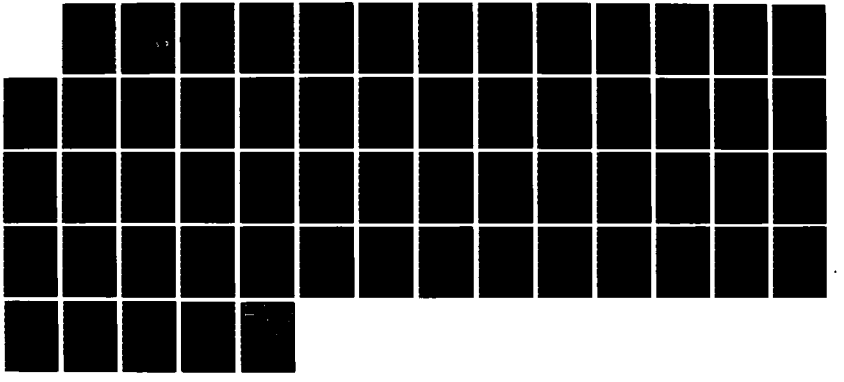
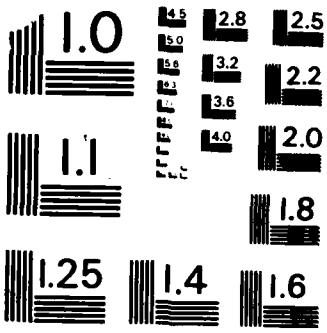
AD-A175 612 MODELING AND SIMULATION III SIMULATION OF A MODEL FOR DEVELOPMENT OF VISU..(U) BROWN UNIV PROVIDENCE RI CENTER FOR NEURAL SCIENCE A B SAUL ET AL. 15 DEC 86 1/1

UNCLASSIFIED TR-36 N00014-81-K-0041 F/G 6/16 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS - 1963 - A

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: #36
2. GOVT ACCESSION NO.:
3. RECIPIENT'S CATALOG NUMBER:

4. TITLE (and Subtitle): Modeling and Simulation III: Simulation of a Model for Development of Visual Cortical Specificity.

5. TYPE OF REPORT & PERIOD COVERED: TECHNICAL REPORT

6. PERFORMING ORG. REPORT NUMBER:

7. AUTHOR(s): A. B. Saul and E. E. Clothiaux

8. CONTRACT OR GRANT NUMBER(s): N00014-81-K0041

PERFORMING ORGANIZATION NAME AND ADDRESS
CENTER FOR NEURAL SCIENCE
BROWN UNIVERSITY
PROVIDENCE, RHODE ISLAND 02912

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: N-201-484

CONTROLLING OFFICE NAME AND ADDRESS
PERSONNEL ANDTRAINING RSCH. PROGRAM
OFFICE OF NAVAL RESEARCH, Code 442PT
ARLINGTON, VIRGINIA 22217

12. REPORT DATE: December 15, 1986

13. NUMBER OF PAGES: 55 pages

MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office):

15. SECURITY CLASS. (of this report):

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE:

DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited. Publication in part or in whole is permitted for any purpose of the United States Government.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

To be published in the Journal of Electrophysiological Techniques.

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Modeling
Simulation
Visual System
Plasticity
Synaptic Modification
Numerical Methods

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

We present a tutorial describing aspects of the coding of simulations of models of visual cortical development. The model considered has an anatomy of an excitatory projection from thalamus to cortex combined with intracortical inhibition. Cortical cells develop specificity to stimulus patterns in this model only when appropriate experience enables synaptic modification to organize the network.

The simulation consists of a time loop. For each iteration of this loop, a

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

stimulus is generated, the cortical response to this timulus is computed, and synaptic weights are modified. The developing network is tested intermittently and the behavior of the system analyzed.

Some of the details of the coding given include a method of describing rearing conditions, a convenient abstract form for the input stimuli, an iterative calculation of the intracortical feedback, a simple way to store synaptic stremgths. and routines for performing the analysis.

A.B. Saul and E.E. Clothiaux

Center for Neural Science

Brown University, Providence, RI 02912

Modeling and Simulation III:

Simulation of a model for development of visual cortical specificity.

key words:
modeling, simulation, visual system, plasticity, synaptic modification, numerical methods.

Partially supported by ONR Contract N00014-81-K-0136.

We thank J.D. Daniels for significant contributions to this paper.

Address for correspondence:

Alan Saul
Department of Psychology
Dalhousie University
Halifax, Nova Scotia, Canada B3H 4J1

Abstract

We presents a tutorial describing aspects of the coding of simulations of models of visual cortical development. The model considered has an anatomy of an excitatory projection from thalamus to cortex combined with intracortical inhibition. Cortical cells develop specificity to stimulus patterns in this model only when appropriate experience enables synaptic modification to organize the network.

The simulation consists of a time loop. For each iteration of this loop, a stimulus is generated, the cortical response to this stimulus is computed, and synaptic weights are modified. The developing network is tested intermittently and the behavior of the system analyzed.

Some of the details of the coding given include a method of describing rearing conditions, a convenient abstract form for the input stimuli, an iterative calculation of the intracortical feedback, a simple way to store synaptic strengths, and routines for performing the analysis.

In parts I and II of this series (refs. 3,8), we discussed the applicability of modeling to problems in visual cortical physiology, and described some of the models which have been proposed, concentrating on the development of specificity in neurons receiving excitatory inputs from the lateral geniculate nucleus (LGN) and inhibitory inputs from other cortical units. We now present a tutorial in coding simulations of such a model. The goal of these simulations is to provide the details of the response properties of the elements of the model under various conditions, including a number of simulated normal and abnormal rearing conditions and over a range of parameter values.

OVERVIEW

Experiment, model, and simulation

The simulations we consider mimic, in form, classic deprivation experiments. Kittens are typically reared under special conditions, and a population of single units is then tested physiologically to determine their visual response properties. The goal of such experiments is to assess the effects of different rearing conditions on the response properties of the cortical units.

The second paper of the series (ref. 8) reviews the results of numerous experiments on the neuronal development of kitten visual cortex. We have concentrated on the development of <u>orientation selectivity</u>. As the second paper illustrates, there are numerous models in the literature which posit mechanisms of orientation selectivity and attempt to show how visual experience might affect components of these mechanisms. The various mechanisms proposed use different schemes of excitation and inhibition to wire up a network of cortical

cells. These excitatory and inhibitory connections are assumed to be modified by visual experience in order to produce the alterations of response properties seen experimentally. Although there is practically no direct evidence in mammalian studies for the hypothesis that it is such changes in wiring which bring about the changes accessible to microelectrodes, there is general acceptance of the central hypothesis that synapses are modified. Most models have applied versions of Hebb's postulate (5), which produce connections whose efficacy is related to the correlation between pre- and post-synaptic activities.

To illustrate some techniques in coding simulations we consider a model based on an anatomy of excitatory afferent fibers from thalamus to cortex combined with inhibitory intracortical fibers. The afferent fibers are divided into two groups: fibers relaying signals from the left eye and fibers relaying signals from the right eye. Visual stimuli are input as activities on the afferent fibers to cortex. These stimuli drive the cortical cells, determining, along with the cortical activities, the modification of the synaptic connectivities.

The model we are considering can be summarized by the following equations.

$$R = f(SA - QR) \tag{1}$$

$$dS/dt = g_S(A,R,S;\text{parameters}) \tag{2}$$

$$dQ/dt = g_Q(R,R,Q;\text{parameters}) \tag{3}$$

A (a vector of dimension equal to the number of afferent fibers, M) represents the activity in the fibers afferent to cortex. R (a vector of dimension equal to the number of cortical cells, N) is the response of the cortical cells. S (an N x M matrix) and Q (an N x N matrix) hold the synaptic weights, excitatory from LGN and inhibitory from cortex, respectively, onto the cortical cells. We let

all synaptic strengths be positive, and subtract the intracortical inputs in order to render them inhibitory. f is a function describing the relationship between the inputs to cortical cells and the resultant response: this function is generally chosen to be sigmoidal, incorporating a firing threshold and a saturation at high discharge rates. The synaptic modification rules are given by g_S and g_Q, which depend on the pre- and post-synaptic activities (respectively A and R for the afferent and R and R for the intracortical synapses), possibly on the current values of the weights, and additionally on a number of parameters which determine, among other things, how rapidly synaptic modification progresses. Although various models choose different forms for the function f in (1), and take a number of positions on what the inputs A consist of, the chief distinction between these models is the choice of synaptic modification rules g (see Table 1 in ref. 8). One can choose to modify only the excitatory synapses, or only the inhibitory synapses, or both - in either the same way or in different ways. Fortunately, this significant variation in models of the form (1)-(3) can easily be handled by the simulation coding we will describe.

Computing the afferent activity, A

Our first job in translating the model into code is to compute the afferent input A. Now the visual inputs to cortex are determined by the rearing conditions. Thus, different rearing conditions lead to different cortical activity patterns, affecting synaptic modification, resulting in the development of different connectivities and hence to different cortical response properties. Our code therfore starts by allowing the desired rearing conditions to be specified. Since this is a major independent variable in running simulated experiments, we code the rearing conditions in an accessible ("English language")

form. For instance, in order to perform an experiment on the results of monocular deprivation, one simply instructs the program to close one eye.

The next step in the code is to take the given rearing condition, and to translate this simple English language description into appropriate simulated visual stimuli (the variable A). Such stimuli can be thought of as activities on a set of afferent axons, and are coded as a vector, with the components of the vector corresponding to the activities on individual axons. Thus, monocular deprivation by closing the left eye will lead to a vector with some pattern of activities in the components corresponding to the right eye, but with only noise in the portion of the vector corresponding to the left eye. At each moment of simulated time, a new stimulus is generated using the general description of the rearing conditions, implying that the simulated visual stimuli continually change as in real life.

Before continuing to describe the sequence of routines in the program, we pause to discuss the crucial nature of the input coding. Despite our appeal above to "real life", the key to successful coding of a simulated visual environment is an appreciation of the necessity of abstraction. The challenge in implementing a simulation of a neural model is to create a formal structure which captures the essence of the real structures being modeled. The visual world is astoundingly complex: digitized video images of it are merely a shadow of the reality. One might be tempted to provide "real" input to a simulated cortex by hooking up a camera to a computer, but this would neither mimic any neural events nor allow understanding and control of the input end of the model. We will not try to create a particularly rich imitation of the visual world, but instead will try to faithfully reproduce the few aspects of the retinal images with which we choose to be concerned, such as orientation.

Most of the models described in ref. 8 arose from the excitement generated

by a number of experiments in which visual experience was restricted to a very limited range of oriented contours (see citations in ref. 8). Kittens were raised, for example, viewing only horizontal or only vertical lines. Modelers thus abstracted out oriented lines as an important feature of the visual environment, whose presentation could be controlled. We suppose then that the environment in our model consists of brief presentations of variously oriented bars. Such a bar in visual space can reasonably be expected to stimulate a set of retinal ganglion cells which form a bar on the retinal surface. Assuming a simple retinotopic projection to cortex on the scale of this bar (not necessarily a valid presumption, since it essentially amounts to considering everything between retina and cortex as strictly relay mechanisms), we can imagine that a sheet of receptors encodes the visual stimuli by determining which receptors are covered by a bar (Figure 1). Different orientations all stimulate a central cell, corresponding to the axis of rotation of the family of bars, but differ in the periphery of the sheet.

The activities of the M receptor elements in the sheet can be thought of as forming the components of a vector of dimension M. The <u>environment</u> of various oriented bars can be implemented as a set of such vectors. A vertically oriented bar stimulates certain receptors and is coded into a corresponding vector, while a horizontal bar corresponds to a different vector. As the orientation of the visual stimulus rotates from horizontal to vertical and back to horizontal, the vectors corresponding to these orientations rotate through the abstract space in which they live. Thinking of these vectors as directed arrows, the arrowhead would trace out a circle as the orientation varies through 180 degrees. (We say that the set of vectors corresponding to oriented stimuli has a <u>circular structure</u>: the vectors in it are permutations of each other and all lie on a circle in their possibly high-dimensional space, since the order in which the

receptors are listed in the vector is arbitrary, and the stimuli are all identical except in terms of which of the receptors are stimulated.) Thus, although it is certainly feasible and sometimes helpful for graphical purposes to represent the environment in terms of a two-dimensional sheet of a large number of cells, as long as we are considering only orientation we do not need the additional possibilities afforded by two dimensions (such as being able to encode position, length, width, or movement). The orientation domain is one-dimensional and periodic and can be represented by any circular set of vectors. Trivial examples of some circular sets of vectors in three dimensions are {(1,0,0), (0,1,0),(0,0,1)} or {(1,1,0),(0,1,1),(1,0,1)}. We will want to derive more useful sets in higher dimensions, without necessarily resorting to the scheme mentioned above where bars were projected onto a sheet of cells.

It turns out that the important parameter in defining the simulated environment determines how close together the input patterns lie on the circle. (Translating back to the real world, what matters is, how similar do various oriented bars look at the level of the input to cortex?) Our approach to constructing input patterns on the computer might be clarified by way of an analogy to color mixing. If we want to construct the color yellow, we can do so by mixing red and green light. What we actually mean is that we can create the perceptual color yellow by mixing the spectral colors red and green. A spectral color is simply monochromatic light, and is best described by the wavelength. However, we make an assignment from wavelength to color name, for convenience. Now, different perceptual colors can have very similar or very different spectral compositions: yellow and orange have similar compositions in the form of a banana and an orange illuminated by white light, while they have quite different compositions in the form of a white piece of paper illuminated with monochromatic light at 580 nm (yellow) and 600 nm (orange) wavelengths. We want to control

the similarity of our input patterns in just this manner. We start with some abstract tokens (unit vectors) which are assigned orientation names analogous to the way that wavelengths are assigned color names. We then mix these tokens in various amounts to create more of one orientation than of others in our final input pattern, which thereby "looks like" the orientation of the dominant token. If we were to use only a single token at a time, different input patterns would consist of completely different tokens. By mixing together tokens which represent similar orientations, we produce "oriented" patterns which overlap in their distribution of tokens, despite having different numbers of each type of token. We retain color names, or orientation names, for the tokens as well as for the mixtures because the correspondence is convenient, once we understand the possible confusion.

We have replaced the two-dimensional sheet of receptors by a circle of input elements. These elements will be our tokens, analogous to spectral colors. Each element on the circle carries an activity, and the afferent input to cortex consists of the vector formed by the activities in these components. We generate a stimulus vector by first designating one of the input elements on the circle as representing the desired orientation. For definiteness, say the input elements are indexed by 1 through 10, with 1 representing vertical and 6 representing horizontal. Say we want to create a vector coding a vertical bar. We designate element number 1 as our "center" component. The stimulus vector is given a large amplitude in this component, indicating that the vector corresponds to this orientation. Neighboring components on the circle (that is, elements indexed by 2, 3, 9, and 10) are given smaller amplitudes, indicating that those orientations are similar but not identical. As we move around the circle of input elements, we assign amplitudes which reflect how similar the orientation represented by each element is to the desired center orientation.

Components which represent similar orientations are active, while elements representing orientations orthogonal to the center orientation (elements 4, 5, 6, 7, and 8) are relatively inactive.

Computationally, we use a function which is maximal at the center element and decreases away from this center (Figure 2). This function sets the activities in each of the input elements. The width of this function controls the overlap between patterns in a straightforward way. Returning to the color analogy, this function is analogous to the spectrum of the illumination, with the width corresponding to the bandwidth. If the function is parametrized as very narrow, then different patterns would overlap very little. (Interpreting back to physiology, the patterns of firing in the optic radiations afferent to a cortical cell would be quite different for a vertical bar and a bar inclined 10 degrees toward horizontal.) If the function decreases slowly, all patterns would be quite similar. Two or three lines of code ('FUNCTION contour' in the code given below) suffice to calculate the input vector with such a system, and it carries with it the graphical form of a tuning curve which can be compared with the output tuning curve, since each input element corresponds to the pattern centered on it. We have moved somewhat far from the reality of oriented contours, however, and we must keep in mind that our construct is an abstract model, and not a realistic model, of the inputs to visual cortex. That is, the components of our stimulus vector do not correspond directly to actual fibers afferent to a cortical cell (but we will still refer to them as "afferent fibers" because of the anatomical analogy).

Computing the cortical response, R

Once the activity on the afferent fibers is determined, we can compute the

activity in each cortical cell by integrating the afferent inputs to each cell with the intracortical inputs (equation 1). To code this aspect of the model we write the synaptic weights to a cortical cell as the components of a synaptic weight vector. To determine the postsynaptic potentials (SA - QR) of the cortical cells we take the product of the synaptic weights (S and -Q) with a vector which is formed from the components of the input vector to the afferent fibers and the other cortical cell activities coming in along the intracortical fibers (A and R). f(SA - QR) determines the actual cell activity R, where the function f comes from some model for the firing behavior of kitten cortical neurons (usually f incorporates a firing threshold and firing saturation, with a somewhat linear range in between). Because the intracortical inputs depend on the activity in other cortical cells which are being computed simultaneously (that is, our computation of R requires that we know R already), we have a problem. Our system is a recurrent, <u>feedback</u> type system, and because of size, nonlinearities, and a desire to mimic reality, we do not compute the activity explicitly, but instead perform an <u>iterative calculation</u> which approximates the steady state activity. Through such computations (described below) we derive an activity for each cortical cell. Writing all the cortical cell activities as a vector, we have the output of the simulation in raw form.

Computing the synaptic modifications, dS/dt and dQ/dt

We can then use the cortical activity to drive synaptic modifications, which are the result of the particular visual stimulus which was experienced at the most recent moment of time. Coding the synaptic modifications amounts to writing the model's modification rule in functional form, which usually takes no more than a few lines of code.

Simulating visual experience

The code for generating a stimulus, computing the cortical response, and modifying the synapses falls within a simulated time loop which is the main driver of the simulation. On each pass of the time loop a potentially different stimulus from the environment which is consistent with the specified rearing conditions, is presented through the set of simulated afferent fibers. (We use a random variable to pick the order in which the stimuli are presented to the system.) Passing through the simulated time loop thus models the experimentally desirable aspect of a kitten's rearing environment, which consists of a controlled set of visual stimuli that the kitten is exposed to in real time in some order, possibly random. In fact, experimental control over a kitten's visual experience is distressingly poor. A simulation can take into account some of the less controlled aspects of this experience, by, for instance, modeling moments of sleep or inattention and the like by noisy, rather than patterned, visual inputs. Rather than presenting a rigid set of oriented patterns, with the inputs to the two eyes identical, we prefer to present some statistical distribution of a large set of stimuli which includes occasional unpatterned inputs and differing patterns in the two eyes. As simulated time proceeds, the various stimuli rewire the network, with a speed which depends on the speed of synaptic modifications and on the the rate at which stimuli which strongly affect synapses appear in the environment.

Analyzing the cortical response properties

The above process of generating a stimulus, computing the cortical response, and modifying synapses, is occasionally interrupted in order to test the network

CODING

The following example of a program designed to simulate a visual cortical development model will illustrate the details involved in coding such a simulation. We have written versions of this program in BASIL, FORTRAN, and PASCAL for various machines, including LM2, AMDAHL, IBM, VAX, and APOLLO, and have found that modifications are easily made, and usually have proven useful in enriching the code. We present a PASCAL version (see ref. 4 for a good description of this language), some of which we give in pseudocode to sketch the structure, leaving out some details and all of the machine-dependent features. The omissions include the graphics routines, which we feel are crucial in understanding the performance of the simulations, but which must be created according to one's own resources and purposes. Complete source codes in PASCAL (for APOLLO) are available from the authors, although these programs are specific examples of the general codings described in this article.

We simulate processes occurring through time, so the program consists of a time loop. Each moment of time corresponds to a stimulus presentation. One could conceivably relate this simulated time unit to some real time unit, by reasoning that real visual stimuli capable of driving synaptic modifications occur with a certain duration and rate (ref. 1). The rate of change of synaptic strength with individual stimulus presentations would be the appropriate quantity to measure in the real world, and could then be set in the simulation as a parameter in the synaptic modification routines (in the example of a synaptic modification rule given below this is the time constant 'h'). The <u>steady-state response</u> to each stimulus presentation is used to perform synaptic modification. More complex simulations might reflect a less discrete view of time, allowing for a smoothly varying stimulus and a <u>dynamic response</u>, simply by

with a standard set of stimuli. These test sessions allow the evolution of the network to be evaluated, so that the responses of cortical cells can be compared at different points in their experience. Each test session consists of three main parts: generating a stimulus, computing responses, and analyzing the responses. The analysis takes the raw data and compiles it into more interesting measures of performance, such as indices of selectivity and ocular dominance and population statistics. Note that no synaptic modification is performed during test sessions. How often one tests the system depends on the detail desired and on the speed at which the network is changing.

Summary of overview

To summarize, the structure of the simulation comprises a loop indexed by simulated time, during each iteration of which several procedures are called. First, a stimulus is chosen according to the desired rearing conditions. Next, this stimulus is translated into an input vector. The cortical response is then computed. Finally, synaptic strengths are modified. A branch to a test session is made at desired intervals (Figure 3).

Before proceeding to a more detailed examination of explicit coding techniques, we would like to emphasize that coding the visual environment of the kitten is the most demanding task in constructing the simulation. A model's anatomy for the kitten cortex and rule for synaptic modification imply the logic of the code that one must write. Capturing the essential characteristics of the kitten's actual environment in code, however, is not straightforward. The amount of discussion in the overview, as well as the large number of lines in the code, devoted to this one task attests to the importance of this task in simulating the present model for the neuronal development of kitten cortex.

inserting an inner time loop.

Note that for purposes of clarity, we pass parameters between routines explicitly even when it is neither necessary nor desirable. (Explicitly passed parameters are listed in the parentheses to the right of the called routine, both in the calling statement and in the definition for the routine. If the called routine changes the value of a parameter, then that parameter is defined as a variable in the parentheses associated with the routine definition.) In general the code given below is not optimized for run-time, but instead we have attempted to make it understandable to the reader.

MAIN PROGRAM LOOP

Figure 4 gives the driver for the program, which generates an input, calculates the output, modifies synapses, and analyzes the responses. Prior to entering this principal portion of the program, the code must initialize variables and interact with the user, but we omit those important but tedious preliminaries for the sake of brevity.

STIMULUS GENERATION

The independent variable in many simulation experiments is the environment. In order to easily handle the various rearing conditions in the most convenient way, we have coded the stimuli in plain English. The routine in Figure 5 determines the current stimulus as a function of time and a random variable. For this example, we simulate an experiment where the early rearing (prior to time 1000) is in the dark, followed by rearing in the light assuming that: 20% of experience is without visual input (stimuli consisting of noise),

20% consists of patterns through only one eye (either the right or the left), 10% has uncorrelated patterns in the two eyes (as might be found in a strabismic animal), and the remaining 50% presents similar (although not always identical) patterns to the two eyes. Other rearing conditions that might be used are 'normalrearing', 'reverse suture', 'adapting', and 'sinusoid', which we will describe shortly. Different simulation experiments are run by inserting the desired rearing conditions into the above procedure. This process can be made "user-friendly" by coding the procedure as a menu-driven interactive choice paradigm.

The integer 'random_seed' is uniformly distributed over a large interval, so that we can obtain random variables uniformly distributed over arbitrary intervals as above by applying the 'MOD' function, which returns the integer remainder upon division by the modulus (10 in the example here).

The individual stimulus-setting routines that can occur in the 'PROCEDURE Choose_stimulus' follow (Figures 7-9). They fill in the components of the variable 'stimulus', which has the type definition given in Figure 6. This particular PASCAL-dependent construction provides a convenient way to translate the "English-language" rearing conditions into code, but could be replaced with a series of conditional statements or arbitrary assignments in other programming languages. The type:

rearing = (correlated,md,rs,dr,disparity,strabismus,adaptation,periodic)

consists of these tags which refer to the various rearing conditions: 'correlated' is used to provide identically corresponding patterns in the two eyes, 'md' means monocular deprivation and presents a pattern in the open eye but only noise in the closed eye, 'rs' stands for reverse suture which comprises successive monocular deprivations, 'dr' abbreviates dark rearing which simply presents noisy inputs to both eyes, 'disparity' gives partly correlated patterns

to the two eyes, 'strabismus' gives uncorrelated patterns to the two eyes, 'adaptation' sets a particular stimulus for repeated presentation to either eye alone or both eyes together, and 'periodic' allows for presentation of periodic patterns ("gratings"), rather than "bars". The other type definitions are: angle = 1.. numangles, where numangles is the number of patterns in the simulated visual environment, and eyes = (left,right,both).

The routines in Figures 7-9 produce the desired simulated experience in an understandable, "English-language" form, by taking advantage of the record-variable 'stimulus'. We must now translate this information into a computational, numerical form, namely an array of values for the activities in the afferent fibers. Active fibers will be assigned numbers near 1, and inactive fibers numbers near 0. Thus, if the visual exposure consists of monocular deprivation, with the left eye closed and the right eye viewing a pattern, then fibers from the left eye will carry small random values while the right eye afferents will be given values determined by the pattern. The procedure in Figure 10 and the functions it calls (Figures 11 and 12) perform this translation.

RESPONSE CALCULATION

At this point we have set up the input and we are ready to turn our attention at last to the model we want to simulate. The input has been coded into the vector 'afferent_activity' (A in equation 1) and will now be fed to the cortical cells, which will respond according to the state of their synapses. This response will then be used to alter the synaptic state.

We must solve equation 1, R=f(SA-QR). The variables S, A, and Q are fixed here, with our only problem being that we must deal with the intracortical

feedback which makes R a function of itself. While negative feedback provides stability to the visual system here, from a strictly computational point of view this feedback can cause instabilities when perroming the iterative calculations needed to solve the equation. For instance, it one starts with R being very small, tnere will be very little intracortical inhibition (QR will be small). But tnen one could reason that the lack of inhibition would lead to overexcitation, which in turn would lead to strong inhibition which would result in little response, and so on through a possibly growing oscillation. This simple-minded iterative scheme, which can be written in scalar form as

$$y_n = a - hy_{n-1},$$

is unstable for h>1. The constant h corresponds to the eigenvalues of the matrix Q in equation 1, and the difficulty of ensuring that the eigenvalues of Q remain less than 1 prevents the application of this simple explicit one-step scheme (where new response values are computed based solely on the values from the last iteration).

Multi-step schemes, such as

$$y_n = a - h(y_{n-2} + y_{n-1})/2,$$

can help but still fail to prevent oscillations for large h. What is needed instead is an implicit scheme (where the current iteration appears on both sides of the equation), for example of the form

$$y_n = a - h(y_{n-1} + y_n)/2.$$

This method is stable for all values of h. Solving such implicit schemes exactly can be done for relatively small linear systems, but is out of the question for our large, often nonlinear equations. Instead, a surprisingly simple method can often give satisfactory results: update individual cells from iteration n-1 to iteration n, while other cells remain in iteration n-1. That is, the updating is asynchronous, not all cells are updated at once. One advantage of this method is

that only one vector of activities need be stored, rather than two (the n-1'th and n'th) as in the scheme above. At any point in the computation, some of the components of this vector will belong to the current iteration (the n'th) while others will belong to the last iteration (the n-1'th). The advantage in conserving memory is usually not great in these days of large virtual memory. We should note here that these iterative methods also tend to be slow: if feedback can be avoided in a model, the simulations will run much faster. Given the necessity of simulating a recurrent network, however, this method seems to be as stable, convenient, and fast as possible.

The usefulness of this method comes from the fact that fluctuations produced by the iteration process are averaged out by adding the two consecutive responses. Clearly, for this process to work some averaging across different cells is needed. For instance, starting from very low activities again, one might expect the first few cells which are updated to respond strongly because of the lack of inhibition. These cells would then provide strong inhibition to later cells, which would never respond. The result would be an unintended separation of the cortex into those cells which were updated early, and are active, and those cells updated later, which are inactive. One way to avoid this problem is to update cells in random order, so that although during the first iteration the early cells might have an advantage (or disadvantage, depending on the initial state), there is no longer any meaning to early and late after the first round of updates. A cell which saw little inhibition when it was first updated could be updated again soon thereafter, when the early cells are providing strong inhibition. A large enough sample of cells is needed to allow this random ordering to effectively compute averages over each type of cortical response property.

The procedure in Figure 13 provides a coding of this random, asynchronous,

iterative method. Since the iterative process changes only the intracortical input, we only need to compute the afferent input to each cell once. The condition 'afferent_input_hasnt_already_been_computed(cell)' is simply given by the boolean array 'used': afferent_input_hasnt_already_been_computed(cell) := NOT used[cell]. Each time we choose a cell we set 'used' to TRUE for that cell, and we choose to iterate until all the cells have been chosen at least once. This means that at least one cell will only pass through this iterative procedure once, but unless a cell is chosen only in the early stages (which has a low probability) there is no harm done. It is important to keep in mind that only the feedback (QR in equation 1) is being forced to converge here, and that this is a global property: we are not really forcing each cortical cell to converge separately and independently of other cells (if one cell is less active and another more active than the ideal levels, the method will still work). Only the total cortical activity, which behaves like the average activity, must stabilize. This globality lets us take advantage of the averaging over many cortical cells. Simulations which generate specificity also provide a strong organizing influence which tends to overcome the relatively small biasing errors from the iterative computations, as cells learn to respond to the input pattern without regard to where they lie in the updating sequence.

The routine in Figure 13 calls separate procedures for afferent and intracortical inputs, but they are essentially identical calls to an inner product function (Figure 14). The other functions called in 'PROCEDURE Compute_response' are given in Figure 15. The sigmoidal function used here (a tanh function) for the function f in equation 1 has some nice properties, but one must be careful not to generate underflows and overflows in the 'EXP' function.

We used the variable 'synapses' in order to weight the afferent and

intracortical inputs to our cells. Its type definition is

```
synapses_type = RECORD
                    afferent,intracortical : ARRAY[cellindex] OF maxvector.
```

Thus, we get two matrices, 'synapses.afferent' and 'synapses.intracortical'. We define these matrices, however, as arrays of arrays ('maxvector' is defined above as an array of the largest dimensionality required), in order to easily pass a single row rather than all the rows of such a matrix. The entire matrix is never used all at once, but only row by row. Even in programming languages which do not provide such an array of arrays construction, one should consider stripping off a row of a matrix, putting the row into a buffer, and working with the buffer, rather than manipulating the entire matrix. On the other hand, if an array processor is available, the software should take advantage of the enormous efficiency of the matrix algebra handling.

SYNAPTIC MODIFICATION

The procedures in Figure 16 sketch a coding for the heart of the model, the synaptic modification algorithms (equations 2 and 3). Here, we call a common routine for modifying both afferent and intracortical connections, but completely different modification schemes would be used in some models, while many models modify only one of these pathways (see ref. 8). As mentioned above, the time constants in the modification rules (the parameter 'h', for example) determine the duration of a single iteration of the time loop in terms of the duration of a stimulus presentation in real time. To make a connection between simulated time and real time one would have to speculate far beyond current knowledge. We

emphasize that the output of the simulation will be very sensitive to the parameters in these key modification routines. The values for these parameters should usually be set interactively, rather than during compilation. However, a complete investigation of the behavior of these models requires sensitivity analysis (ref. 2), and systematic variation of parameters should be written into a program at some point for this purpose (see Discussion).

ANALYSIS

This completes the simulation of the model. In order to observe and study the simulation at intervals within the time loop, we occasionally interrupt the loop in order to test the network. A test usually consists of three steps: 1) compute the cortical cell activities using a standard set of test stimuli; 2) analyze the activities to make explicit the relevant measures of interest, such as selectivity and ocular dominance; and 3) write the data to output devices and files. This can be a tremendously complex task, and we will only indicate some of the basics. Much of the work in coding program outputs involves device-dependent routines, especially for graphics. The procedure in Figure 17 sketches in one block of pseudocode the sorts of actions needed. Note that the main steps replicate the simulation's driver (Figure 4), with the differences being that stimuli are chosen differently and the synaptic modifications are bypassed. The following example (Figures 18-22) makes the sketch explicit.

To begin the analysis we need to create a standard set of test stimuli and compute the cell responses for these stimuli (Step 1). In the example of Figure 18 the testing stimuli are a subset of the patterns used to train the network. The variable 'stimcount' controls the manner in which the patterns are presented to the network. For 'stimcount' equal to 0 the system tests the cortical cells

by presenting the patterns to both eyes simultaneously. For 'stimcount' equal to 1 the system presents the patterns to the left eye and nothing to the right eye, while for 'stimcount' equal to 2 the system presents the patterns to the right eye and nothing to the left eye. When 'stimcount' equals 3, the system leaves step 1 and passes the raw data stored in variable 'analysis_data' to the routine 'Do_statistics', which comprises step 2. In practice, one may want to avoid the storage of all the raw data, by updating the statistics as each test stimulus is presented.

In this particular routine the measures of interest are selectivity, ocular dominance, and facilitation, which are defined within the routine (Figure 19c). The statistical moments that the routine calculates are the mean and variance for each measure and the correlations between selectivity and ocular dominance, as well as facilitation and binocularity. For each value of 'eyetest', or more generally of 'stimcount', the routine must determine several quantities: the pattern which maximally drives each cortical cell and the value of the maximal response, the average response of each cortical cell over all stimuli in the test set, and the response of each cortical cell when the pattern that drives the cell best binocularly is presented to one eye at a time. Using these quantities, the routine is able to calculate the measures of interest and the statistical moments. This completes step 2.

To finish the analysis the data must be sent to the output devices and files. This section of the routine can be complicated, particularly if a graphics device is used to display the output data. In this example the routine simply writes the results to the terminal in tabular form (Figure 19d).

The test session and analysis shown here is entirely deterministic. Stochastic testing is also of great interest. Noise can be injected into the system in a number of places (the input, the synapses, the response) and the

variability of the response can be studied. Unfortunately, such testing requires even more run-time and analysis.

Discussion

Once the simulation is coded, debugged, and compiled, test runs should be performed to assess the most basic behaviors. For instance, does the system remain stable after sufficient experience in a stable environment? Do the responses change as expected under monocular rearing? In order to obtain satisfactory results from these basic simulations, adjustments will need to be made to parameters. The numbers of afferents and cortical cells, the initial synaptic weights, the amplitudes of the inputs, the form and values of the function f in equation 1, the parameters in the modification rules (equations 2 and 3), and the sequence of stimuli must be played with until a consistent set of parameters becomes evident. There is no good substitute for experience with observing the direct effects of altering these various inputs.

However, once a rough feeling is obtained for when the simulation runs reasonably, a great deal of frustration can be avoided by varying the large parameter set in a systematic manner. Since a typical program potentiallly contains 20 or 30 parameters, preliminary considerations obtained from early runs or analysis of the model should be used to exclude certain of these parameters from further analysis by virtue of their lack of significance for the eventual results. Those parameters which may affect the output should then be varied around their nominal settings, as estimated by the preliminary runs.

The analysis of the system output as a function of the parameters is termed "sensitivity analysis". By examining the results of a large number of runs at different parameter values, one can establish the sensitivity of the output

variables to each parameter, and at the same time optimal values for the parameters can be located. Cukier et al (ref. 2) developed a method of sensitivity analysis which applies to our simulations. A large number of runs are required, but fewer than with a brute force method of changing one parameter at a time while others are fixed. Their method varies each parameter sinusoidally at different, independent frequencies, then Fourier analyzes the system output back into each of the independent frequencies. The power at a given frequency and its harmonics indicates the sensitivity of the output to the associated parameter. Unfortunately, the range over which a parameter is varied influences the measure of sensitivity obtained, since varying a parameter over a small range will result in less variation of the output than with a larger range. Inspection of the data obtained from the many runs allows one's judgments of the nominal ranges to be improved, along with improving the nominal values upon which these ranges should be centered.

As an example of the usefulness of sensitivity analysis, consider the dependence of orientation selectivity on some of the parameters of our simulation. In this model, we begin with an initial state of low selectivity, and with appropriate experience selectivity increases to some asymptotic level. Our measure of selectivity takes values between 0 and 1, with 0 corresponding to a flat tuning curve, while 1 corresponds to a tuning curve which is infinitely narrow (i.e. a delta function). We discussed above how we code visual stimuli which represent an abstraction of oriented bars, and pointed out that the important parameter in these stimuli is the overlap between different stimulus vectors. The parameter 'width_of_input' in Figure 12 determines this overlap. If the overlap is large, the different stimuli will look alike to the cortical cells, which should therefore show less selectivity than if the overlap is smaller and the stimuli are more easily distinguished. By performing a

sensitivity analysis, one can investigate the extent to which selectivity depends on this input tuning, compared to the dependence on other parameters such as the synaptic modification time constants. Furthermore, one series of runs informs the investigator not only about the asymptotic level of selectivity, but about the progression of the sensitivity with experience. Early in a run, selectivity might be found to depend on the excitatory modifications, while later the inhibitory modifications become more important, and eventually the asymptotic level of selectivity may be sensitive to the width of the input tuning as well as the modification speeds. Of great importance in this analysis is the discovery of parameters to which the output is insensitive. If, for example, the system is insensitive to the parameters involved in the inhibitory processes, these processes contribute little to the operation of the model.

Once the very hard job of finding appropriate parameters is completed, the simulation can be run under various rearing conditions to show that experimental results can be replicated (at least in a very limited, abstract context!) and novel rearing conditions can be used to "predict" the results of yet-to-be-performed experiments.

The main value of running simulations, we feel, is not to pretend that one is imitating the nervous system in any detail, but instead to gain an understanding of some of the concepts which may someday help to model the physiology in detail. For instance, the use of various forms of Hebbian synapses enables very powerful computations to be performed by parallel networks, with little need for prior organization. Simulations aid in appreciating both the power and the limitations of the concept of self-organization by synaptic modification. For example, one learns quickly to pay attention to input coding, which is practically ignored in most models prior to their realization in simulations. Whereas a model of central processes might assume that the

peripheral nervous system copies external stimuli faithfully, a simulation might assume that the central nervous system receives inputs which are all but completely processed.

Simulations of neural systems serve to develop applications in artificial intelligence. One such application might be in the programming of parallel computers. A neural network with modifiable synapses provides one of several potential architectures for parallel hardware, and knowledge of the behavior of these systems will allow future machines to be used to not only simulate neural models at high speeds, but also in applications where continuous adaptive behavior or associative recall of large databases is required. The code described in this article was constructed to simulate a self-organizing system, where no external information about the system output is available. However, this program can be adapted to run a supervised learning system, where the desired output is fed to the synaptic modification algorithm as a goal (see ref. 7). The error between the actual output (R) and the desired output can then be used to modify synaptic weights. Such an error-correcting method allows the system to be taught to respond to given inputs with outputs which are determined in advance. The system can be taught to discriminate between slightly different inputs, or to categorize all inputs which are similar to some prototype as a given class. Although the above discussion concerns sensory processing primarily, one could substitute a motor system's anatomy in order to develop effector instruments.

SUMMARY

In our previous article (ref. 8) we discussed models of the development of visual cortical specificity. The behavior of such a model can be simulated on a serial

computer by stepping through a sequence of stimulus presentations which mimic visual experience. The simulated experience controls the development of cortical responses through an algorithm which changes the simulated synaptic weights. Such simulations provide a means to study models, and should be regarded more on an abstract level than as simulations of real neural processes.

References

1. Blasdel, G.G.; Pettigrew, J.D.: Degree of interocular synchrony required for maintenance of binocularity in kitten's visual cortex. J. Neurophysiol. 42:1692-1710; 1979.

2. Cukier, R.I.; Levine, H.B.; Shuler, K.E.: Nonlinear sensitivity analysis of multiparameter model systems. J. Comp. Physics 26:1-42; 1978.

3. Daniels, J.D.; Saul, A.B.: Modelling and simulation I: Introduction and guidelines. J. Electrophysiol. Tech. 13:95-109; 1986.

4. Grogonio, P. Programming in PASCAL. Reading, Mass: Addison-Wesley; 1980:117.

5. Hebb, D.O. The Organization of Behavior. New York: Wiley; 1949.

6. Knuth, D.E. The Art of Computer Programming, Vol. 2: Seminumerical Algorithms. Reading, Mass: Addison-Wesley; 1969.

7. Rumelhart, D.E.; McClelland, J.L. (eds.) Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1: Foundations. Cambridge, Mass: Bradford Books/ MIT Press; 1986.

8. Saul, A.B.; Daniels, J.D.: Modeling and simulation II: Specificity models for visual cortex development. J. Electrophysiol. Tech.

Figure Captions

Figure 1:

One way to generate input stimuli is to project bars onto a sheet of receptor elements. The activity of each receptor is proportional to the degree to which that receptor is covered by the bar.

Figure 2:

Examples of abstract stimuli. The curves are gaussians on a circle (see 'FUNCTION contour' in Figure 12). Two stimuli are shown, centered on 0 (solid) and 45 (dashed). On the left, the curves are displayed in polar form, with amplitude given by the radial distance from the circle. On the right, the circle is unwrapped to a cartesian plot of amplitude versus orientation (or input fiber). The overlap between two stimuli depends on the distance between their centers and on the width of the curves.

Figure 3:

A flow chart for a simulation. The program consists of a loop which simulates the passage of time. Following initialization of variables, for each iteration of the time loop an appropriate input is chosen. This input induces a response, which often must be computed iteratively. Display of this input/output relationship may be useful, and we indicate this by the "graphics" box. At certain points in the time loop the simulation is interrupted for analysis of the system. The analysis routine can use the same stimulus-generation and response-computation procedures, by setting a standard sequence of test stimuli. During analysis the system is left unaltered, but during the main simulation sequence, synaptic strengths are modified based on the stimulus and response.

Data is written to output devices when desired, generally before each time step.

Figure 4:

The driver for the simulation. When the driver calls the routine 'Choose_stimulus', the variables 'time', random_seed', and 'stimulus' pass explicitly. Notice that 'PROCEDURE Choose_stimulus' changes the value of 'stimulus'.

Figure 5:

Routine for determining rearing conditions. The 'CASE (random_seed MOD 10) OF' statement calls the routine that is numbered by 'random_seed MOD 10'. For example, if random_seed equals 128, then (random_seed MOD 10) equals 8 and the CASE OF statement calls 'PROCEDURE Disparate'. The variables 'time' and 'random_seed' must be specified before entering 'PROCEDURE Choose_stimulus'. 'PROCEDURE Choose_stimulus', however, assigns a value to the variable 'stimulus' and passes this new value back to the main driver of the program.

Figure 6:

Record variable for storing visual stimulus conditions. The variable 'stimulus' contains a 'tag' component, which takes on values of the type 'rearing', and for each value of the 'tag' component several other variables. For example, 'tag = md', 'open = right', and 'pattern = 5' specifies the 'tag' component of 'stimulus', i.e. we are looking at the 'md' component of 'stimulus', and the values of the variables for 'tag = md', i.e. 'stimulus.open = right' and 'stimulus.pattern = 5'.

Figure 7:

Routines for defining rearing conditions.

Figure 8:

Routines for defining rearing conditions. The routine 'Procedure Disparate' uses a crude method to generate disparities between the left and right eye stimuli. The function 'random' returns a number between 0 and 1, which is then used to generate 'dispshift' which lies between -2 and +2, which finally leads to the difference between 'leftangle' and 'rightangle' (the disparity) lying between -4 and +4. However, the distribution of 'dispshift' is not uniform, so that disparities tend to be between -2 and +2.

Figure 9:

Routines for defining rearing conditions.

Figure 10:

Translation of rearing conditions into stimulus vector. The main driver of the program passes 'stimulus' and 'afferent_activity' explicitly to 'PROCEDURE Code_afferent_input'. The value of 'stimulus' determines the pattern types that are assigned to the vector 'afferent_activity'. In this routine we have two loops. The first (second) pass through the outer loop assigns activities to the components of 'afferent_activity' corresponding to the left (right) eye. The inner loop calculates the activity assigned to each component by calling 'FUNCTION afferent_component' which uses the tag value of 'stimulus' to set the activity. Notice that 'FUNCTION afferent_component' returns a numerical quantity to 'PROCEDURE Code_afferent_input'. In general, PASCAL procedures change variable values, whereas PASCAL functions perform some calculation and return the result of the calculation as the value of the function.

Figure 11:

Function which runs through conditions for each type of stimulus.

Figure 12:

Functions to set activity in afferent fibers.

Figure 13:

Iterative computation of cortical activity. Variable 'used' is a vector of dimension N, the number of simulated cortical cells. Each component of 'used' carries a value of TRUE or FALSE, where TRUE (FALSE) indicates that the procedure has (has not) calculated the afferent input to the corresponding cell. The statement 'used := all_false' sets all of the components to FALSE. When the REPEAT loop has updated each cell at least once, all the components of 'used' are TRUE and the implicit updating ends.

Figure 14:

Inner product routines.

Figure 15:

Functions used in response calculations.

Figure 16:

Synaptic modification routines. Notice that modifying the synapses takes very little code. Each pass through the loop of 'PROCEDURE Modify_synapses' modifies the synapses of one cell by calling 'PROCEDURE Modify' which changes the synaptic weights of the cell one by one.

Figure 17:

Analysis driver.

Figure 18:

Analysis routine.

Figure 19:

Subroutine for analysis. a) initializations; b) preliminary calculations; c) statistics; d) writing output.

Figure 1

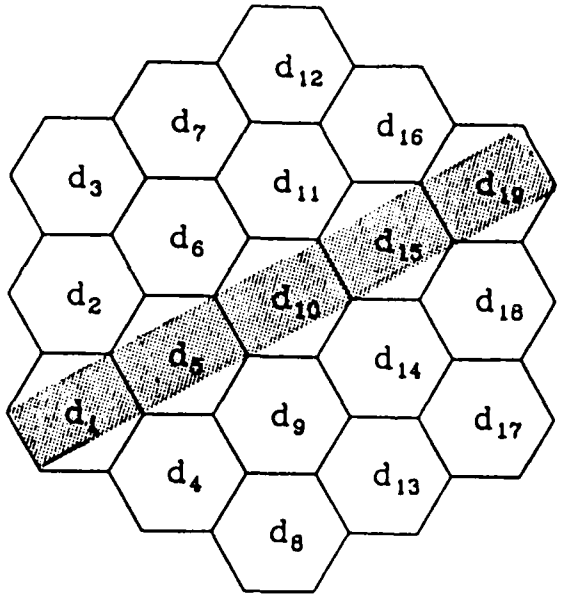


Figure 2

INPUT TUNING

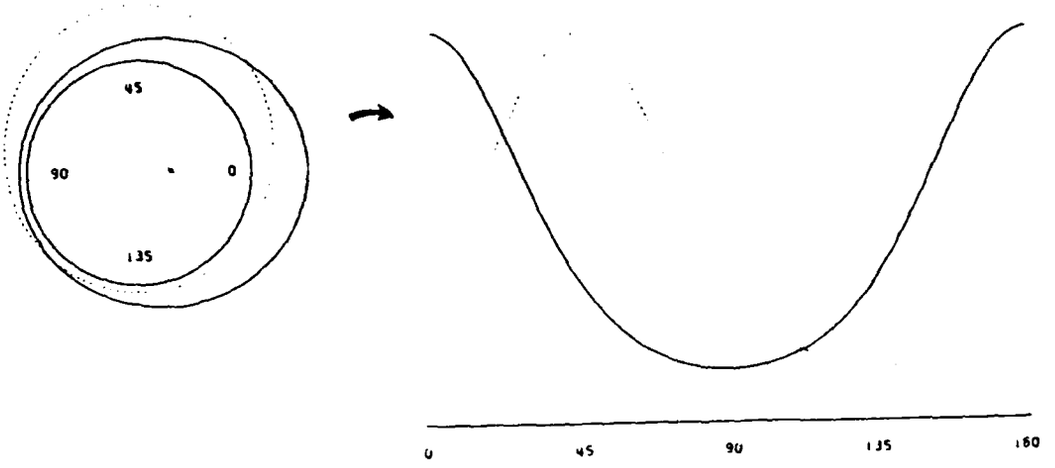
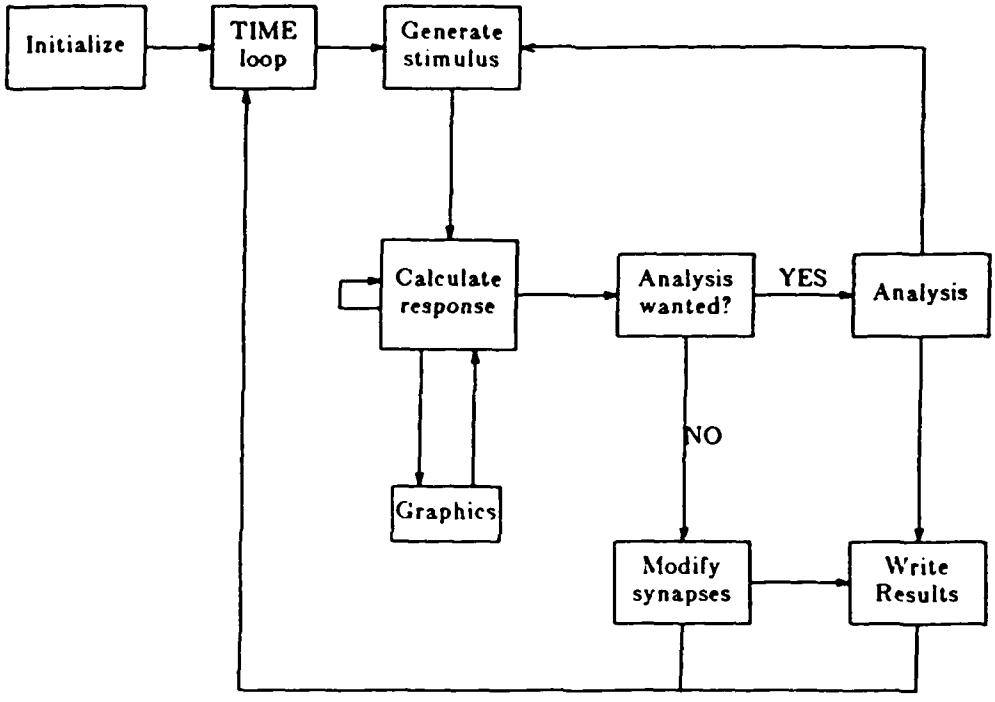


Figure 3


Initialize
TIME loop
Generate stimulus
Calculate response
Analysis wanted?
YES
Analysis
NO
Graphics
Modify synapses
Write Results

```
FOR time := 0 TO end_of_time DO
  BEGIN
    Choose_stimulus (time,random_seed,stimulus);
    Code_afferent_activity (stimulus,afferent_activity);
    Compute_response (afferent_activity,synapses,response);
    Modify_synapses (afferent_activity,response,synapses);
    IF time IN analysis_times THEN Analysis (synapses);
    Log_results_to_output_devices
  END;
```

```
PROCEDURE Choose_stimulus (time:times; random_seed:INTEGER;
                           VAR stimulus:stimulus_type);

    BEGIN
      IF time < 1000 THEN Dark (random_seed,stimulus)
                     ELSE
                       CASE (random_seed MOD 10) OF

                         0,1 : Dark (random_seed,stimulus);
                         2 : Closeleft (random_seed,stimulus);
                         3 : Closeright (random_seed,stimulus);
                         4 : Strabismic (random_seed,stimulus);
                         5,6,7,8,9 : Disparate (random_seed,stimulus)

                       END
    END;
```

```
stimulus_type = RECORD
                  CASE tag : rearing OF

                    correlated : (orientation : angle);
                    md : (open : eyes; pattern : angle);
                    rs : (firstopen : eyes; reversal : times;
                                            bar : angle);
                    dr : ();
                    disparity : (rightangle,leftangle : angle);
                    strabismus : (rtangle,ltangle : angle);
                    adaptation : (adapted_eye : eyes;
                                  adapting_pattern : angle);
                    periodic :  (frequency : real; phase : angle)

                  END;
```

```
PROCEDURE Normalrearing (random_seed : INTEGER;
                         VAR stimulus : stimulus_type);

    BEGIN
      WITH stimulus DO
        BEGIN
          tag := correlated;
          orientation := 1 + (random_seed MOD numangles)
        END
    END; {Normalrearing}


PROCEDURE Closeleft (random_seed : INTEGER;
                     VAR stimulus : stimulus_type);

    BEGIN
      WITH stimulus DO
        BEGIN
          tag := md;
          open := right;
          pattern := 1 + (random_seed MOD numangles)
        END
    END; {Closeleft}


PROCEDURE Closeright (random_seed : INTEGER;
                      VAR stimulus : stimulus_type);

    BEGIN
      Closeleft (random_seed,stimulus);
      stimulus.open := left
    END; {Closeright}


PROCEDURE Reverse_suture (random_seed : INTEGER;
                          VAR stimulus : stimulus_type);

    BEGIN
      WITH stimulus DO
        BEGIN
          tag := rs;
          firstopen := right;
          reversal := 2000; {or whatever time is desired}
          bar := 1 + (random_seed MOD numangles)
        END
    END; {Reverse_suture}
```

```
PROCEDURE Dark (random_seed : INTEGER;
                VAR stimulus : stimulus_type);

    BEGIN
      stimulus.tag := dr
    END; {Darkrearing}


PROCEDURE Disparate (random_seed : INTEGER;
                     VAR stimulus : stimulus_type);

  CONST
    dispwid = 3.2;
    disptrans = 1.6;

  VAR
    dispshift : INTEGER;

    BEGIN
      WITH stimulus DO
        BEGIN
          tag := disparity;
          rightangle := 1 + (random_seed MOD numangles);

          IF rightangle > numangles DIV 2
                          { Only half the patterns (the "vertical" ones)}
                          {  induce disparities.                        }

           THEN
            BEGIN
              dispshift := ROUND(dispwid * random(random_seed) - disptrans);
              leftangle := 1 + (rightangle + numangles - 1 + dispshift)
                                          MOD numangles;
              dispshift := ROUND(dispwid * random(random_seed) - disptrans);
              rightangle := 1 + (rightangle + numangles - 1 + dispshift)
                                          MOD numangles
            END
           ELSE
              leftangle := rightangle
        END
     END; {Disparate}
```

```
PROCEDURE Strabismic (random_seed : INTEGER;
                      VAR stimulus : stimulus_type);

    BEGIN
      WITH stimulus DO
        BEGIN
          tag := strabismus;
          rtangle := 1 + (random_seed MOD numangles);
          ltangle := 1 + TRUNC(numangles * random(random_seed))
        END
    END; {Strabismic}


PROCEDURE Adapting (random_seed : INTEGER;
                    VAR stimulus : stimulus_type);

    BEGIN
      WITH stimulus DO
        BEGIN
          tag := adaptation;
          adapted_eye := right;
          adapting_pattern := 17
        END     {This is an example.  Any stimulus can be fixed here.}
    END; {Adapting}


PROCEDURE Sinusoid (random_seed : INTEGER;
                    VAR stimulus : stimulus_type);

    BEGIN
      WITH stimulus DO
        BEGIN
          tag := periodic;
          phase := 1 + (random_seed MOD numangles);
          frequency := two_pi_over_n * (random_seed MOD 10)
        END
    END; {Sinusoid}   { two_pi_over_n = (2 * PI) / numangles }
```

```
PROCEDURE Code_afferent_input (stimulus : stimulus_type;
                               VAR afferent_activity : maxvector);

  VAR                              { Numafferents equals the total number }
    fiber,                         {  of afferent fibers to each cell.    }
    fiber_number : maxindex;       { maxindex = 1..numafferents           }
    left_or_right : eyes;          {          or 1..numcells              }
                                   { depending on which is larger.        }
    BEGIN                          { maxvector = ARRAY[maxindex] OF REAL  }
      fiber_number := 0;
      FOR left_or_right := left TO right DO
        FOR fiber := 1 TO (numafferents / 2) DO
                        { We take                                         }
                        {     numangles = numafferents / 2.               }
                        { That is, each pattern (or angle) corresponds    }
                        {  to an afferent fiber from each eye.            }
          BEGIN
            fiber_number := fiber_number +1;
            afferent_activity[fiber_number] :=
               afferent_activity_component(left_or_right,fiber,stimulus)
          END
    END; {Code_afferent_input}
```

```
FUNCTION afferent_component (left_or_right : eyes;
          fiber : maxindex; stimulus : stimulus_type) : REAL;

  VAR
    x: REAL;
    timerev,lropen : BOOLEAN;

    BEGIN
      WITH stimulus DO
        CASE tag OF

          correlated :
            x := contour(fiber,orientation);

          md :
            IF left_or_right = open
              THEN x := contour(fiber,pattern)
              ELSE x := noise(noise_amplitude);

          rs :
            BEGIN
              timerev := time < reversal;
              lropen := left_or_right = firstopen;

              IF (NOT timerev OR lropen) AND (timerev OR NOT lropen)
                                {timerev IFF lropen}
                THEN x := contour(fiber,bar)
                ELSE x := noise(noise_amplitude)
            END;

          dr :
            x := noise(noise_amplitude);

          disparity :
            IF left_or_right = left
              THEN x := contour(fiber,leftangle)
              ELSE x := contour(fiber,rightangle);

          strabismus :
            IF left_or_right = left
              THEN x := contour(fiber,ltangle)
              ELSE x := contour(fiber,rtangle);

          adapting :
            IF (adapted_eye = both) OR (left_or_right = adapted_eye)
              THEN x := contour(fiber,adapting_pattern)
              ELSE x := noise(noise_amplitude);

          periodic:
                  x := (1+COS(frequency*(fiber-phase)))/2

        END;
      afferent_component := x
    END; {afferent_component}
```

```
FUNCTION contour (fiber : maxindex; center_angle : angle) : REAL;

  CONST                          { Plotting fiber activity versus fiber    }
    width_of_input = 1.0;        {  number gives a Gaussian curve with the }
                                 {  peak value occurring at 'center_angle'.}
                                 {            (see Figure 2)               }
    BEGIN
      contour := EXP(-width_of_input *
                     distance_squared(fiber,center_angle))
    END; {contour}


FUNCTION distance_squared (fiber : maxindex; center_angle : angle) : REAL;

                     { This function calculates the distance between }
                     {  points lying on a unit circle.               }
    BEGIN
      distance_squared := 1.0 - COS(two_pi_over_n * (fiber-center_angle))
    END; {distance_squared}
                               { two_pi_over_n = 2 * PI / numangles }


FUNCTION noise (amplitude : REAL) : REAL;

    BEGIN
      noise := amplitude * random (random_seed)
    END; {noise}


FUNCTION random (VAR random_seed : INTEGER) : REAL;

        {random is uniformly distributed between 0 and 1}

  CONST
    alpha = 779;           { See refs. 4,5 on the generation }
    lambda = 361;          {  of random variates.           }
    pea = 16384;

    BEGIN
      random := random_seed/pea;
      random_seed := (alpha * random_seed + lambda) MOD pea
    END; {random}
```

```
PROCEDURE Compute_response (afferent_activity : maxvector; synapses : synapses_type;
                                                    VAR response : maxvector);

  VAR
    used : ARRAY [cellindex] OF BOOLEAN;        { cellindex = 1..numcells }

    BEGIN
      used := all_false;
      REPEAT
        cell := random_integer_between(1,numcells);
        IF afferent_input_hasnt_already_been_computed(cell)
          THEN Compute_afferent_input;
        used[cell] := TRUE;
        Compute_intracortical_input;
        response[cell] :=
            sigmoid(afferent_input[cell] - intracortical_input[cell]);
      UNTIL used = all_true
    END; {Compute_response}
```

```
PROCEDURE Compute_afferent_input;

    BEGIN
      afferent_input[cell] :=
          inner_product(afferent_activity,synapses.afferent[cell],numafferents)
    END;


PROCEDURE Compute_intracortical_input;

    BEGIN
      intracortical_input[cell] :=
          inner_product(response,synapses.intracortical[cell],numcells)
    END;


FUNCTION inner_product (x,y : maxvector; lastindex : maxindex) : REAL;

  VAR
    ip : REAL;
    index : maxindex;

    BEGIN
      ip := 0.0;
      FOR index := 1 TO lastindex DO
        ip := ip + x[index] * y[index];
      inner_product := ip
    END;
```

```
FUNCTION random_integer_between (a,b : INTEGER) : INTEGER;

  VAR
    interval : INTEGER;

    BEGIN
      interval := b-a+1;
      random_integer_between :=
          a + ROUND(interval * random(random_seed)) MOD interval
    END;


FUNCTION  sigmoid (x : REAL) : REAL;

  CONST
    threshold = 3.0;
    steepness = 2.2;
    too_small = -20.0;
    too_big = 10.0;

    BEGIN
      IF steepness*(threshold - x) < too_small
        THEN sigmoid := 1.0
        ELSE IF steepness*(threshold - x) > too_big
               THEN sigmoid := 0.0
               ELSE sigmoid := 1.0/(1.0 + EXP(steepness * (threshold - x)))
    END;
```

```
PROCEDURE Modify_synapses (afferent_activity,response : maxvector;
                           VAR synapses : synapses_type);

    BEGIN
      FOR cell := 1 TO numcells DO
        BEGIN
          Modify (numafferents,afferent_activity,response[cell],afferent_parameters,
                  synapses.afferent[cell]);

          Modify (numcells,response,response[cell],intracortical_parameters,
                  synapses.intracortical[cell])
        END
    END; {Modify_synapses}


PROCEDURE Modify (lastindex : maxindex; presynaptic : maxvector;
                  postsynaptic,"parameters" : REAL;
                  VAR junctions : maxvector);

    BEGIN
      FOR index := 1 TO lastindex DO
        junctions[index] := "the appropriate function of" (6)
                            (presynaptic,postsynaptic,parameters)


(e.g. junctions[index] := (1.0 - h) * junctions[index]
                          + h * presynaptic[index] * postsynaptic  )
    END; {Modify}
```

```
PROCEDURE Analysis (synapses : synapses_type);

    BEGIN
      initialize output variables
      REPEAT                                                { Step 1 }
        choose stimulus
        make input vector
        compute response
        log response into output variables
      UNTIL all desired patterns have been tested;
      compute averages and other output functions           { Step 2 }
      write results and draw graphics                       { Step 3 }
    END;
```

```
TYPE
  cell_analysis : ARRAY[eyes,cellindex,angle] OF REAL;

VAR
  stimcount : INTEGER;
  yes : BOOLEAN;
  eyetest : eyes;
  orientest : angle;
  analysis_data : cell_analysis;

  BEGIN
    askforanalysis(time,yes);     { 'PROCEDURE Analysis' is used only when     }
                                  {   the variable 'time' equals a time        }
    IF yes THEN                   {   specified in 'PROCEDURE askforanalysis', }
      BEGIN                       {   which we omit here.                      }
        stimcount := 0;

        REPEAT
          BEGIN
            FOR orientest := 1 TO numangles DO
              BEGIN
                WITH stimulus DO
                  BEGIN
                    IF stimcount = 0
                      THEN tag := correlated
                      ELSE
                        IF stimcount < 3
                          THEN tag := md;
                    CASE tag OF
                      correlated :
                        BEGIN
                          orientation := orientest;
                          eyetest := both
                        END;
                      md :
                        BEGIN
                          pattern := orientest;
                          IF stimcount = 1
                            THEN
                              open := left
                            ELSE
                              open := right;
                          eyetest := open;
                          noiseamplitude := 0.0
                        END
                    END {CASE tag}
                  END; {WITH stimulus}
                Code_afferent_activity (stimulus,afferent_activity);
                Compute_response (afferent_activity,synapses,response);
                FOR cell := 1 TO numcells DO
                  analysis_data[eyetest,cell,orientest] := response[cell];
              END; {FOR orientest}
            stimcount := stimcount + 1
          END {REPEAT}
        UNTIL (stimcount = 3);

        Do_statistics(analysis_data);
      END {yes}
  END; {Analysis}
```

```
PROCEDURE Do_statistics(analysis_data : cell_analysis);

  CONST
    fldwd = 7;
    prec = 2;

  TYPE
    analyint = ARRAY[eyes,cellindex] OF INTEGER;
    analydata = ARRAY[eyes,cellindex] OF REAL;
    overcells = ARRAY[eyes] OF REAL;

  VAR
    eyetest : eyes;
    orientest : angle;
    best_stimulus : analyint;
    max,mean,sel,resp_at_binoc_max : analydata;
    ocdom,facilitation,responsiveness : maxvector;
    mean_sel,weighted_mean_sel,variance_sel : overcells;
    mean_ocdom,weighted_mean_ocdom,variance_od,
      mean_facil,weighted_mean_facil,variance_fac,
        mean_responsiveness,
          correlation_sel_od,correlation_fac_binocularity : REAL;

  BEGIN

    { initializations:  }

     FOR eyetest := left TO both DO
       BEGIN
         FOR cell := 1 TO numcells DO
           BEGIN
             max[eyetest,cell] := 0.0;
             mean[eyetest,cell] := 0.0;
             responsiveness[cell] := 0.0
           END; {FOR cell}
         mean_sel[eyetest] := 0.0;
         weighted_mean_sel[eyetest] := 0.0;
         variance_sel[eyetest] := 0.0;
       END; {FOR eyetest}
     mean_ocdom := 0.0;
     weighted_mean_ocdom := 0.0;
     variance_od := 0.0;
     mean_facil := 0.0;
     weighted_mean_facil := 0.0;
     variance_fac := 0.0;
     mean_responsiveness := 0.0;
     correlation_sel_od := 0.0;
     correlation_fac_binocularity := 0.0;

    { End of initializations }
```

```
FOR cell := 1 TO numcells DO
  BEGIN

    FOR orientest := 1 TO numcells DO
      BEGIN
        IF max[both,cell] < analysis_data[both,cell,orientest]
         THEN
          BEGIN
            max[both,cell] := analysis_data[both,cell,orientest];
            best_stimulus[both,cell] := orientest
          END; {IF max}
      END; {orientest}

    FOR eyetest := left TO right DO
      BEGIN
        FOR orientest := 1 TO numcells DO
          BEGIN
           IF max[eyetest,cell] < analysis_data[eyetest,cell,orientest] THEN
             BEGIN
               max[eyetest,cell] := analysis_data[eyetest,cell,orientest];
               best_stimulus[eyetest,cell] := orientest
             END; {IF max}
           IF orientest = best_stimulus[both,cell]
             THEN resp_at_binoc_max[eyetest,cell] :=
                                    analysis_data[eyetest,cell,orientest];
           mean[eyetest,cell] := mean[eyetest,cell]
                                 + analysis_data[eyetest,cell,orientest]
          END {FOR orientest}
      END {FOR eyetest}

  END; {FOR cell}
```

```
    BEGIN
      FOR eyetest := left TO both DO
        BEGIN
          mean[eyetest,cell] := mean[eyetest,cell]/numangles;
      { Definition: }
          sel[eyetest,cell] := 1.0 - mean[eyetest,cell]/max[eyetest,cell];
          mean_sel[eyetest] := mean_sel[eyetest] + sel[eyetest,cell];
          IF responsiveness[cell] < max[eyetest,cell]
            THEN responsiveness[cell] := max[eyetest,cell]
        END; {FOR eyetest}

      FOR eyetest := left TO both DO
        weighted_mean_sel[eyetest] := weighted_mean_sel[eyetest] +
              sel[eyetest,cell] * responsiveness[cell];
      { Definition: }
      ocdom[cell] := max[right,cell]/(max[left,cell] + max[right,cell]);
      mean_ocdom := mean_ocdom + ocdom[cell];
      weighted_mean_ocdom := weighted_mean_ocdom +
                                ocdom[cell] * responsiveness[cell];
      variance_od := variance_od + SQR(ocdom[cell]);
      { Definition: }
      facilitation[cell] := resp_at_bin_max[both,cell] /
                (resp_at_bin_max[left,cell]+resp_at_bin_max[right,cell]);
      mean_facil := mean_facil + facilitation[cell];
      weighted_mean_facil := weighted_mean_facil +
                               facilitation[cell] * responsiveness[cell];
      variance_fac := variance_fac + SQR(facilitation[cell]);
      mean_responsiveness := mean_responsiveness + responsiveness[cell]
    END; {FOR cell}

  FOR eyetest := left TO both DO
    BEGIN
      mean_sel[eyetest] := mean_sel[eyetest]/numcells;
      weighted_mean_sel[eyetest] :=
            weighted_mean_sel[eyetest]/mean_responsiveness
    END; {FOR eyetest}

  mean_ocdom := mean_ocdom/numcells;
  weighted_mean_ocdom := weighted_mean_ocdom/mean_responsiveness;
  mean_facil := mean_facil/numcells;
  weighted_mean_facil := weighted_mean_facil/mean_responsiveness;
  mean_responsiveness := mean_responsiveness/numcells;

  FOR eyetest := left TO both DO
    variance_sel[eyetest] :=
     (variance_sel[eyetest] - numcells * SQR(mean_sel[eyetest]))/(numcells-1);
  variance_od := (variance_od - numcells * SQR(mean_ocdom))/(numcells-1);
  variance_fac := (variance_fac - numcells * SQR(mean_facil))/(numcells-1);

  FOR cell := 1 TO numcells DO
    BEGIN
      correlation_sel_od := correlation_sel_od +
                (sel[right,cell]-mean_sel[right]) * (ocdom[cell]-mean_ocdom);
      correlation_fac_binocularity := correlation_fac_binocularity
                + (mean_facil-facilitation[cell]) * abs(ocdom[cell] - 0.5)
    END; {cell}

  correlation_sel_od := correlation_sel_od/SQRT(variance_sel * variance_od);
  correlation_fac_binocularity := correlation_fac_binocularity /
                                    SQRT(variance_fac * variance_od);
```

```
    writeln;
    writeln('selectivity':23,'od':9,'fac':8,'resp':7,
                                                'pref pattern':23);
    write('cell   left  right  binoc   ':29);
    writeln('    left     right      binoc':49);

    FOR cell := 1 TO numcells DO
     BEGIN
      write(cell:5,sel[left,cell]:fldwd:prec);
      write(sel[right,cell]:fldwd:prec,sel[both,cell]:fldwd:prec);
      write(ocdom[cell]:fldwd:prec,facilitation[cell]:fldwd:prec);
      write(responsiveness[cell]:fldwd:prec);
      write(best_stimulus[left,cell],best_stimulus[right,cell]);
      writeln(best_stimulus[both,cell]);
     END; {cell}

    writeln;
    write('avrgs ':5,mean_sel[left]:fldwd:prec,mean_sel[right]:fldwd:prec);
    write(mean_sel[both]:fldwd:prec,mean_ocdom:fldwd:prec);
    write(mean_facil:fldwd:prec,mean_responsiveness:fldwd:prec);
    writeln;
    write('      ':5,weighted_mean_sel[left]:fldwd:prec);
    write(weighted_mean_sel[right]:fldwd:prec,
                    weighted_mean_sel[both]:fldwd:prec);
    write(weighted_mean_ocdom:fldwd:prec,weighted_mean_facil:fldwd:prec);
    write('    weighted by responsiveness');
    writeln;
    write('   standard deviation of ocular dominance is ',
                                              SQRT(variance_od):10:4);
    writeln;
    writeln;
    write('correlations:  between selectivity in right eye and ':52);
    writeln('shift toward right eye :',correlation_sel_od:fldwd:prec);
    write(' ':15,'between facilitation and binocularity          :');
    writeln(correlation_fac_binocularity:fldwd:prec);
    writeln;
    writeln

END; {Do_statistics}
```

END

2-87

DTIC