Behavior Research Methods, Instruments, & Computers
1997, 29 (2), 194-199

AI AND STATISTICAL APPLICATIONS

Chaired by Paula Goolkasian, *University of North Carolina, Charlotte*

Mathematica: A flexible design environment for neural networks

SEAN C. DUNCAN and RYAN D. TWENEY
Bowling Green State University, Bowling Green, Ohio

Several neural networks were developed in *Mathematica* in order to explore the role of "spiky" neurons in neural network memory simulations. Using *Mathematica* for this task confirmed its value as a powerful tool for neural network development: It exhibited distinct advantages over other environments in programming ease, flexibility of data structures, and the graphical assessment of network performance.

Neural networks have received an enormous amount of attention from psychologists in recent years. Rumelhart and McClelland (1986) first showed that powerful techniques for training networks, combined with the increasing speed and memory of modern computers, permitted theoretical models to be developed that exceeded by orders of magnitude the complexity possible even a few years earlier. At present, a large number of software and freeware packages that can quickly and efficiently support a variety of network configurations are available to psychologists. Nevertheless, it is still a major challenge to develop and use any nonstandard network architecture; such development generally demands facility in a powerful computing language such as C or C++ (e.g., Masters, 1993). In the present paper, we argue the merits of using *Mathematica* (Wolfram, 1994), a high-level interpreted language, for the development of novel neural network architectures.

Ratcliff (1994) indicated that use of *Mathematica* can be advantageous for modeling purposes, even given its computational slowness relative to compiled languages. In the particular instance of neural networks, even though a compiled network will always be faster than an interpreted network, the ease with which one can modify simulations in some interpreted language programming environments makes them useful in the construction of neural networks. In the present paper, we thus extend Ratcliff's point by using *Mathematica* for a series of simulations based on a nonstandard network design.

Mathematica is a symbolic programming environment especially suitable for applications that involve the processing of mathematical expressions. It can evaluate symbolic expressions, and it has powerful, high-precision, computational capabilities and flexible graphics routines. Developed and used primarily by and for physical scientists, mathematicians, and engineers, it is relatively underused by psychologists, even as it goes into its third—greatly enhanced—version (Wolfram, 1996). Finally, although developed originally on a Macintosh platform, it is available on a large variety of platforms, and it can be obtained for any of a variety of operating systems, including Windows and UNIX.

Mathematica has had some exposure in the behavioral sciences. Gronlund, Sheu, and Ratcliff (1990) showed that it was useful in the modeling of global familiarity models. More recently, Lorig and Urbach (1995) utilized it for the analysis of event-related potentials. In these cases, *Mathematica* was shown to be quite useful for modeling psychological phenomena. In the present paper, this utility is extended further.

Freeman (1994) wrote a textbook introduction to neural networks that included a series of useful neural network routines in *Mathematica*. Freeman chose *Mathematica* as the means of presentation because of the clarity with which its expressions can be related to the more abstract equations that generally define neural networks. In using Freeman's text in a graduate psychological modeling course, we confirmed his point: Relatively straightforward code is provided by Freeman for standard neural net architec-

Acknowledgment is made to In Jae Myung and Cheongtag Kim for graciously sharing their neural network code and their advice in helping the authors to understand the principles underlying "spiky" neural nets. The authors also thank Elke M. Kurz, Mark Rivardo, and Rose Strasser for their criticisms and suggestions. Correspondence should be addressed to either of the authors at the Department of Psychology, Bowling Green State University, Bowling Green, OH 43403 (email: seand@bgnet.bgsu.edu or tweney@bgnet.bgsu.edu).

tures, such as perceptrons, multilayer networks, and even adaptive resonance networks, as well as for learning routines, such as backpropagation. The "transparency" of the code, a function of *Mathematica*'s highly mnemonic design, made it especially powerful for bridging the gap between the theoretical statement of a net's function, generally given in the form of vector and matrix equations, and functioning code that actually runs the network. Furthermore, the ease with which *Mathematica* can be learned meant that attention could be focused upon the design of the networks rather than the language of their implementation.

According to Freeman (1994, p. iii), "the ease and speed with which [one is] able to implement a new network spoke highly of using *Mathematica* as a tool for exploring neural-network technology." Despite its relatively slow speed in the modeling of large networks, *Mathematica*'s flexible front-end environment made it clear that it might have utility in the design of neural networks and for explorations of novel network designs. With this in mind, we decided to try our hand at the modeling of a small, unconventional connectionist model.

We began by attempting to replicate results obtained by Kim and Myung (1995), in which temporal summation was used in a simple two-layer network to simulate semantic priming effects. Our goal in studying their network design was initially didactic; we wished to "test and stretch" our skills—and *Mathematica*'s efficacy as a development environment—by replicating a published finding based on a nonstandard network design.

"Spiky" Neural Networks

Kim and Myung (1995) built a two-layer network using an activation function that mimicked the "spiky" firings of real neurons. Such a network manifests gradual activation across time when presented with an input; an input item is "recognized" when the associated output activation exceeds a predetermined threshold level. In spite of the network's simplicity, Kim and Myung showed that such a network could manifest semantic priming effects; when given a "prime" item to recognize, a subsequent similar item would reach threshold more quickly than if the prime had not been presented. The finding is interesting because it suggests that even very simple networks can manifest these well-known priming effects.

Furthermore, because they used spiky neurons, their work suggests that neuronally realistic nets may have an important contribution to make in modeling cognition using neural nets. For us, Kim and Myung's results seemed appropriate as a test case, first, because the network was simple and small and, second, because its use of spiky neuron activation functions resembled actual neurons more than did the usual nonlinear activation function.

Kim and Myung found that a two-layer (24 input node, 6 output node) network utilizing this function, with weights trained using the Hebb rule, exhibited a semantic priming effect. The network's output activations were cumulative and built upon the previous states of the network, allowing one to study the rising activation of a pattern as it was presented to the network over the course of time. This differs from many other simple two-layer networks, in which the actual "running" of a network entails the production of a single set of output activations from one presentation of an input.

To determine whether or not an input "neuron" would fire at a given time step, Kim and Myung used a Poisson-distributed random process, thus adding what they called a *temporal summation* dimension to the usual spatial summation used in computing node activation. Their activation function was defined by Equation 1:

$$\Delta \text{net}_j(t) = \sum_{i=1}^{m} w_{ij} a_i \left[\int_{t}^{t+\Delta t} S_i(h)dh \right] - \xi \cdot \text{net}_j(t)\Delta t. \quad (1)$$

The weights between each input unit (i) and output unit (j) are represented by the w_{ij}, activations of the input layer are represented by a_i, the integral of $S_i(h)dh$ represents a Poisson-distributed random neuron firing process, and $\xi \cdot \text{net}_j(t)\Delta t$ represents a leakage term for the network. It should be noted that this equation represents the change in the activation of the output nodes; the values of Δnet_j computed by this equation are added to the previous time step's output activation values.

With regard to the semantic priming effect, the amount of time for the network output activations to reach threshold was less for a "semantically similar" pair of input patterns than for "semantically dissimilar" input pairs. In addition, Kim and Myung showed that their network exhibited, as expected, the usual pattern of effects due to stimulus onset asynchrony (SOA), which we will describe more fully below. Replicating this, Kim and Myung's major result, was a first goal for our initial *Mathematica*-designed networks.

The Simulations

We began by loosely adapting Freeman's code for an adaptive linear combiner (Freeman, 1994, p. 40)—essentially a two-layer perceptron network. Since Kim and Myung's networks (originally written in C) involved following the course of output activations over time and used an activation function that included a probabilistic, temporal summation aspect, we retained very little of Freeman's actual code while keeping many of his programming conventions. The "style" of our *Mathematica* network is quite similar to Freeman's, but the specific coding used was a functional equivalent of that used by Kim and Myung.

Replicating Kim and Myung's results involved coding the network using *Mathematica*'s Macintosh front end; we created our networks in a graphical environment within which we performed mathematical calculations, ran the simulations, and viewed the results of the network performance. Annotated code for the "prime presentation" portion of the replication networks, as displayed in *Mathematica*'s front end, can be found in Figure 1. (The code is described more fully below.)

Mathematica is particularly useful in allowing one to quickly modify code and to do "on-the-fly" calculations.

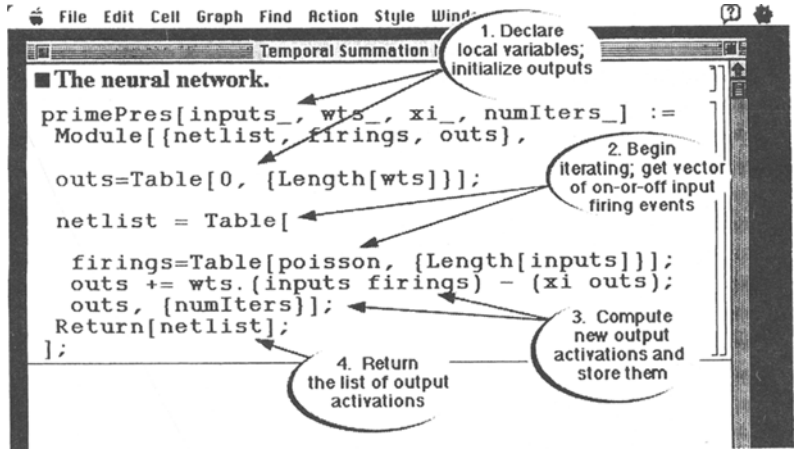

Figure 1. A screen grab of *Mathematica*'s front end, illustrating the "prime presentation" portion of the temporal summation networks.

This follows from the fact that it is an "interpreted" language—that is, one in which code is compiled and processed by the programming language one line at a time—rather than a "compiled" language such as C in which source code must be compiled by the user before running the program. While the runtime for a neural network in an interpreted language is normally much higher than for a compiled language, the ease with which we were able to make minor adjustments and experiment with our networks justified the extra time needed to run the simulations. Incidentally, though we did not need to use the option, *Mathematica* allows one to compile regularly used functions in order to increase runtime speed. In addition, it is relatively simple to embed calls to compiled routines written in other languages; thus, were we to study the effects of "scaling up" the present network, to see if its behavior changed as the net grew in size, the logical way to proceed would be to incorporate a C or C++ routine to do any heavy number crunching.

We began by studying the network performance of multiple replications of the networks. The heart of our routine (the first half of which is illustrated in Figure 1) was a function that continuously fed a given input to the network and tracked its output activations over time.

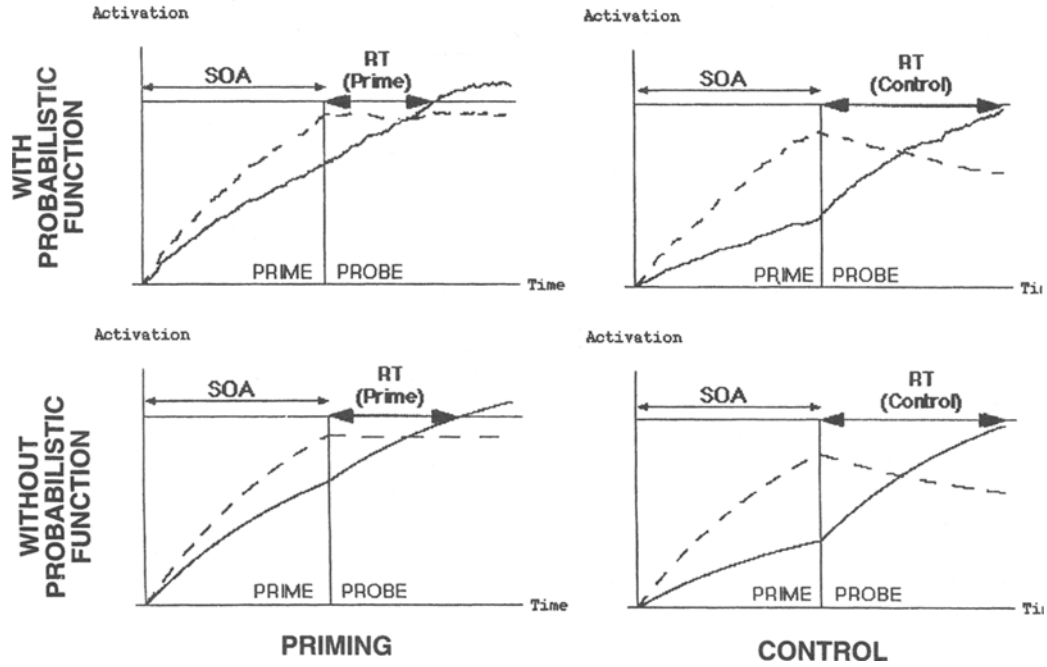

Figure 2. Examples of all four simulation conditions, with probabilistic function (priming and control) and without probabilistic function (priming and control). Each "prime" and "probe" consisted of 500 network iterations. The ordinate is on an arbitrary activation scale. Reaction times for the priming conditions [RT (Prime)] are clearly shorter than reaction times for the control conditions [RT (Control)].

Networks were constructed and run in both the priming condition and the control condition, in which either similar or dissimilar pairs of inputs were successively presented to the network. Our simulations exhibited the same priming effect as Kim and Myung's, and we considered the replication portion of our study a success.

To determine what aspect of the network was chiefly responsible for the priming effect, we modified the network in several steps, successively removing sections of the network's functionality and testing its performance. The results of our simulations are reported in Duncan, Kurz, Rivardo, and Strasser (1996). By removing the spiky neurons from the network, we found strikingly similar results to the network with spiky neurons included. We concluded that the semantic priming effect is not a result of the inclusion of a probabilistic element into the network but rather was due to the fact that the network is a gradual activation network in which each input item is presented repeatedly over time. In the course of this work, *Mathematica* was quite useful in presenting the results of both the replication and the summation-removed networks. Plotting the course of activations with *Mathematica* made the similarity between the temporal summation networks and spatial summation networks very clear. One set of activation plots can be seen in Figure 2. With such plots, we were able to conclude that the addition of the probabilistic component of the temporal summation process did not adversely affect the priming effect other than adding a small amount of "noise."

Kim and Myung also showed that the priming effect increased with length of SOA—that is, the difference between the time of presentation of the probe and the prime. To assess the contribution of the temporal summation (the Poisson-distributed random process that dictated whether or not a neuron would fire), we ran multiple simulations with and without temporal summation while varying the SOA length. A comparison of 200 replication networks and 20 networks without temporal summation can be seen in Figure 3. Except for the noise introduced by the probabilistic character of the spiky neurons, the two graphs appear nearly identical.

Again, *Mathematica*'s combination of a sophisticated programming environment with data visualization tools made the assessment of our networks' performance quite simple.

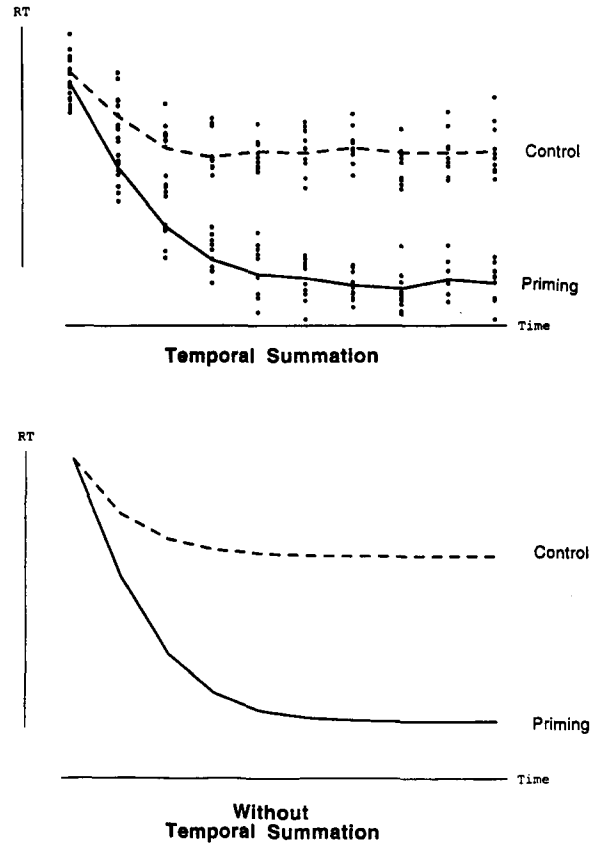

Figure 3. Results for the temporal summation and nontemporal summation networks as a function of stimulus onset asynchrony (SOA). SOA varied from 1 to 4,501 time steps for both simulations, with activations plotted on an arbitrary scale.

Using *Mathematica*

Following Freeman (1994), we implemented node activations and weight matrices as *Mathematica* lists, allowing us to capitalize on the language's multiple representations of lists as arrays and vectors. Similarly, Ratcliff (1994) briefly described a memory model of spreading activation in which *Mathematica* lists were treated as matrices and vectors, permitting activations to be computed utilizing simple matrix multiplication.

The symbolic nature of *Mathematica*'s programming language has several benefits for neural network modeling. Indexing capabilities are easily handled within the framework of standard *Mathematica* commands. For example, the command "Table" can take a symbolic expression and evaluate it for a symbolically expressed number of times, placing the result in a list structure that can itself be named as a symbolic expression. Thus, in the network code shown in Figure 1, the line "*firings* = Table[*poisson*, {Length[*inputs*]}]" creates a table of values from the output of "*poisson*" (here, the integral found in Equation 1), evaluates it for a number of times that corresponds to the computed length of the input string (*inputs*), and makes the result (i.e., the entire table) itself a new variable called *firings*. Although iterative processing is obviously not something specific to *Mathematica*, the manner in which one can reference the result of the iterative process as a single symbol is quite useful in neural network programming.

With this in mind, we translated Kim and Myung's activation function into the following *Mathematica* code:

```
outs += wts.(inputs firings) − (xi outs);
```

In this code, we were able to treat the network input activations (*inputs*) as a list of activations multiplied by a Poisson-distributed on/off neuronal firing variable (*fir-*

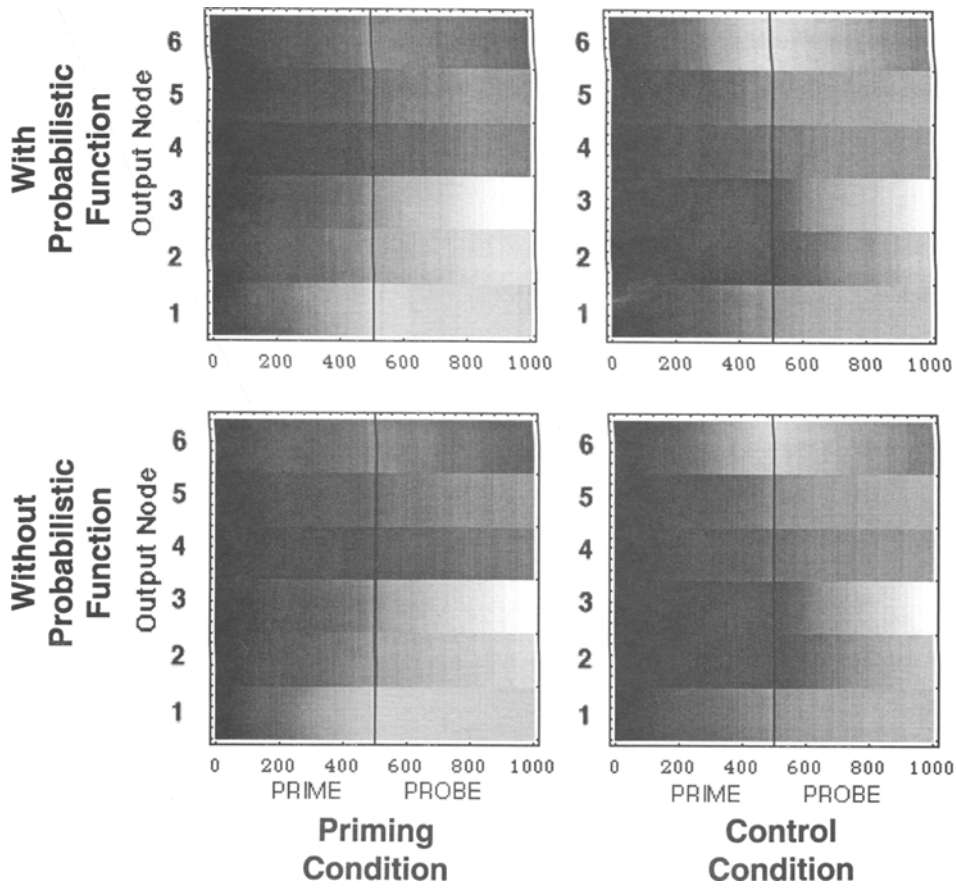

Figure 4. Hinton diagrams of network performance over time, plotted using *Mathematica*'s ListDensityPlot command. Six prime/probe pair activations are shown from top to bottom; time is on the abscissa. The change from prime presentation to probe presentation occurs at time step 500.

ings). The result was treated as a vector that was multiplied by the network weight matrix (*wts*) and added to the previous output activation (*outs*) in, again, a list-like manner.

Mathematica's flexibility in this regard cannot be understated. In stepping through the above line of code, first, note that the network takes the input activations (*inputs*, a vector of ones and zeros) and multiplies each element by a vector of equal length (*firings*, also consisting of ones and zeros). This second vector represents, for each time step, the Poisson-distributed random firing events—that is, a value of 1 represents a fired node, and a value of 0 represents a nonfired node. Then, the inner product of the weight matrix and the resultant value of the previous computation is calculated, from which a decay term equal to a constant multiplied by the previous output activation is subtracted.

Mathematica makes it easy to create Hinton diagrams (Freeman, 1994, p. 23; Hinton, McClelland, & Rumelhart, 1986) of various portions of neural networks. Although these diagrams are normally used to illustrate the weights or activations of a network after a training procedure, they can also be used in small networks such as ours to view the progress of output activations over time. In Figure 4, a kind of "extended" Hinton diagram, we can see the behavior (across time) of all 6 output nodes for each of the four kinds of neural networks run (priming vs. control, temporal summation vs. no temporal summation). Although not shown here, the diagnosticity of such plots can be enhanced even more when presented in color. Whether black and white or color, *Mathematica* allows one full control over display parameters, such as density, color value, gray-scale range, and so on.

This illustrates another strength of *Mathematica*: the ease with which one can take a set of data and represent it multiple ways. Note that the density plots effectively present the values of six variables over hundreds of time steps; corresponding line plots (which can also be done with *Mathematica*) are simply too cluttered to be useful.

In sum, we believe that *Mathematica* is a powerful environment for designing neural networks for psychological research. As an interpreted language, it is much slower than other programming environments, yet much more easily modifiable than compiled languages, allowing the experimenter to "explore" the behaviors of the networks. The representation of data structures within the language

allows for the simple programming of computationally complex neural network activation code. Finally, the combination of programming ease and powerful visualization tools permits a wide range of methods to assess the performance of neural networks.

REFERENCES

DUNCAN, S., KURZ, E., RIVARDO, M., & STRASSER, R. (1996, November). *Semantic priming in a simple two-layer neural network.* Poster presented at 37th Annual Meeting of the Psychonomic Society, Chicago.

FREEMAN, J. A. (1994). *Simulating neural networks with Mathematica.* New York: Addison-Wesley.

GRONLUND, S. D., SHEU, C.-F., & RATCLIFF, R. (1990). Implementation of global memory models with software that does symbolic computation. *Behavior Research Methods, Instruments, & Computers*, **22**, 228-235.

HINTON, G. E., MCCLELLAND, J. L., & RUMELHART, D. E. (1986). Distributed representations. In D. E. Rumelhart & J. L. McClelland (Eds.), *Parallel distributed processing: Explorations in the microstructure of cognition. Vol. 1: Foundations* (pp. 77-109). Cambridge, MA: MIT Press.

KIM, C. & MYUNG, I. J. (1995). Incorporating real-time random time effects in neural networks: A temporal summation mechanism. In J. D. Moore & J. F. Lehman (Eds.), *Proceedings of the Seventeenth Annual Conference of the Cognitive Science Society* (pp. 472-477). Hillsdale, NJ: Erlbaum.

LORIG, T. S., & URBACH, T. P. (1995). Event-related potential analysis using *Mathematica. Behavior Research Methods, Instruments, & Computers*, **27**, 358-366.

MASTERS, T. (1993). *Practical neural network recipes in C++.* San Diego: Academic Press.

RATCLIFF, R. (1994). Using computers in empirical and theoretical work in cognitive psychology. *Behavior Research Methods, Instruments, & Computers*, **26**, 94-106.

RUMELHART, D. E., MCCLELLAND, J. L. (Eds.) (1986). *Parallel distributed processing: Explorations in the microstructure of cognition. Vol. 1: Foundations.* Cambridge, MA: MIT Press.

WOLFRAM, S. (1994). *Mathematica* (Version 2.2.2) [Computer programming language]. Champaign, IL: Wolfram Research, Inc.

WOLFRAM, S. (1996). *Mathematica* (Version 3) [Computer programming language]. Cambridge: Cambridge University Press.

(Manuscript received September 30, 1996;
revision accepted for publication January 6, 1997.)