace the backpropagation network in Figs. 4 and 6The required training procedure is again simply that for a static network.

Our new trick of concatenating neuron-layer arrays can work equally well in other computer languages. But Desire's combination of an interpreted experiment protocol and fast
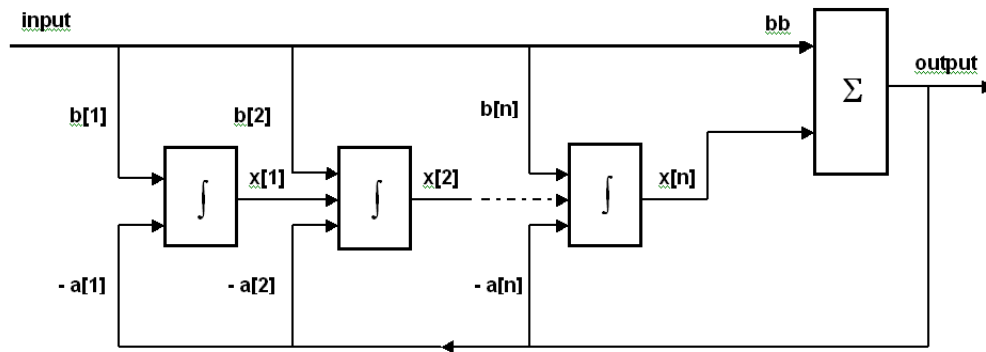
---

where **tdelay** is a time-delay operator, and **a**, **b**, and **tau** are specified constants.

runtime-compiled simulation runs makes interactive modeling – which can involve hundreds of program changes in one day – especially convenient.

Fig. A-1. An **n**th-order digital filter with the z transfer function

$$H(z) = \{bb\ z^n + b[n]z^{n-1} + b[n-1]z^{n-2} + \ldots + b[1]\}/\{z^n + a[n]z^{n-1} + a[n-1]z^{n-2} + \ldots + a[1]\}$$

Fig. A-2. An **n**th-order analog filter with the transfer function

$$H(s) = \{bb\ s^n + b[n]s^{n-1} + b[n-1]s^{n-2} + \ldots + b[1]\}/\{s^n + a[n]s^{n-1} + a[n-1]s^{n-2} + \ldots + a[1]\}$$

APPENDIX A: MODELING FILTERS WITH VECTOR-SHIFT OPERATIONS (Korn, 2009)

The following examples further show the power of our index-shift operations. Given a state vector **x[n]** and coefficient vectors **a[n]**, **b[n]** declared and filled in the experiment protocol, *only 2 assignments*

> **output = x[n] + bb * input**
> **Vector x = x{-1} + b * input – a * output**

model a general digital filter of any order **n** (Fig. A-1). Similarly, *the two assignments*

> **output = x[n] + bb * input**
> **Vectr d/dt x = x{-1} + b * input – a * output**

model a general analog filter of any order n (Fig. A-2).

Appendix B: Models Combining Differential Equations and Neural Networks

When a DYNAMIC program segment contains scalar or vector differential equations (statements starting with **d/dt** or **Vectr d/dt**) the starting value **t0** of the simulation time **t** defaults **t0 = 0** instead of **t0 = 1**, and time histories are sampled at **NN** *communication points*

   t = 0, COMINT, 2 COMINT … , (NN - 1)COMINT = TMAX   COMINT = TMAX/(NN - 1)

Desire returns an error message is the selected number **NN** of sampling points makes **COMINT** smaller that the specified integration step **DT**.

Desire programs combining differential-equation models with sampled-data systems like neural networks execute program lines following an **OUT** statement only at the sampling points.] Note that sampled data returned to a differential equation are "sample-hold" state variables" starting from specified initial values.[5] Initial values of subscripted variables (and thus **ARRAY** elements in neural-network programs) default to zero. Multirate sampling is also possible (Korn, 2007).

References

Korn, G.A. (1995). *Neural Networks and Fuzzy-logic Control on Personal Computers and Workstations*, Cambridge, MA, MIT Press.

Korn, G.A. (1998). *Interactive Dynamic-system Simulation under Windows*, London, Gordon and Breach.

Korn, G.A. (2007). *Advanced Dynamic-system Simulation: Model-replication Techniques and Monte Carlo Simulation*, Hoboken, NJ, Wiley.

Korn, G.A. (2008). Fast Simulation of Digital and Analog Filters Using Vectorized State Equations, *Simulation News Europe*, 18,1, April 2008 issue..

Principe, J., et al. (2000). *Neural and Adaptive Systems*, Wiley, Hoboken, NJ,

Elman, J.L. (1990). Finding Structure in Time, *Cognitive Science*, 14, 179-211.

Narendra, K.S., and K. Parthasarathy (1990). Identification and Control of Dynamic Systems Using Neural Networks, *IEEE Trans. on Neural Networks*, 1, 4-27.

---

5 This is necessarily true for *every* dynamic-system simulation program capable of handling sampled-data systems. But most reference manuals ignore this problem...