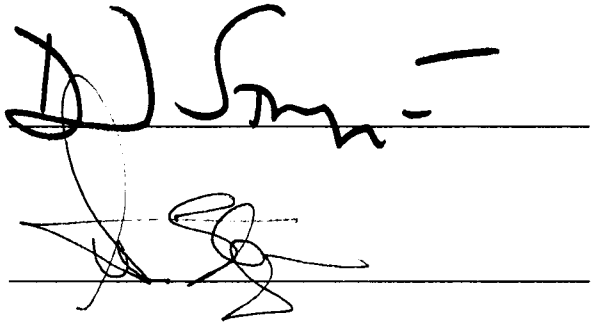
To the Graduate Council:

I am submitting herewith a thesis written by Charles W. Joyce, Jr. entitled "Network Data Structures and Representation in the Simulation of Neural Networks". I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

Bruce J. MacLennan, Major Professor

We have read this thesis and recommend its acceptance:



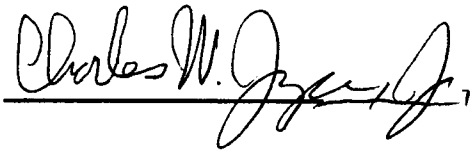
Accepted for the Council:

Vice Provost
and Dean of the Graduate School

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a Master's degree at The University of Tennessee, Knoxville, I agree that the Library shall make it available to borrowers under rules of the Library. Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgment of the source is made.

Permission for extensive quotation from or reproduction of this thesis may be granted by my major professor, or in his absence, by the Head of Interlibrary Services when, in the opinion of either, the proposed use of the material is for scholarly purposes. Any copying or use of the material in this thesis for financial gain shall not be allowed without my written permission.

Signature 

Date August 16, 1996

NETWORK DATA STRUCTURES AND REPRESENTATION IN THE SIMULATION OF NEURAL NETWORKS

A Thesis

Presented for the

Master of Science Degree

The University of Tennessee, Knoxville

Charles W. Joyce, Jr.

December, 1990

ABSTRACT

The purpose of this paper is to examine the issues of network representation and methods of referring to elements of artificial neural networks as they are implemented in three publicly available simulators for such networks. It will be shown that these issues are central to the performance of these simulators, and have other implications as well. A critical comparison of other features of these simulators will also be offered.

The three simulators used in the study are: the Rochester Connectionist Simulator (RCS) from the University of Rochester, the California Institute of Technology's GENESIS simulator, and the SFINX simulator from the University of California at Los Angeles.

The question of network representation arises when a neural network is prepared for simulation. It involves precisely how the network should be described for thew simulator, and what data structures will be used by the simulator to carry out the simulation. A related question involves how the user will refer to the elements in the network in controlling the simulation.

The approach taken will be to provide some background material about each of the simulators under discussion, their history and development, methods of network representation and reference, and their basic resource requirements. Then a test simulation designed to establish the simulator's response to networks of increasing size will be described and its implementation on the simulators will be discussed. Finally, the results of the tests will be examined.

It is hoped that this study may be of interest those who must choose a simulator (either from the public or commercial market places) for use in an educational/pedagogic or research setting.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

Introduction

The purpose of this paper is to examine the issues of network representation and methods of referring to elements of artificial neural networks as they are implemented in three publicly available simulators for such networks. It will be shown that these issues are central to the performance of these simulators, and have other implications as well. A critical comparison of other features of these simulators will also be offered.

The question of network representation arises when a neural network is prepared for simulation. It involves precisely how the network should be described for thew simulator, and what data structures will be used by the simulator to carry out the simulation. A related question involves how the user will refer to the elements in the network in controlling the simulation.

The approach taken will be to provide some background material about each of the simulators under discussion, their history and development, methods of network representation and reference, and their basic resource requirements. Then a test simulation designed to establish the simulator's response to networks of increasing size will be described and its implementation on the simulators will be discussed. Finally, the results of the tests will be examined.

It is hoped that this study may be of interest those who must choose a simulator (either from the public or commercial market places) for use in an educational/pedagogic or research setting.

1.1 Terminology

Research is underway in the properties of neural networks in a wide variety of fields, ranging from the study of neurophysiology to the strictly computational mathematics of network behavior. This diversity of activity is certainly reflected in the variety of terms used to describe the neural network itself. Some authors use the biological terminology to describe the principle element of the network as a 'neuron', whereas others take a more utilitarian view and describe these elements as 'units'. The mathematically inclined may refer to them as 'predicates'. With perfectly adequate reason, and a refreshing sense of humor, a support group established for one of the simulators has been named 'BABEL'.

The intention of this study is not to participate in any particular school of thought with regard to research in this field. It is worth passing comment that the choices of terminology in descriptive terms and commands naturally reflect the preoccupation of the creators of any particular simulator. And the diversity of the field certainly supports this practice. It is difficult, however, to see where the choice of biologically accurate but generally obscure terms communicate effectively with a general user. By the same token, the use of the language of vector notation may indeed be mathematically appropriate, but not contribute substantially to the modeling of a biological neuron. The problems of network reference we wish to examine are defined in a more specific way below, however it is worth noting that a lack of general consensus in the field makes the choice of terminology more difficult for the creators of this type of software.

To attempt to contrast the naming practices in the various simulators this paper will adopt the following general policy with regard to terminology. When speaking of a particular simulator, those terms adopted by the authors of that simulator will be used. In general usage, the terms 'unit', 'neuron', and 'node' may be considered equivalent. The terms 'connection', 'link' and 'synapse' may also be used interchangeably.

Apart from the difficulties associated with the terminology of neural networks, this paper presumes that the reader is at least passingly familiar with the C programming language and the facilities of the Unix operating system.

1.2 Simulators

There are many programs in the public domain for use in research with neural networks. (Appendix A contains a listing of 20 simulators with a brief description of each.) These programs can be classified into three basic groups. A 'demonstration' program is intended to demonstrate a single type of network and a single aspect of that network's behavior. A 'special purpose' simulator is usually intended to allow a broader range of experimentation with a particular type of network. Finally, a 'generalized' simulator supports a broad range of network types. Most of the publically available programs were originally intended to support the teaching and research activities of their authors, and were subsequently released into the public domain.

The need for a generalized simulator for research and modeling of neural networks may not be immediately apparent. Any simulation can be most efficiently provided by a program written to the exact specifications required for that simula-

tion. However, such simulations may be very difficult to modify, requiring redesign of possibly large portions of a program. Further, research with simulations of this kind may, in fact, be bent on discovering things which are not known or suspected in advance. Finally, any useful interaction with a human user will require a certain amount of effort on the part of a simulation programmer, and the use of graphical interfaces may involve substantially more effort than that required by the simulation itself.

The broad range of interest in the various aspects of network behavior includes many disciplines in which researchers do not always have the programming skills which might be required to prepare a complex simulation. Further, the time required for any substantial programming task may be more than some researchers can reasonably spend on a project.

A general neural network simulator may offer a number of advantages in this regard. It may provide a more flexible vehicle for researchers, educators and students who may not wish to invest the time for extensive programming. While it is probably not possible to completely eliminate programming from the process of developing new simulations, the types of programs needed may be made much simpler and smaller by allowing them to function within the context of a simulator environment. A flexible user interface can be provided to allow a consistent and convenient means of interaction for the user, with much less effort than would be required in a special purpose program.

An additional advantage can be found in the ability of the a simulator to be used over a wide variety of platforms. This provides a useful means of communication between researchers, who can exchange not only results, but the actual simulation code so that others may duplicate or extend their work. It is interesting

to note in this regard that several of the simulator creators have developed mechanisms (usually via electronic mail) for the sharing of research and experience. (See Appendix A.)[1]

It is useful to examine some of the other design issues involved in the creation of a generalized simulator in order to more fully understand the decisions made by the implementors of the various simulators. There is an enormous number of neural network models under examination; detailed investigations involving a single neuron, large and complex networks using several kinds of neurons, and also generalized models which may not mention neurons specifically at all. Within a single network model one may find a need for multiple activation functions. Connections may feed forward, require the ability to trace backward along the connections (for back propagation), loop back over one or more layers (recurrent connections), or even move both forward and backward (bidirectional associative memory.) If biological function is being modeled, the precise timing of impulses within the network (propagation delay) may need to be controlled.

The operation of a simulation also depends, to a great extent, upon what results are being examined. What inputs must be provided, what outputs are sought, and what variables must be monitored during the simulation? When and how are parameters and variables modified? Again, a very wide range of possibilities exists. Therefore, a distinct advantage of a simulator is to be able to construct various network models, activation functions, and subroutines in a manner which would allow their use in diverse combinations.

[1] Matt A. Wilson, Documentation for GENESIS and XODUS (Pasadena, California : by the California Institute of Technology, 1989), 1.

Finally, the importance of an efficient, interactive user interface must be considered. While it is possible to simulate network behavior in 'batch' processes, the evaluation of data and the testing of the effect of different parameters on a given simulation is slow and extremely tedious. An interface which provides the user with more immediate feedback from the simulation can save enormous amounts of time in developing a simulation.

1.3 Alternatives in Network Representation

There are two important issues which form the basis for this examination. The first involves the actual choice of data structures used to implement the network simulation and how the network is described by the user. The second involves the means used by the user to refer to portions of a network or simulation. The simulators examined here provide a good contrast in both areas.

The network simulations are generally based on a data structure which represents the neuron, or node. This structure contains some reference to an 'activation function' or procedure for evaluation of the node's input and production of the node's output. One or more lists of other nodes which provide the input (or, in some cases, receive the output) is generally maintained, along with some weight associated with each connection. Additional information may be stored with each node; threshold, current state values, and node names or identifiers are common.

The node structures can be grouped as arrays of nodes, or as a linked list. Each has advantages and disadvantages.

An array of node structures requires only one memory allocation, if the number

of elements is known in advance. If the number is not known, a default amount could be allocated. If this number is not sufficient it is possible, although awkward, to reallocate an array. In fact, it would be necessary to allocate extra empty elements for use in interactive sessions as newly created nodes. Arrays are easily accessed in a random fashion, and require no list pointers as is the case with linked lists. On the other hand, the default allocation of a large array may be wasteful of memory.

If a linked list is used for the nodes, individual allocation of structures may be effective from the standpoint of exact use of memory. However, individual allocations are computationally expensive, and memory may be wasted through problems of alignment or fragmentation. While it is possible to maintain a desired sequence of nodes in a linked list, random access to any given node is more difficult, and various methods to facilitate access generally add to the simulator's burden of pointer variables, which can become important in large networks (over 10,000 nodes, for example.) Finally, the time required to shift between elements in a list is much greater than that required to shift between adjacent array elements. For example, consider that consequtive array elements may be accessed by simply incrementing the element's address. This is a simple operation, which can be performed completely in the registers of a computer's central processing unit. To access the next item in a list, one must access the current list element's pointer to the next list element, then assign that value to the current element. This operation involves an indirect memory reference and fetch. It is therefore subject to the speed with which memory may be accessed on any given machine, and may also be subject to memory swapping in a virtual memory envirionment.

Therefore it would appear that arrays offer several advantages over linked lists

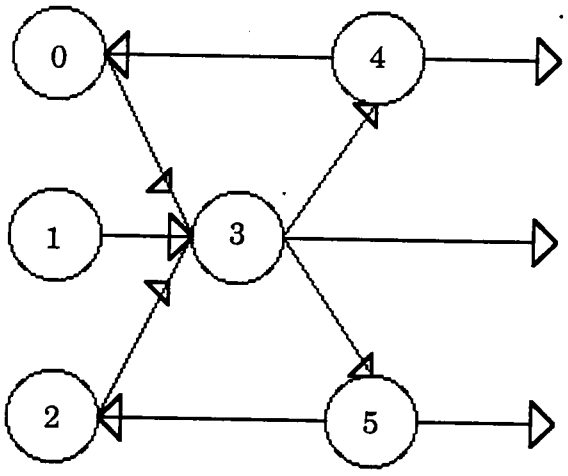
as an overall network data structure. However, it should be pointed out that nodes in a network may have to maintain a variable (possibly varying) number of connections. List structures generally offer advantages in this regard, and it seems common to use some combination of array and list structures in representing an overall network.

The representation used for network data structures will be one of the items examined in more detail in this survey.

While the specific data structures used determine how a program will access the individual nodes in a network, another aspect of the problem is how to allow the human user to refer to the network's individual elements. The use of an index number is certainly a possibility, but not a very desirable one. Networks usually contain groupings of elements. Layers are common features, but other possibilities include groups of units which make up components of a larger network, such as a grouping which performs an exclusive-or function, or chambers in a biological model which make up a specific type of neural cell. A user may wish to refer to such a grouping as a whole, or apply a command to an entire portion of a network. A good reference system should allow the user to address elements (nodes, connections, etc.) individually, in specified or logical groupings, and in other ways. The means used to do this have implications for the amount of information which must be stored in each node, for how the simulator's commands perform (both in terms of efficiency and versatility), and for the system's overall performance. We will also examine this issue for the simulators presented in this study.

There are several problems which must be considered with regard to the data structures used by simulators. Obviously, the structures must be compact and easy to access, and the program must be able to add or delete new units during

the simulation. In addition, the network must be set up in a computationally efficient manner. Finally, there are specific requirements imposed by the type of simulations allowed. For example, in recurrent networks there is a problem similar to the "race" condition in logic design. Consider the following diagram of a simple recurrent network:



Activations feed forward from units 0, 1 and 2 to unit 3; and from unit 3 to units 4 and 5. Activations feed back from unit 4 to unit 0 and from unit 5 to unit 2. An input goes to units 0, 1 and 2 at time t1; the results reach unit 3 at time t2, units 4 and 5 are activated at time t3 and the feedback arrives back at units 0 and 2 at time t4 with new input. A delay of this kind has been found to be useful in recognizing continuous input, such as speech or motion.

If the simulation proceeds node-by-node in numerical order with output made immediately available, the results of the four time steps are accomplished in a single simulation cycle. So long as the next input at t4 does not differ from that

which would have occurred at t2, no problem occurs and the network simulation will perform very efficiently. If, however, steps t2 and t3 present important new information to the network then the feedback due to arrive at t4 will arrive too early. This example also assumes that the units are arranged in the proper sequence for execution, and in an interactive network this may not be the case. If, for example, the numbers on units 3 and 4 are exchanged the network will behave in quite a different manner when a sequential cycle is used. Further, the problem of ordering the units in a recurrent network for proper execution is not trivial.

To further complicate matters, suppose in our example we desired the feedback to arrive at time t6 rather than t4. In the modeling of biological processes this may be necessary to accurately reflect the properties of the items being studied. In the examination of continuous input the amount of delay may be a critical parameter in the operation of the network. The control of this time, called propagation delay, represents an additional requirement of the network structures.

We will comment further on the approaches to this problem as we review the simulators.

CHAPTER 2

The Simulators

In this section we introduce the three simulators used in the study: the Rochester Connectionist Simulator (RCS) from the University of Rochester, the California Institute of Technology's GENESIS simulator, and the SFINX simulator from the University of California at Los Angeles.

These three simulators were chosen as substantial examples of generalized simulators. Each allows the user to create large networks and operate simulations of many types. All allow the user to incorporate new functions and commands into the simulator, and to use interesting methods of network representation and reference. Each simulator also supports a graphical user interface, although these are not considered in the present study.

We will begin with some brief discussion on the history and development of each simulator, proceed to examine the form of network representation and reference employed and conclude the section with a comparison of the simulator commands.

2.1 The Rochester Connectionist Simulator

Currently in version 4.2, RCS was originally implemented in LISP and was gradually translated to the C programming language for performance reasons. The 4.0 version represented a major revision, completed in the Spring of 1987, which in-

cluded a graphical interface for Sun Workstations(SunView). Version 4.1, released in the winter of 1988 contained a number of enhancements in network manipulation and memory management and the introduction of a facility for propagation delays. The current version was released in October of 1989, and includes an interface for X11 and an Apple MacIntosh port of version 4.1, and adopts the license terms of the Free Software Foundation.[1]

The simulator was orginally written by Stephen Small, Lokendra Shastri, Gary Cottrell and Mark Fanty later translated the program from LISP. Nigel Goddard was prepared the latest version, with Michael McInerny providing the X11 interface.[2]

The design objectives are summarized:

> . . . The highest emphasis in the design was placed on flexibility. We believe the connectionist discipline is still in its infancy and that it would be a mistake to settle on one particular model or set of functions. . . . So we've made a very deliberate effort to provide a tool that has a lot of flexibility to begin with as well as the potential to be extended to cover new paradigms and requirements.[3]

The authors of RCS choose to refer to the objects of their simulations as 'connectionist models'. These consist of a number of computational elements called simply 'units' (corresponding to neurons). These elements communicate via 'links' which enter one or more 'sites' on each unit. Each unit, link and site can, if desired,

---

[1]Nigel H. Goddard, Kenton J. Lynne, Toby Mintz and Luidvikas Bukys, Rochester Connectionist Simulator (Rochester,New York : The University of Rochester Computer Science Department, 1989),1-2,3-5. All of what follows is based on this manual, which is supplied with the simulator.

[2]Ibid., 7.

[3]Ibid., 1.

use a different process for dealing with its input. Multiple sites on a single unit allow for the use of multiple methods of processing a unit's input.

The simulator can be used without additional programming by the user, although, the use of user-written functions in the C language permits enormous flexibility. The simulator itself may be called as a subroutine from other programs written in C, LISP, or MIT's Scheme. The data structures and programming of the simulator are straightforward and should be familiar to those acquainted with the C Language.

A network can be set up and modified by simulator commands, although the recommended manner for establishing a network is through a user-written program conventionally named 'build'. The user is provided with a full set of simulator functions for the purpose of unit allocation and the establishment of connections.

The simulator uses a variable type of integer or float, which is designated at the time the source code is compiled. User functions may use a 'typedef' to make functions usable in any form of the simulator. Further, a special compilation of the simulator is required to support propagation delays. This is apparently done for performance, as additional variables in the unit, site and link structures are required along with functions to support their use. The integer version of the simulator contains a 'scaled' method for representing floating point weights with integers, also for improved performance.

Basically, to prepare a network one first allocates an array of units (the number of units may be adjusted 'on the fly', so this need not be done with an exact number.) This array is called the 'unit vector.' Each unit is a data structure which contains several variables (named potential, state and output), a pointer to

a unit function which computes the unit's output, and the head of a linked list of sites. Sites, in turn, hold a value for the processed input, a pointer to the function for processing the input, and the head of a list of links from which the input is taken. Each link contains a pointer to a link function, a weight value, a pointer to the output value of the source unit and the index of that unit in the unit vector.

In addition, each unit, site and link contain a 'generic' data field. This field may be used by the programmer as an additional variable, or as a pointer to a user defined data structure or an additional function.

A unit, site or link function is a standard C language function called with pointers to the specified structure and any superior structures in the hierarchy (a link function receives pointers to the associated site and unit, for example) and operates on data contained in those structures and/or on global variables established by the simulator or user functions.

A simulation cycle basically involves looping through the unit vector, performing each unit's link and site functions, then the unit function. The output of each unit is updated at the conclusion of a complete simulator cycle.

Reference to units in RCS takes several forms. Each unit is assigned an index number (its subscript in the unit vector) which can be used to refer to it in user programs and on the simulator command line. In addition, sites are given names and units are given a character string type description when they are created. Site names are used in commands when units may have more than one site. Units may be given individual names, or named as VECTOR (analogous to a one-dimensional matrix) or ARRAY (a two-dimensional matrix) groups. Thereafter, those units may be refered to by name (possibly subscripted as appropriate.) A range of units

may be addressed by using names and index numbers in any combination.

Units may also be designated as members of one or more sets, which may be operated on by using the common set operations of union, intersection, and so forth.

User programs and functions are compiled and linked into the simulator code, becoming part of the simulator executable. This is accomplished using the Unix 'make' utility.

The simulator may be operated from a command prompt, or from within one of the graphical user interfaces, through the use of a mouse and keyboard entry. In addition, the simulator can read files of simulator commands ('scripts') prepared by the user, or recorded ('logged') by a previous session, which can be advantageous in accomplishing repetitive tasks. The simulator commands are summarized in Table 1, at the end of this chapter.

2.2 GENESIS

The GENESIS (GEneral Network SImulation System) simulator and its companion graphical interface XODUS (X-based Output and Display Utility for Simulators) was written by Matthew A. Wilson, Upindar S. Bhalla, John D. Uhley, David H. Bilitch, Mark E. Nelson and James M. Bower of the Division of Computation and Neural Systems of the California Institute of Technology (Caltech) and

was released as a Beta-test version in January of 1990 incorporating perfor-

mance improvements by Michael D. Speight.[45]

One of the guiding principles behind the design is a desire to facilitate the simulation of interacting processes on many different scales, and in this regard the authors claim that "no currently available simulator (to the best of our knowledge) is flexible and general enough to support complex and detailed models of biological neural networks at many levels of detail." This biological orientation is also reflected in the examples provided with the simulator. Other design objectives include modularity and flexibility ("central to the ability of a simulator to implement models at all levels of detail"), expandability, the use of a 'high level' network specification language "accessible to people with limited programming experience" , user interaction through a flexible 'friendly' graphical user interface, and as efficient an implementation as these other constraints will allow. Finally, concerns for device independence and the ease with which the simulator can be moved to parallel computing architectures are mentioned as important factors.[6]

The GENESIS simulator is object-oriented and written in the C programming language. This is not as contradictory as it may sound, since object-oriented programming is essentially a programming style. Object oriented programming

[4]The authors are named in a file 'AUTHORS' included with the distribution. The words 'Beta Test' appear at the login screen. It was developed beginning in 1987, and has been used for laboratory courses at Caltech and at the Marine Biological Laboratory in Woods Holes, Massachusetts.

[5]Taken from the file 'README_REGISTER' included with the distribution.

[6]Matt A. Wilson, Documentation for GENESIS and XODUS (Pasadena, California : by the California Institute of Technology, 1989), 1-12. This item is only available after payment of a $200.00 fee, at which time it may be acquired electronically via ftp. As before, much of what appears in this discussion is drawn from this document. Specific Quote is from p. 2.

in C is not, perhaps, as simple as it might be in a language such as smalltalk or C++ which have features designed specifically to support the paradigm, but it is possible.

Because of its relevance to this discussion, a brief explanation of object-oriented programming seems appropriate. A program is made by combining various programming 'objects' which each accomplish one or more of the tasks the program is to perform. An 'object' is a specific instance of a 'class' - - a program module which contains data structures and functions which perform some computation, or 'action' on the input to produce an output. The programs, functions or subroutines which perform a given action may be called 'methods'. When an object is instantiated one may define the specific methods and data to be used by that particular object, and in this way objects may be modified to extend their functionality without the need for complete reconstruction (a process known as inheritance), thus reducing the need for changes to other portions of a system which deal with the object.

How one defines, within any given programming context, what 'objects' may be appropriate to use in solving the problem at hand is apparently an open question; however, in theory, the creation of an appropriate class of objects increases the reusability of the class. A class can be chosen to reflect the real-world organization of a problem.[7]

[7]This description is necessarily brief, and is intended merely to provide a basis for the discussion which follows. Further discussion of object oriented programming is beyond the scope of this paper: for a concise description of object oriented programming in C, which strongly resembles the style used in GENESIS, see, David Brumbaugh, "Object-Oriented Programming in C," The C Users Journal, 8 (July 1990) 113-122.

The GENESIS simulator provides several basic classes of modules, and the simulation is built up from objects instantiated from these classes. Certain classes provide for system services and the graphical interface and do not concern us. Two basic classes are used in network simulations: computational modules and communications modules. The computational modules, called 'elements', of two specific classes are used to represent the 'neuron'. The SEGMENT class acts like the neuron body and is analogous to the 'unit' in RCS. The PROJECTION class passes the activation of the segment forward to the SEGMENT-class objects which receive it, using the communication modules of the CONNECTION class. The PROJECTION class differs from the RCS 'site' in that it passes activations forward, where RCS fetches activation from connected units. A second type of communications module provides 'messages' between any classes of objects, and is not subject to time delay as are objects of the CONNECTION class. A 'message', for example, would be used to pass the output of a SEGMENT to its PROJECTION element. The functions, or methods, which drive each object must accept one of a specific group of actions (typically, 'INIT' to initialize, 'RESET' to clear previous activity, and 'PROCESS' to perform their primary function). Each object's methods must also account for any specific type of message that object may receive or send. When an object is instantiated, the specific types of actions and messages the object responds to must be defined.

The user may define new classes of objects, methods and actions as appropriate to their specific needs.

Unlike RCS, user-written code in C is not linked with the simulator code, but placed in simulator libraries, which may be loaded automatically at the simulator start up or on command. A simulation is established by issuing simulator com-

mands (like RCS, GENESIS can read a file of commands) to establish and operate the network.

Unlike RCS, the GENESIS simulation is built up in a tree structure of elements. The elements, consistent with the object-oriented approach, may be elements of the network (such as neurons, synapses, and so forth) or they may be objects which perform the functions of the graphic display, or provide disk i/o and other system services, or they may be 'neutral' elements used merely to organize the tree. This tree structure uses the Unix file system format for addressing elements of this tree. For example the base, or root of the tree is '/', and an element further away in the structure of the tree might be addressed as:

'/branch1/branch4/element3'

This method of reference attempts to address some of the same problems addressed in RCS by the use of character string names, types and sets for the units and sites. GENESIS 'path' references can use wildcards, and GENESIS uses commands which allow the user to move about the simulation structure in a manner similar to that of the Unix operating system in which the simulation runs. This should make the simulator easier to learn for someone familiar with Unix, and appears to be part of an intentional design decision to closely integrate the simulator with the Unix operating system (more on this later).

A simulation must be organized into a hierarchical structure of elements, and it must be constructed in a manner which creates that structure from the top down. To facilitate the process of setting up a simulation, once an element (possibly involving several objects, messages and connections) has been created, other elements or arrays of elements may be created using the element-name and the new

elements will contain the same sub-structure of objects, messages and (internal) connections.

GENESIS uses several different data structures to represent and control the network simulation. Each element of the network is individually allocated, and maintains a parent-child relationship with other elements in the tree - - one parent, possibly many children. Each of these relationships is doubly-linked,(with a pointer from parent to child and another from child to parent) to provide for random traversal of the tree structure. Connection objects are maintained as a linked list attached to each projection object. Messages are individually allocated structures linked to both sender and receiver.

Control of the simulation cycle is vested in an array which contains pointers to the network elements, their functions, and the action to be performed. In this way, simulation cycles escape the necessity of traversing a linked list, at the expense of additional memory usage. A secondary aspect of this method of organization is the concept of a task. There may be multiple schedules created, each run separately, or simulataneously. Potential uses of this feature might include the creation of a 'learning cycle' to alternate with the feed-forward cycle for back-propagation networks, or precise simulation of input stimulus in a biological model.

GENESIS provides for 'clocks' to be associated with each element of a network for precise control of propagation delays and time-based simulations.

The simulator may be controlled through the graphical interface or from a command line, but a preferred method appears to be through the use of command scripts making use of its own script language interpreter (SLI) which provides a set of operators, basic program constructs such as 'if' clauses, and function definition

constructs.

2.3 SFINX

The SFINX simulator (Structure and Function In Neural conneXtions [sic]), currently in version 2.0, was developed at the University of California at Los Angles by Edmond Mesrobian, Michael Stiber Josef Skrzypek and Kyle D. Henriksen. It was, in turn, based on an earlier simulator named PUNNS (Perception Using Neural Network Simulation) written by David Gunger. This earlier simulator was apparently used with gray scale images as input and specified its network structure by means of a 'connectivity language'. This 'connectivity language' allowed the user to specify a node's name, activation functions and connections to other nodes. The authors acknowledge that the experience of the earlier simulator demonstrated that the explicit specification of each individual node, while offering a high degree of flexibility, was computationally expensive at run-time for parsing and allocating memory for each node, and generally resulted in the creation of large network specification files.[8]

The authors of SFINX approached this problem in two ways. First, they drew a distinction in network types, choosing to call those networks which require node-by-node specification 'explicit', while networks consisting of "two-dimensional layers of functionally identical nodes whose connectivity patterns are highly regular and

[8]Edmond Mesrobian, Michael Stiber, and Josef Skrzypek, UCLA SFINX: Structure and Function in Neural Connections, (Los Angeles, California: UCLA Machine Perception Laboratory, 1989), 12-16. This manual is part of the simulator distribution.

fixed" [9] are called 'implicit' networks.

To create an explicit network, a simple description of each node and its activation functions and patterns of connectivity is written in a file (similar to that used for the predecessor, PUNNS). This file's format is somewhat Spartan, and the authors point out that it may easily be prepared by a user program. The file is then passed through an assembler-type program to produce a binary representation which can be loaded directly into the simulator at run-time. This reduces the run-time overhead of parsing and interpreting a descriptive language, allows for a single large allocation of memory, and somewhat mitigates the problem of the size of the specification files. Each node in an explicit network may have any number of 'registers' (of the type designated at the time the simulator is created -essentially any valid C language data type: char, short, int, float, long or double) and the user writes two functions for each node-type used in the network (one to initialize the node, one to perform its processing functions.) These user-supplied functions are compiled into the run-time environment of the simulator.

To create an implicit network, no specific 'nodes' exist. They are implicitly represented by a three-dimensional array structure (which is called a 'buffer') where elements share the same topological location, and their mandatory registers (activation function pointers, for example) and optional registers occur as the third dimension of the array. An implicit network activation function must be capable of determining the necessary pattern of connections as positions relative to that of the node being processed. As with explicit networks, the implicit network functions are compiled and linked into the run-time simulation environment. No network description file is used. Simulator commands allocate the buffer structures directly,

[9] Ibid., 30.

since the burden of the node's connectivity falls upon the activation function.

The implicit network approach is unique among the simulators surveyed for this study, although other simulators do have special commands to take advantage of groups or layers of nodes with identical function and connectivity. Regrettably, this aspect of the SFINX simulator was less suited for comparison with the other simulators than the explicit network type, and the test program could not be reasonably implemented using it. We will speak no more about this particular form of representation for now.

The identification of a particular node in SFINX (in either type of network) is accomplished through the use of a node identifier. This is a set of three integers, or ranges having the form:

[z_range, y_range, x_range]

where a single value indicates a specific co-ordinate, and a range is expressed as 'lower_bound:upper_bound'. A range can also take the form 'lower_bound:' for all values greater than the lower bound and ':upper_bound' for all values less than the upper bound. If a range is left empty it signifies all possible values in that range.

To simulate an explicit network, the network description is prepared and assembled into a binary representation in advance. This involves the use of separate arrays for nodes, connections and their registers. When the simulator is started, it can be given precise parameters regarding the size of the memory it must allocate (only required if this exceeds the default parameters) and the binary representation is loaded into the simulator. During simulation, the nodes are processed in

the order in which they are defined, and the output for all nodes is updated only at the end of a complete cycle through the network.

User functions may be added to the simulator, and may access and modify the values within the network. A series of C language macro definitions are supplied for common functions, and individual nodes may be addressed as offsets from a network base address –indicating that the network is arranged as an array of nodes similar to the 'unit vector' in RCS.

There is no specific provision for the support of propagation time controls within the simulator itself. However, user functions could undertake to control the specific timing of a simulation.

2.4 SIMULATOR COMMAND COMPARISON

Table 1, below, summarizes the commands needed to establish and operate a simulation in each of the three simulators.

Table 1. Simulator Command Comparison.

Command Purpose	RCS	GENESIS	SFINX
Add/Change/Delete elements and connections	Primary means of building up network is a conventional call to a 'build' function written in C. Interactive commands include: AllocateUnits <number> MakeUnit, AddSite, DeleteSites, MakeLink, DeleteLinks	create <element-type> <pathname> createmap (multiple copies) delete <element-path> sendmsg/deletemsg - send or delete messages between elements.	load filename (explicit networks must be 'assembled' into binary representation prior to use. No changes to the pre-defined network are allowed within simulations.
Set Node values	Out <UnitID><value> set output value also Pot, State, Weight, UFunc, SFunc, LFunc, Reset, delay	set <path> <field> <value> - set the field at path to value	en - for 'Explicit Network' will set or display one or more fields in specified unit(s) depending upon parameters.
Get or Display Network Values	List < link | set | unit UnitID > show values of the described item	show <path> <field> - list the value of a given element's field(s)	
Execute Simulation Cycles	go <stepcount>	Necessary to create a simulation schedule with AddSchedule, ClearSchedule and then use the command step <stepcount> to execute it.	runen <stepcount> for explicit networks and runfa <stepcount> for implicit networks.
Save/Restore the Simulation State	checkpoint/restore will save values but will not restore the network. save/load will retore values and it will rebuild the network itself.	No provision for saving or restoring is provided.	save/load restores the entire network, with the exception of function pointer values.
record and replay commands	log/read	logfile/include	no provision for recording exec <filename> reads commands
global user variables	Available in user-coded functions	int five = 5 float ten = 10.0 str hello = "Hi There!"	set - envirionment variables setu - user's variables usetu - delete user's variables
Name Conventions	Units may be named individually, or as VECTORS or ARRAYS. They may be affiliated with sets. In commands, units may be referenced by index or by name. A range is separated by a ' - ' string and may consist of any combination of name and/or index. In some cases set names may be used. Units must be named using the NameUnit <UnitID> command.	Elements are referenced using their full 'pathname'. Wildcards may be used for portions of the pathname so that additional organization of the network may be established.	Nodes are 'named' with a three-digit subscript of the form [z, y. x]. A range may be referenced using a colon: low:high -- from low to high low: -- Greater than low :high -- Less than high <nothing>-- All possible values Ranges may take any part of the 'name' (see text.)
User Written Functions	call <funcname> <args> - Call user function using (argc, argv) for parameter passing.	addfunc <funcname> <sliname> - bind a user function to a script language command. Uses (argc, argv) also.	<funcname> invokes user functions User functions are included in simulator control lists, and use a command structure to parse the input and pass parameters.

CHAPTER 3

Performance Testing

In this chapter we will describe the implementation of a test network on each of the simulators introduced in chapter two. These test simulations were intended to provide experience in dealing with each simulator, and to provide results which might be used to gain insight into the relative merits of their respective implementations.

3.1 A Hamming Decoder Network

In order to test the basic size/response characteristics of the simulators, a network simulation which could be scaled to arbitrary size was required. Insofar as training time for such arbitrary networks could become unreasonable, it was also considered desirable to make use of a network which could be programmed, that is to say, could have weights assigned a priori. For this purpose, a network which would decode the well known Hamming single-error correcting code was chosen.

R. W. Hamming developed the single-error detecting and correcting code which bears his name.[1] In essense, a string of bits is encoded by adding parity bits at positions representing powers of two, and those bits are given a value which will

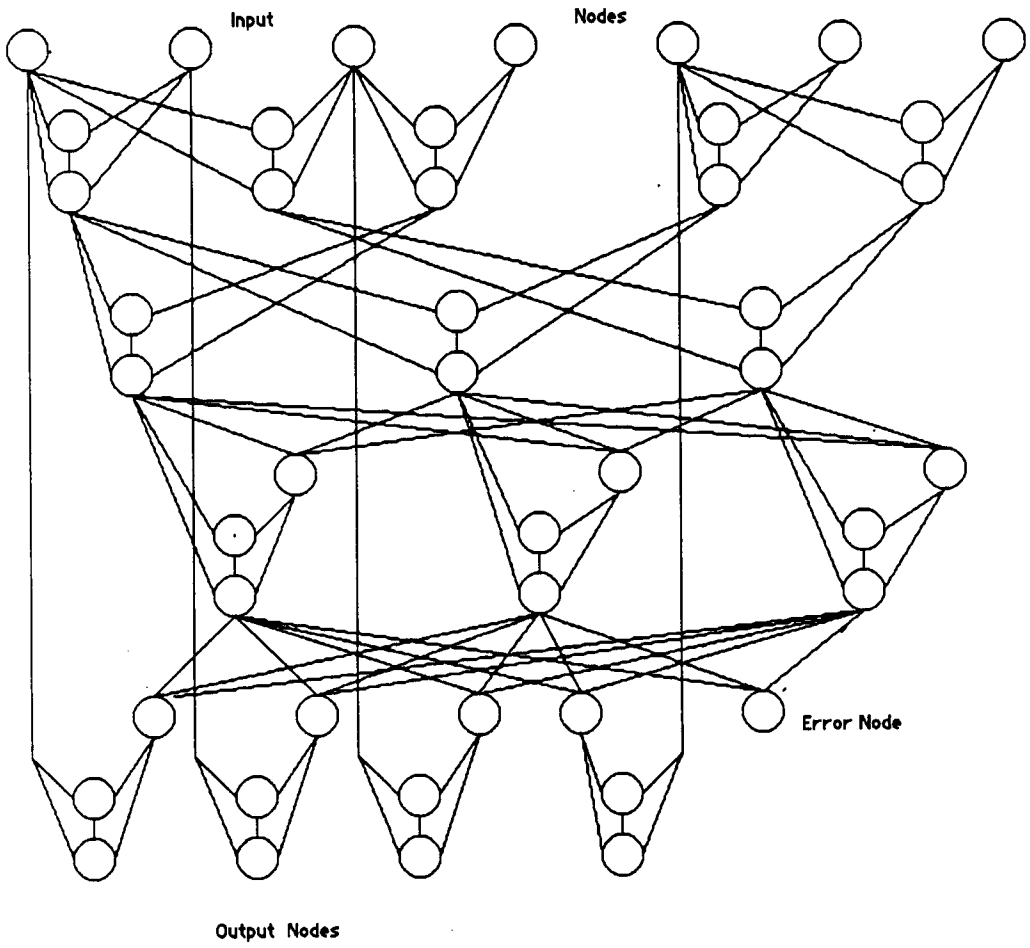
[1] R.W. Hamming, "Error Detecting and Correcting Codes", Bell System Tech. Jour. 29, (April 1950): 147-160.

cause the exclusive-or of all the bits whose position-numbers contain the same power of two to be zero. In decoding the string, the process is reversed, and an error in any single bit will cause a one to appear in every parity bit representing a power of two used to describe that bit's location in the string. In other words, the binary value represented by the parity bits is the location of the error, which can then be corrected. This technique is widely known and used in data communications and computer networking.

It should be understood that the implementation of this Hamming network is little more than a form of logic design. It is used here merely as a test vehicle.

To establish the methodology for this test, a program was written in C which could create networks to accept Hamming encoded input of arbitrary length. The program was 'bare code', producing only minimal output –a linked list of units and connections. This list could then be modified for use by the simulator being tested, or used to produce a file describing the network, if the simulator being tested used this a means of input. Ancillary functions were provided to create random input strings, encode them and test the output of the network for correctness. These programs were verified exhaustively for small networks and selectively for large networks.

This 'network generator' program functions by establishing an input array of units, then constructing a chain of units which performs the exclusive-or operations required by any given parity bit. Each of these exclusive-or units consists of two neurons in two layers. A simple list of existing connections is maintained, which allows several of these parity chains to share the results of the exclusive-or operations to prevent redundant units. An example of a decoder network for four data bits is shown below.



Four-bit decoder network

When the parity chains are activated, the result is the binary representation of the number of the bit in error in a particular input. If this number contains only a single bit (i.e., is a power of two) then it is one of the parity bits which is in error, and this error is corrected first. Otherwise, each data bit is exclusive-ored with the parity bits to correct the error. An uncorrectable error is recognized when the

parity bits indicate a bit number that is greater than the number of data bits in use. The output of the network is the binary representation of the decoded input plus the error bit. When the error bit is set, the binary representation should be ignored.

The activation function employed in this network performs hard thresholding against a sum of weighted inputs. Each unit maintains a threshold value and a list of incoming connections, consisting of a pointer to the unit from which the input is drawn and the weight by which that input is multiplied. If the sum of the weighted inputs is greater than the unit's threshold, the unit's output is one. If not, the output is zero. Floating point representation is used for weight, threshold and activation values. This choice was made to reflect the needs of learning algorithms (back propagation, for example) which must use floating point values. Each unit also maintains a unit identification number, layer number, an index on the layer, a unit type classification, list pointers and certain other information, depending upon the specific simulator's needs.

It was also found useful (in some cases necessary) to order the network description by layer and index within each layer. The specific modifications provided will be discussed along with each simulator, but the creation of the network description list was essentially the same in all cases.

In preparing the simulations, it was necessary to convert the linked list into the form required to meet the simulator's specific requirements, and adapt the functions which supplied the network with input and checked its output. In addition, a simple timing function based on the Unix system clock was used, in appropriate form, with each simulator. This function read the system clock at each start and stop, computed the elapsed time, and accumulated multiple trials. It then

reported to a file the mean, standard deviation and standard error of the event timed. Other performance figures were gathered by use of the Unix system 'time' command.

Items specifically examined include the creation of the simulator network (the list created by the program described above being a reasonably constant figure), the actual time needed to set up the network for use, and the network operation time (defined as the time required to process information from input to output –not including the process of loading the input nodes or unloading the output nodes). Figures derived from this data include the per-simulation-cycle averages and approximate overhead time (the timing functions and data handling functions can again be regarded as reasonably constant.)

For each of fourteen network sizes (input bits ranging from 4 to 2048, networks ranging from 45 to 16433 nodes) a simulation was run which included the program startup, network instantiation, and at least forty input/output cycles. The comparisons to be made with these figures involve time increase with regard to size. Raw time itself is not an important number, insofar as it is primarily dependent upon the speed of the machine being used. The relationship of time change to network size change, however, will be a useful measure of the efficiency of the simulator. This should be reasonably comparable across the various simulators. Tests were conducted on a Sun 3/60 workstation in a networked environment, with no other users on the local machine. The workstation was equipped with a 68881 floating-point processor, and eight megabytes of memory.

3.2 Adapting the Hamming network generator for RCS

The RCS simulator, as has been described, creates a network by a set of calls to system functions from a user-supplied program conventionally named 'build'. In adapting the Hamming decoder network program to function under the RCS simulator, two options present themselves. In the first option a file description consisting of simulator commands would be written for the network, and the RCS simulator would read this file and make the appropriate calls to create the network. However, the more direct method, advocated in the manual, would be to take the linked-list provided by the basic program and convert it directly into the simulator calls necessary to create the network.

In order to utilize this approach the nodes are ordered into layers to reflect the sequential organization of the unit vector in RCS -units are simulated in the order in which they are created. Each cycle through the network produces new output only after all nodes have been processed, so the results in the output and error nodes require one less cycle than the number of layers. The simulator uses a modified fixed-point scheme to avoid floating point operations, and network weights had to be adjusted accordingly.

It was necessary to provide unit and site functions for the simulation. These are shown as pseudocode below:

```
UFthreshold( unit_pointer )
                {
                        sum = 0;
                        for every site on this unit
```

```
                        add site's value to sum
                if( sum > threshold )
                        unit's output = 1
                else
                        unit's output = 0
        }

SFweightsum( unit_pointer, site_pointer )
        {
                site value = 0
                for every connection on this site
                        multiply the connection's
                        input by the connection's
                        weight and add this to the
                        site value
        }
```

The names follow the implied convention of RCS: 'UF' for unit function, 'SF' for site function. The unit function is a simple hard threshold against the site value. The site function performs a weighted summation. The weight and threshold values are integer (scaled upward if needed, but not reduced later.) The weight is limited to the value of a 16-bit integer, so the upper limit of the example network is 16 parity nodes.

In addition to these functions, a user command was written which operated the test simulation by calling for specific input, loading and unloading the network

and operating the timer functions. In this manner, the entire test sequence could be run from a series of command scripts, without operator intervention.

3.3 The GENESIS test implementation.

In order to implement the Hamming decoder test program as a GENESIS simulation a number of things had to be done. As the simulator operates on scripts, the network description had to be written to a file, then that file read back into the simulator. This could conceivability have been done be done from within the simulator itself, but there is no particular advantage in doing so. The network generation program was modified to write the description to a file.

It was decided that the tree structure of the network would have branches for each layer, with the neurons on each layer appearing as leaves. The 'branches' would be named 'layer0' through 'layerN' (where N is the number of the output layer). This would allow the description file to use commands which facilitate the creation of larger simulations –specifically 'createmap'(see the command summary in chapter two, above). A single neuron structure (the SEGMENT and PROJECTION objects and the message between them) were created, then copied into the appropriate places. The createmap command replicates the name of the source item by using a subscript for multiple copies (i.e., neuron, neuron[1], neuron[2], ..., neuron[N]). This scheme allows the layer and layer index numbers to map onto the naming scheme of the simulator. 'Layer 6, index 4' becomes '/layer6/neuron[4]' –necessary since the script language interpreter requires use of the full pathname.

After the units have been created, the connections and weights are established. The established convention of feeding activations forward from projection objects

required that the list provided by the network generator be resorted, but this was trivial. The process of establishing the connections and weights did not prove to be so trivial. Separate commands are required to establish a connection and to set its weight. The method used to refer to the connection shows a weakness in the GENESIS scheme.

Suppose an element at '/layer5/neuron[4]' has five connections, and that they have all been established. We would have to refer to the first connection established as:

'/layer5/neuron[4]:4'

and to the last established as:

'/layer5/neuron[4]:0'

Obviously, the connections are being pushed down on a stack. This means of reference runs counter to the intuitive nature of the system.

Finally, a simulation schedule needed to be established. In order to accurately time the simulation process, a 'stopwatch' object was created, and given the actions START and STOP during the actual simulation cycle. Between these actions, the other active elements were invoked with the action PROCESS. Only input units were not active, but each element in the network is composed of a SEGMENT-class object and a PROJECTION-class object, which must be activated in that order.

For the larger simulations, the description files became enormous and this caused problems, as we shall see in the next chapter.

The SEGMENT-, PROJECTION- and CONNECTION-class objects for this network required that methods be written for those objects. The methods for the SEGMENT and PROJECTION classes are represented below.

```
HardThreshold( Object_Pointer, Action )
{
        In case the Action is:
        INIT or RESET do
                set the object's activation and
                output to zero.
        PROCESS do
                If the activation > threshold
                        set object's output to 1
                else
                        set object's output to 0
}

ProjectOutput( Object_Pointer, Action )
{
        In case the Action is:
        INIT or RESET do
                set the object's state to zero.
        PROCESS do
                Obtain new state from a message.
                If new state is
                        for every connection
```

```
                                    Add the connection weight
                                    to the connection target's
                                    activation
                        else
                                Do nothing.
        }
```

The basic form for an object method is that of a 'switch' statement in the C programming language. An object method always receives a pointer to the data structure associated with that object and a pointer to a structure containing the type of action required (and additional arguments, if any). These two methods perform the same task as the RCS examples shown above.

As the GENESIS simulator operates using a script language interpreter, the functions to provide input and check the output of the network and to operate the simulation itself were prepared in this script language. The style of this language is similar to C.

For testing purposes, the network generator was invoked to produce the network description file, and this script file invoked two other script files, which instantiated the network objects and established the functions shown above. After the network description file was produced, the simulator was invoked to use it. This process was then described in a Unix shell script, so that the entire testing sequence required no operator intervention.

3.4 Implementation of the SFINX test network.

Implementation of the SFINX test simulation was more straightforward. A simple modification of the network generation program wrote a description of the network to a text file, then wrote a Unix script file to compile the network (invoking the Unix c-shell timing facility), and finally prepared a simulator script to be used to run the simulation itself. As before, the entire test run was prepared in a Unix shell script requiring no operator intervention.

Since SFINX pre-compiles the network description into a binary form, it avoids the run-time overhead of interpreting the network description and repeatedly allocating individual nodes. However, the binary representation prepared by the assembler cannot know the address of the node activation functions, since these are determined when the simulator itself is loaded prior to execution. This information must be provided during the simulator's initialization of the network, and for a network using many different activation functions (admittedly, perhaps, a rare bird) this has the potential of becoming a pretty mess.

Fortunately, the manner in which SFINX identifies its nodes is both flexible and reasonably mnemonic. Recalling that each node is addressed as a set of three ranges, the scheme chosen for the test simulation naturally involved the use of a node's layer and index on that layer for an identifier. With the range description capability it was a simple matter to assign all nodes on the zero, or input, layer no function (their values would be set by the data input command) and to assign the hard thresholding activation function to all nodes above layer zero.

Preparation of the node's initialization and activation functions was very simple. In this case no initialization function was required. The activation function

makes use of the predefined simulator register type (see above) to provide for portability of activation functions to simulators with other basic register types. Activation functions in explicit networks receive standard parameters consisting of a pointer to the current node, the previous output value and an array of current input values. (Some minor confusion was created here, as the documents refer to this parameter as a pointer to a list.) The activation function returns the node's output value.

```
enfn_threshold( node_pointer, old_activation, inputs )
{
        new_activation = 0
        for all links coming into this node
                Add the input multiplied by the
                weight of the connection to the
                new_activation.
        if the new_activation > threshold
                Return a value of 1.
        else
                Return a value of 0.
 }
```

SFINX conventionally names explicit network functions 'enfn_something' (to contrast with 'infn_something' for implicit networks; see above). This function performs the same tasks as the two functions in each of the previous examples.

The process of incorporating new functions into the simulator as commands

involves modification of the Makefiles and the addition of the command descriptions to the simulator files which must keep track of them, and recompiling the simulator. Simulator commands are parsed by the simulator into a specified command input type, and the command program itself must test the input type for validity, and take appropriate action. New command input types may be specified, using the Unix 'yacc' utility. Addition of simulator commands to provide input and evaluate the output of the network, and to perform the timing functions was hampered only by the simulator's desire to deal only with the its specified register type. This was overcome by compiling those functions which used types other than the simulator's separately, and linking them in explicitly. (This works well when only a few files are involved. A more general method would be to make all such functions into a support library, and then add that library to the appropriate parts of the Makefile.)

In this section we have discussed the Hamming decoder network used to test each simulator and have used the implementation of this test to provide some general examples of how the simulators are programmed. In the next chapter we will present the results of these tests.

CHAPTER 4

Test Results and Conclusions

In this chapter, the results of the Hamming decoder network test series will be presented and discussed, concluding with a critical assessment of the simulator performance.

To provide an additional comparison, a custom simulation was established by providing an activation function and operating directly on the linked list provided by the network generation program. This simulation was given random data, its output was verified, and it was timed in a manner similar to that of the other simulations.

Figure 1 shows the total time required to process the test sequence from start to finish, including the building of the network list, set up of the simulator, establishing the network and running the simulation through forty iterations.[1]

It would be a serious error to pass judgment on these, or any other programs, solely on the basis of execution speed, or any other single criterion. While effi-

[1]The data from the test runs on which this and the other figures in this chapter are based are presented in tabular form in Appendix Two. They are shown here as connected points. The apparent non-linearity of these graphs is a product of the Hamming model itself. When a new data bit is added to the input, it increases the total number of nodes by a fixed amount (three nodes for error correction and output) and a variable amount based on the number of parity groups to which it belongs. When the addition of a new bit triggers the introduction of a new parity bit, an additional fixed increase occurs, and a new parity chain is created. Each parity chain contains roughly half of the total bits. With the addition of each new parity chain, the average number of nodes added to the network as a result of memberships in a parity group increases. This, in turn, increases the slope of the line after each new parity bit is added.

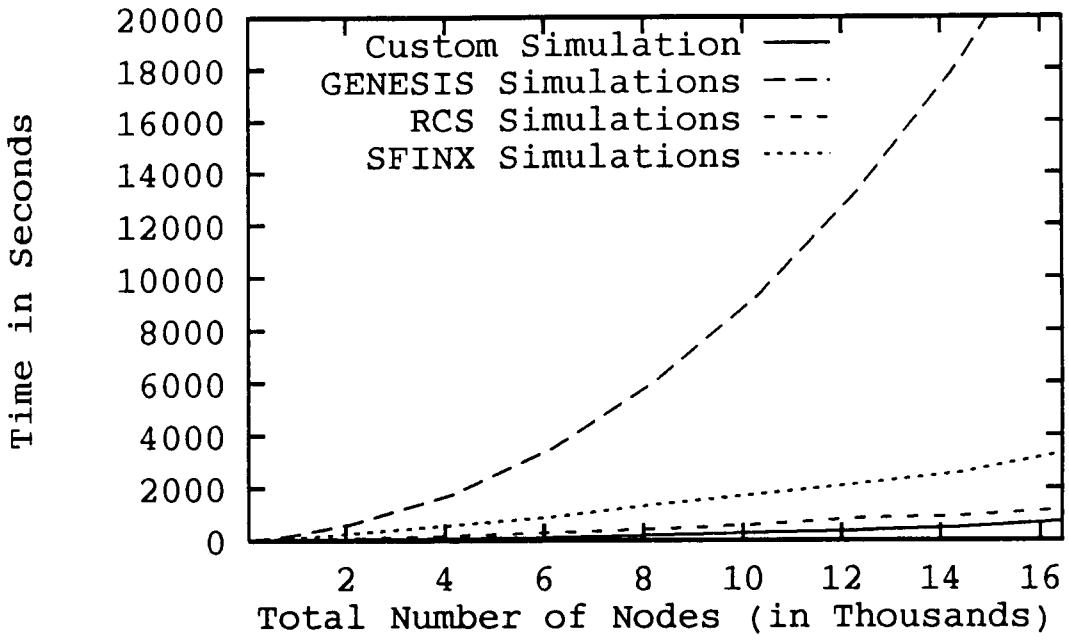


Figure 1: Total Job Time

cient performance was on every designer's list of objectives, in no case was it the paramount concern. Further, it would be only fair to point out that the use of an instrument such as the Hamming decoder network represents an artificial test.

That said, we will try to interpret these results for what they are intended to be –very general guidelines. In Figure 1 it should come as no surprise that the custom simulator is fastest as it requires no additional setup or loading of simulation code. The RCS simulation builds its simulation directly from the output of the network generator, while both SFINX and GENESIS create an intermediate description file.

The time required to build the network, including operation of the network generator is shown in Figure 2. The increase in the time required to create the SFINX and GENESIS descriptions may be attributed to the necessity of writing

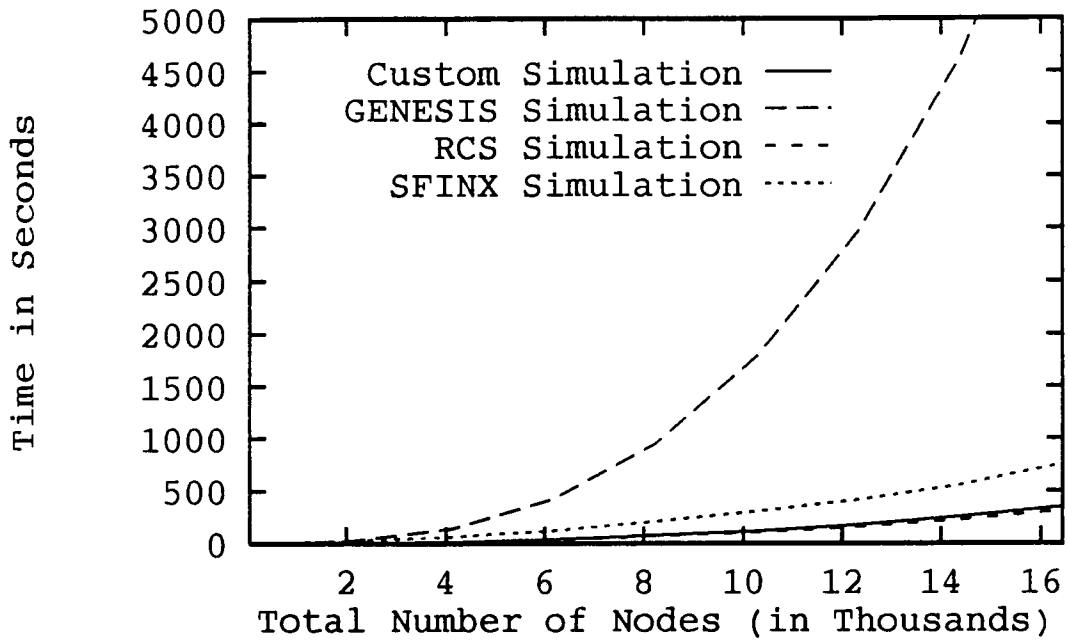


Figure 2: Total network building time

a file. The disproportionate amount of time spent on the GENESIS file is due in part to the complexity of the description (the effects of which will be seen again later), and in part to the need to create a simulation schedule.

In Figure 3, the time per single simulation cycle is presented. This time represents a single simulation of the network, whereas the time required to simulate the network from input to output may involve several cycles. Both RCS and SFINX delay the update of unit outputs until the end of a complete simulation, requiring one less cycle than the total number of layers to completely pass the inputs through the network. This effect is shown in Figure 4, below. The relative inefficiency of the custom simulation is undoubtedly the product of its overall structure as a linked list, using other linked lists for connections. RCS and SFINX use arrays as the basis of their network data structure. The RCS simulator also incorporates 'scaled' floating point representation which reduces floating point numbers to integers. The

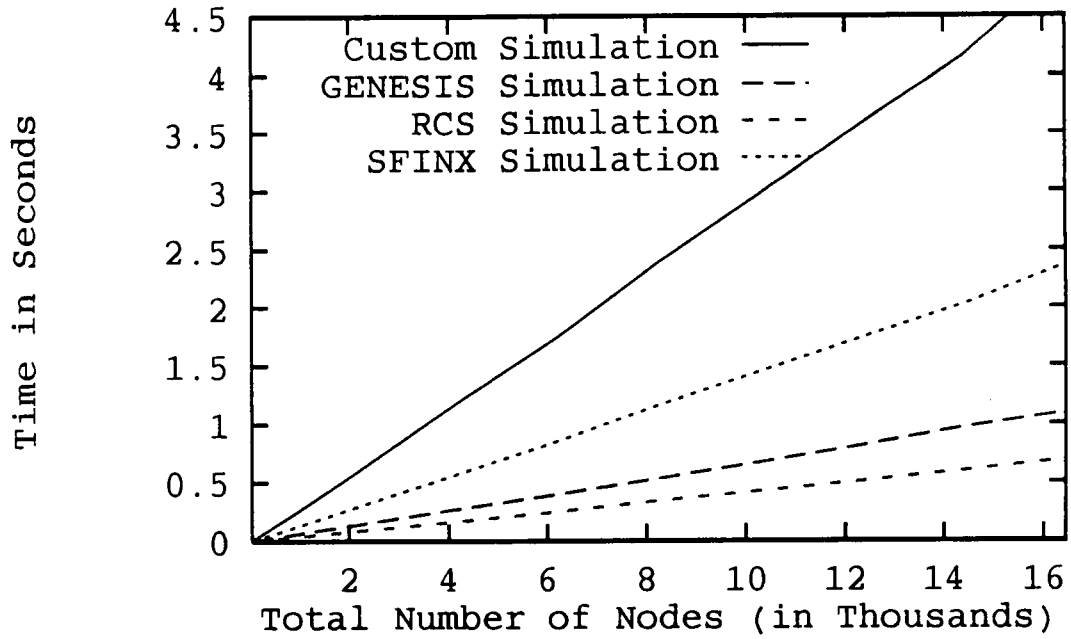


Figure 3: Single simulation cycle time

efficiency of the GENESIS approach is a combination of the network data structure and use of an efficient simulation schedule structure. The schedule contains pointers to the functions and elements in an array, and therefore circumvents the time required to move through a linked list. Both the RCS and GENESIS simulators use a simple set of parameters, where SFINX provides an array of current input values, which increases the parameter-passing overhead per call.

4.1 Components of Individual Performance

In the section which follows, the component times of the individual simulators will be considered. The components examined will be network building time, set up time (the period from the completion of the network description to the beginning of the simulation, based on actual timings), simulation time (this time is estimated

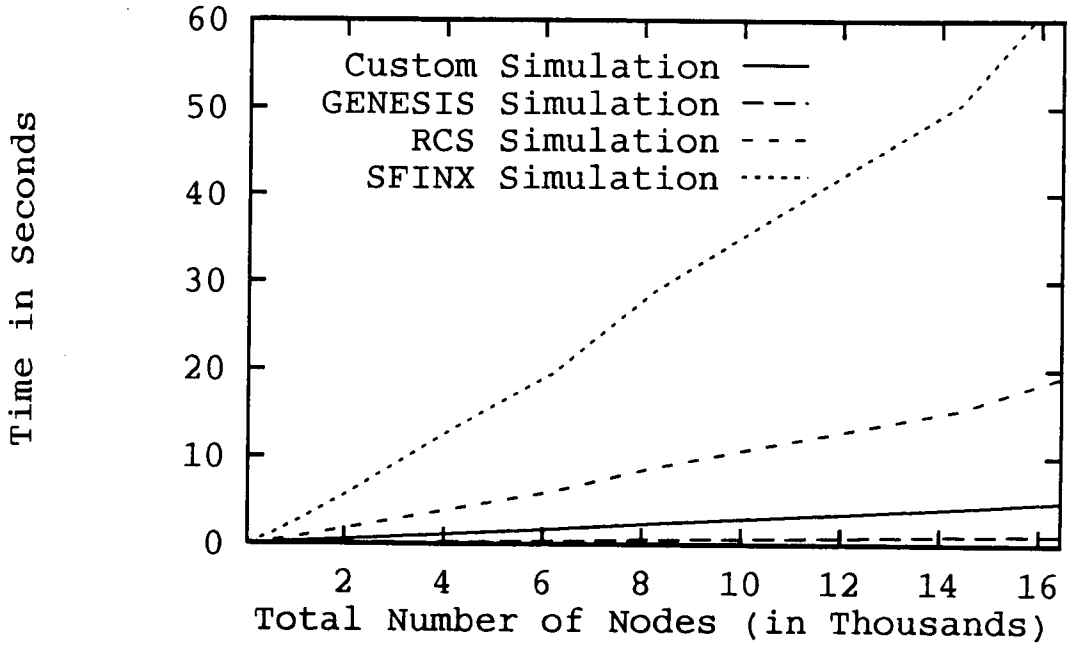


Figure 4: Total Network Simulation Time

by multiplying the average network simulation time by the number of simulation cycles performed), and total job time as presented in Figure 1. The portion of the graph between the simulation time and the total time provides a reasonable estimate of the amount of 'overhead', or time consumed by other simulation tasks.

The RCS simulator appears to spend the bulk of its time processing the simulation. Despite some obviously problematic estimates, it would appear to spend very little time on network set up and overhead activities. This would seem to be attributable to the ability of user programs to create and directly access the simulation.

The network building components show in Figure 6 include the assembly of the description file into a loadable binary image. As a result of this design approach actual network set up time is practically insignificant (in real terms, four seconds to load a 1.3 megabyte file and perform 16K assignment operations for function

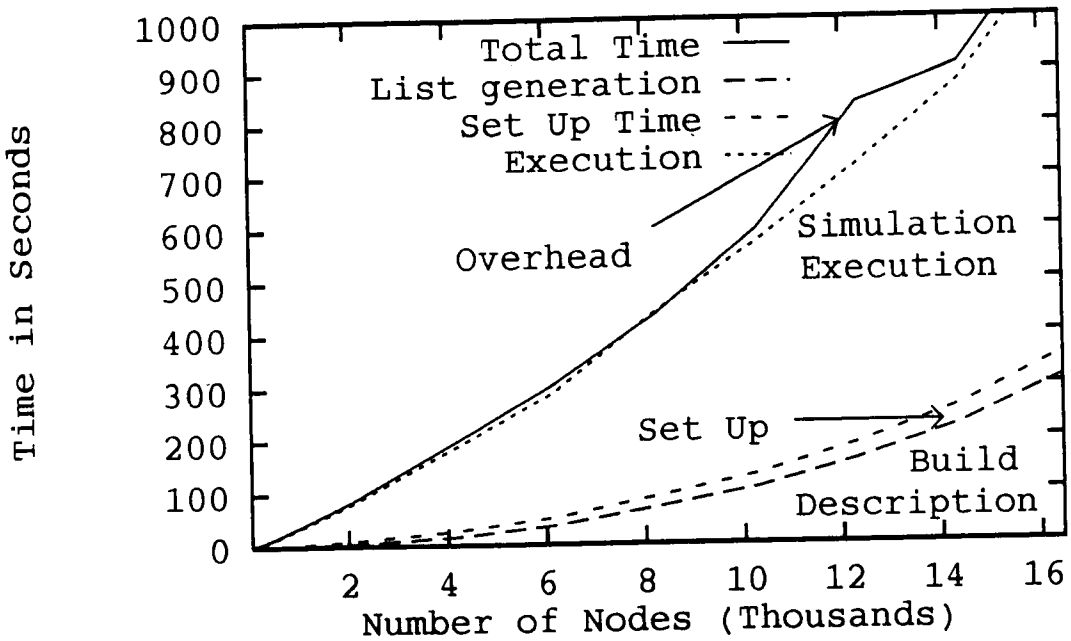


Figure 5: RCS Components

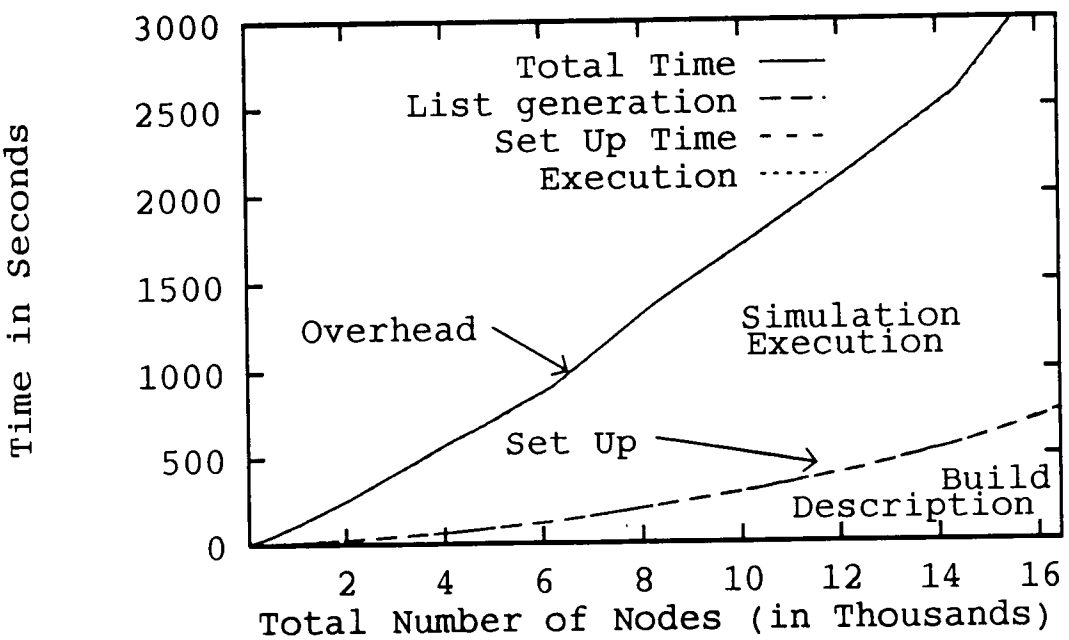


Figure 6: SFINX Components

pointers).

Of the simulators examined, SFINX is the only one which contains no specific facility for implementation of propagation delay simulations. It is, of course, conceivable that such delays might be implemented through the use of user variables and the specific user-provided node functions. It is also necessary, due to the 'gating' of node output, to execute several simulator cycles to obtain output. However, as is the case with RCS, this limitation can be overcome by user-provided activation functions althought the programmer must be prepared to go digging into the code for methods to provide this type of function.

Even though this study does not focus on the graphics interfaces of the simulators examined, it is worth mentioning that this simulator requires color graphics (minimum four-bit color representation) to function. As it is written for use with the X Window graphical user interface, it seems possible that this limitation could have been avoided, but one can only envy UCLA where such equipment must be taken for granted! Happily, a new version of the simulator (expected in September of 1990) will include support for monochrome displays.

The time required to build the description file has already been commented on, as has the remarkable efficiency of the simulation action itself, which makes the simulation time in Figure 7 lost in the line above setup time at this scale. GENESIS does not require multiple simulator cycles to pass input through a multi-layered network. The proportion of total time devoted to setting up the network is due to the overall complexity of the data structures used to contain the network. When an element is created it must be individually allocated, named from its path parameter, linked to both parent and child. Connections must be individually allocated, linked and weighted. Messages must be linked to sender and reciever.

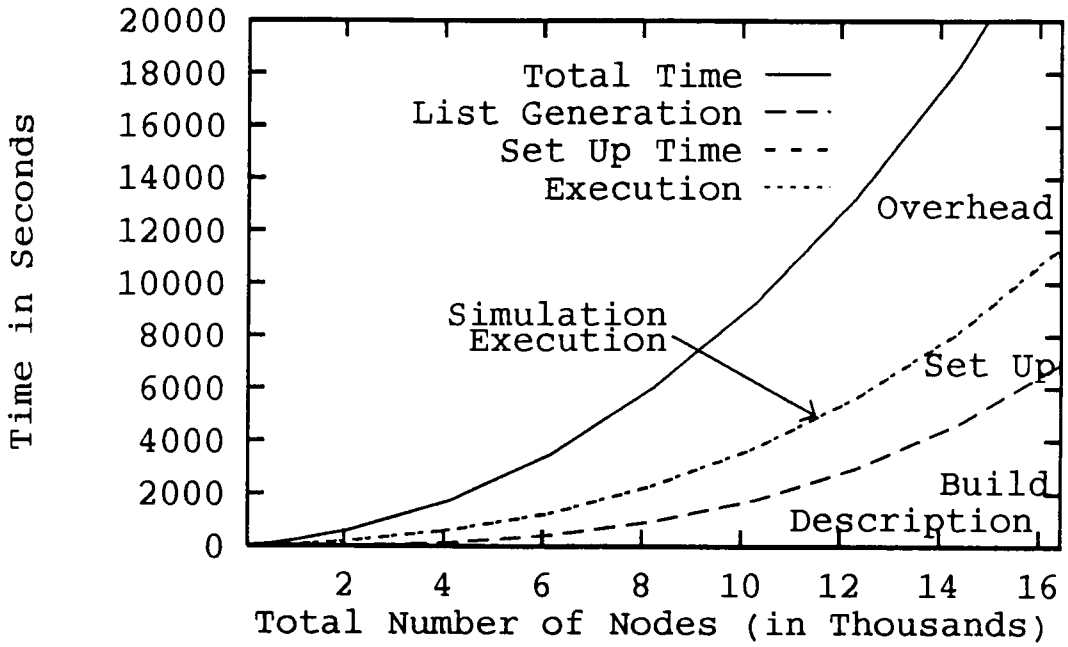


Figure 7: GENESIS Components

Finally, the simulation schedule must be independently created -a feature not appearing in the other simulators. The enormous amount of 'overhead' time seen in the graph would appear to be taken up by the script language interpreter, the overall complexity of the network structures and the very wordy nature of the network reference system.

The reference system mimics the file system structure in the Unix operating system. In fact, it would appear that a conscious effort has been made to make the operation of the simulator as similar to Unix as possible: the commands used to traverse the simulation's structure resemble those used for moving about the file system (see Table 1 in chapter two), and Unix commands typed at the simulator prompt 'drop through' to be executed by the operating system. This is both a blessing and a bane. On the one hand, the concepts presented are familiar and

reasonably comfortable to a new user familiar with Unix. On the other hand, Unix is not widely applauded for its user friendliness. And where similar commands differ by only one letter, hands may type a wrong command from force of habit.

The system of addressing elements in the GENESIS simulation is very intuitive. It provides the same abilities provided by the various name and set functions in RCS, and is more mnemonic than the subscript-style names used by SFINX. However, each time a command is invoked a full pathname must be typed, parsed and interpreted (or the user must make the object of the command their 'current element', a process requiring two commands in place of one and only advantageous in certain conditions).

Finally, the interpretation of the script language takes up an amount of time which is not expended by the other simulators. In RCS and SFINX, simulator commands are acted on directly and new commands must be programmed in C. The parsing of these commands and the passing ofarguments, is done in a much narrower context. The GENESIS script language facility provides capabilities for the creation of functions, for example, and the evaluation of expressions. On the other hand, the C language –in an unusual (though not particularly difficult) style– must still be used to create new classes of objects and methods. The user still must program the simulation, and must learn the script language, which is quite extensive, it would appear.

4.2 Ease of Use

This report has examined three simulators with regard to their network structure and reference systems, and offered some observations with regard to their

relative performance with network simulations of various sizes. We will conclude with some brief remarks regarding their ease of use.

RCS is clearly the most mature of the simulators. Its documentation is extensive and clearly written. It is versatile, and flexible enough to encompass most simulations in the connectionist domain.

The SFINX documents are well written, contain a number of excellent examples and are organized as a useful series of concise sets of instructions. The only difficulty encountered was in obtaining the diagrams, which were provided as separate files. SFINX is, perhaps, a bit more limited in scope than RCS, but it is a newer product. The SFINX approach to network representation is innovative and efficient and the 'implicit' network representation, which was not examined in this paper, appears to offer distinct advantages for large networks with regular patterns of connections.

The GENESIS simulator is still in a process of creation. The object oriented approach seems appropriate and is not hard to grasp. The simulator is also probably capable of providing much more detail and control of a simulation at many levels, as the authors claim. But the documents are incomplete, disorganized and in some cases just wrong.[2]

To require a fee to obtain them, and then to have to decipher the mazes of documents requires the patience of Job. Yet the simulator is clearly an excellent piece of work, and deserves the effort required to bring it to maturity.

[2]Apparently Caltech does not possess a spelling checker.

BIBLIOGRAPHY

Bibliography

Brumbaugh, David. "Object-Oriented Programming in C." The C Users Journal 8 (July 1990) 113-122.

Goddard, Nigel H., Kenton J. Lynne, Toby Mintz and Luidvikas Bukys. Rochester Connectionist Simulator. Rochester, New York : The University of Rochester Computer Science Department, 1989.

Hamming, R. W., "Error Detecting and Correcting Codes." Bell System Technical Journal 29 (April 1950): 147-160.

Koch, C., and I. Segev, eds. Methods in Neuronal Modeling: From Synapses to Networks. Cambridge, Mass. : MIT Press, 1989.

Kohavi, Zvi., Switching and Finite Automata Theory. New York : McGraw-Hill Book Company, 1978.

Mesrobian, Edmond, Michael Stiber and Josef Skrzypek. UCLA SFINX: Structure and Function in Neural Connections. Los Angeles : UCLA Machine Perception Laboratory, 1989.

Uhley, John D., Upinder S. Bhalla, Matt A. Wilson, David H. Bilitch, Mark E. Nelson, and James M. Bower. An Interpretive Tool for Creating and Manipulating Widgets And Their Attributes under XODUS: An X-Based Output and Display Utility for Simulations. Pasadena, California : California Institute of Technology, 1989.

Wilson, Matt A. Documentation for GENESIS and XODUS. Pasadena, California : by the California Institute of Technology, 1989.

This document is available to registered users only.

Wilson, Matt A., John D. Uhley, Upinder S. Bhalla, David H. Bilitch, Mark E. Nelson, and James M. Bower. GENESIS & XODUS: General Purpose Neural Network Simulation Tool. Pasadena, California : California Institute of Technology, 1988.

Wilson, Matt A., Upinder S. Bhalla, John D. Uhley, James M. Bower. GENESIS: A System for Simulating Neural Networks. Pasadena, California : California Institute of Technology, 1989.

Zipser, David, and Daniel E. Rabin. "P3: A Parallel Network Simulating System." In Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Vol. 1, Foundations, ed. David E. Rumelhart and James L. McClelland. Cambridge, Mass.: MIT Press, 1986.

APPENDICES

APPENDIX A

Listing

A list of publicly available simulators for artificial neural networks.

The following list is by no means to be considered complete or exhaustive, it merely represents those simulators which have come to my attention through postings on the internet and articles in various journals. Inclusion in this list represents no particular endorsement, and ommission from the list should be regarded as unfortunate oversight on the author's part. The list is provided as a starting point for individuals and institutions who desire to obtain software suitable to their needs who wish to examine these offerings.

The term "publically available" may be taken to mean anything available at nominal or no cost through channels other than those conventionally regarded as commercial software products. Since the term 'public domain' has become synonimous with free and unrestricted software it should be noted in passing that much of what is available is, in fact, copyrighted work which must be licensed -albeit at no cost.

The term 'ftp' used below stands for file transfer protocol, which is a means of obtaining and transfering files between computers linked by an international computer network collectively know as the Internet. To obtain more information concerning ftp or the Internet, please consult operators of your local computing facility. The format used for site addresses and electronic mail addresses below conforms to that currently used on the Internet. I have included alternate means of obtaining the simulators where they have come to my attention.

---

bps - George Mason University Back Prop Simulator - Current version is 1.01 (Nov., 1989)

A special-purpose simulator for Back Propagation and a BP speedup technique called 'gradient correlation' [IJCNN, Jan., 1990].

Available via anonymous ftp from gmuvax2.gmu.edu (129.174.1.8).

Distributed as executables for VAX 8530 under Ultrix 3.0, and versions for 8088 based IBM PC, and 80286/386 IBM PC machines. Includes examples and a tutorial document. Source code license is available. Contact:
baselinskip=5mm

Eugene Norris
Computer Science Department
George Mason University
Fairfax, Virginia 22032
(703) 323-2713
enorris@gmuvax2.gmu.edu

Back Propagation simulator (Name and version not known.)

A special-purpose simulator for back propagation using several training methods.

Distributed on disk for the IBM PC, with mouse and VGA or EGA display.

To obtain it, send a stamped, self-addressed envelope and a floppy disk (3.5" or 5.25") to:
baselinskip=5mm

Universitt Kassel
Fachbereich Mathematik
Forschungsgruppe Neuronale Netzwerke
Heinrich-Plett-Str 40
3500 Kassel
West Germany

[Report taken from Internet News, October, 1989.]

MIRRORS/II - Maryland MIRRORS/II Connectionist Simulator A general-purpose connectionist simulator.

To obtain this simulator you must sign an institutional site license. A license for individuals is not acceptable. The only costs incurred are for postage for a printed copy of the manual and tape cartridge (you send your own 1/4" cartridge or TK50 cartridge to them, if desired.) Instructions for obtaining the software via

ftp are returned to you upon receipt of the license agreement. To obtain a copy of the license send your physical mail address via e-mail to:

mirrors@cs.umd.edu

or by U.S. Mail to:

Lynne D'Autrechy
University of Maryland
Department of Computer Science
College Park, MD 20742

MIRRORS/II is implemented in Franz Lisp and will run under Opuses 38, 42, and 43 of Franz Lisp on UNIX systems. It is currently running on a MicroVAX, VAX and SUN 3.

NeurDS – The Neural Design and Simulation System. Current Version is 3.1 (May, 1989.)

A general purpose simulator.

The system is licensed on a no-fee basis to educational instutions by Digital Equipment Corporation. To obtain information, send your physical or electronic mail address to:

Max McClanahan
Digital Equipment Corporation
1175 Chapel Hills Drive
Colorado Springs, Colorado 80920-3952
mcclanahan

You should receive instructions on how to obtain a copy of the manual and copies of the license agreement. [Beyond receipt of the license agreement, I do not know the details of distribution.]

The NeurDS system will run on any Digital platform including Vax/VMS, Vax/Ultrix, and DECsystem/Ultrix. A graphics terminal is not required to support the window interface.

Specific models are described using a superset of the C programming language, and compiled into a simulator form. This simulator can accept command scripts or interactive commands. Output can take the form of a window-type envirionment on VT100 terminals, or non-window output on any terminal.

FULL – Fully connected temporally recurrent neural networks. A demonstration network described in "Learning State Space Trajectories in Recurrent Neural Networks" [no other reference material!]

The author (whose name is Barak Pearlmutter of the Journal of Neural Computation, 'bap@f.gp.cs.cmu.edu') describes this as "a bare bones simulator for [. . .] temporally recurrent neural networks" and claims that it should vectorize and parrallelize well. It is available for ftp from doghen.boltz.cs.cmu.edu. Login as 'ftpguest' password 'oaklisp'. Be sure to ftp as binary for the file 'full/full.tar.Z' (you must either use a directory named full on your local machine, or use 'get' and let it prompt you for remote and local file names.) Do not attempt to change directories. It is copyrighted and is given out for academic purposes.

[The information dates from November of 1988.]

GRADSIM Connectionist Network Simulator. A special-purpose simulator specifically designed for experiments with the temporal flow model.

Latest Version 1.7

The simulator is available for anonymous ftp from ai.toronto.edu (128.100.1.65). For information contact:

Raymond Watrous
Department of Computer Science
University of Toronto
Toronto, Ontario M5S 1A
4 watrous@ai.toronto.edu

In C, implementations on VAX (VMS & Ultrix), Sun and CYBER are mentioned. A graphical interface is 'under development.' (March, 1988.) Includes an excellent article with references.

opt – A Neural-Net Training Program Based on Conjugate-Gradient Optimization. A special-purpose simulator. [Current version unknown.]

Available for anonymous ftp from cse.ogc.edu (129.95.40.2) from the directory "/ogc2/guest/ftp/pub/nnvowels". Consult the file 'README' for more instructions. For further information, you might contact 'opt-dist@cse.ogc.edu'.

Basically C code to be compiled under BSD Unix, with no graphic interface. They do maintain a list of users, perhaps a mailing list. An unsigned paper describing the technique and use of the simulator is included.

CasCor1 – Cascade-Correlation Simulator A special-purpose simulator for experimenting with the Cascade-Correlation algorithm described in:

Fahlman, Scott, and C. Lebiere. "The Cascade-Correlation Learning Architecture." In Advances in Neural Information Processing Systems 2, edited by D. S. Touretzky. New York : Morgan Kaufman Publishers, 1990.

It is available for anonymous ftp from pt.cs.cmu.edu (128.2.254.155) in the directory "/afs/cs/project/connect/code" (subdirectories may be available, but parent directories may not be.) There are Lisp and C versions available, as well as several other programs. The simulator is placed in the public domain. For information contact:

Scott E. Fahlman
School of Computer Science
Carnegie-Mellon University
Pittsburgh, PA 15217
fahlman@cs.cmu.edu

The original version by Scott Fahlman was written in Common Lisp and has been tested on CMU Common Lisp on the IBM RT, Allegro Common Lisp (beta test) for Decstation 3100, and Sun/Lucid Common Lisp on the Sun 3. This program was translated into C by Scott Crowder.

GENESIS - GEneral NEural SImulation System with XODUS - X-windows Output and Display Utility for Simulations - A general simulator. Currently Beta-Test Version, 1990.

From the release announcement (January, 1990 by Jim Bower):

"Full source for the simulator is available via FTP from genesis.cns.caltech.edu (131.215.135.64). To acquire FTP access to this machine it is necessary to first register for distribution by using telnet or rlogin to login under user "genesis" and then follow the instructions (e.g. 'telnet genesis.cns.caltech.edu' and ' login as 'genesis'). When necessary, tapes can be provided for a small handling fee ($50). Those requiring tapes should send requests to genesis-req@caltech.bitnet. Any other questions about the system or its distribution should also be sent to this address.

GENESIS and XODUS are written in C and run on SUN and DEC graphics work stations under UNIX (version 4.0 and up), and X-windows (version 11). The software requires 14 meg of disk space and the tar file is approximately 1 meg.

The current distribution includes full source for both GENESIS and XODUS as well as three tutorial simulations (squid axon, multicell, visual cortex). Documentation for these tutorials as well as three papers describing the structure of the simulator are also included. As described in more detail in the "readme" file at the FTP address, those interested in in developing new GENESIS applications are encouraged to become registered members of the GENESIS users group (BABEL) for an additional one time $200 registration fee. As a registered user, one is provided documentation on the simulator itself (currently in an early stage), access to additional simulator components, bug report listings, and access to a user's bulletin board. In addition we are establishing a depository for additional completed simulations.

SunNet - A generalized simulator. Version 5.5.2.4

Available for anonymous ftp from boulder.colorado.edu (128.138.240.1).

While this program was obviously written for Sun workstations (versions for suntools and the X-window envirionment), the documents list other configurations. These include a non-graphic version which runs on "any UNIX machine", and versions which run on an Alliant or UNIX machine and send data to a graphics support program running on a Sun workstation. It is very easy to install.

A mailing list exists for users of the simulator.

RCS - The Rochester Connectionist Simulator - A general simulator. Version 4.2

Available for anonymous ftp from cs.rochester.edu (192.5.53.209). Tapes may be purchased (1600 bpi 1/2" reel or QIC-24 Sun 1/4" cartridge) from:

Peg Meeker
Computer Science Department
University of Rochester
Rochester, New York 14627

C source code is provided, including a graphic interface which may function under X Windows or SunView on Sun Workstations. A wide variety of Unix machines are supported, and the simulator may be used without the graphics interface. A version for the MacIntosh is included in the distribution. Mailing lists exist for users and bug reports.

SFINX – Structure and Function in Neural Connections - A General Simulator. Version 2.0 (November, 1989)

In order to ftp this simulator, a license agreement must be submitted. Upon receipt of this agreement, instructions and the password to ftp the software are made available. To obtain the license write:

Machine Perception Laboratory
Computer Science Department
University of California
Los Angeles, CA 90024

This system requires color to operate the graphics interface, but may be operated without graphics. Support for Sun, Ardent Titan, HP 300, and IBM PC RT machines is specifically mentioned –but other Unix platforms should function as well. Specific graphics support is provided for Matrox VIP 1024, Imagraph AGC-1010P, HP Starbase and X Windows. A version providing support for monochrome graphics is expected to be released in Fall, 1990.

Mactivation - A specialized simulator for investigating associative memory using the delta rule and Hebbian Learning. Version 3.2

A public domain version is available for anonymous ftp from the University of Colorado at Boulder (boulder.colorado.edu, 128.138.240.1) or possibly by contacting the author.

Mike Kranzdorf
University of Colorado
Optoelectronic Computing Systems Center
Campus Box 525
Boulder, Colorado 80309-0525
mikek@boulder.colorado.edu

Future versions will probably not be public domain, but will be availble from Oblio, Inc., 5942 Sugarloaf Road, Boulder, Colorado 80309.

Provided as executables for the Apple MacIntosh.

Several special purpose simulators are provided with the following book:

McClelland, J. L., and David.E. Rumelhart. Explorations in Parallel Distributed Processing. Cambrige: MIT Press, 1988.

Versions exist which contain C code for the IBM PC, and a version has recently been released for the Apple MacIntosh.

Hopfield-style Network Simulator - A Special Purpose simulator for experimentation with the Hopfield-style network developed by the author.

Software is available by e-mail upon request from the author, Arun Jagota (jagota@cs.buffalo.edu). It is written in C and should be useful on 32-bit Unix machines, and a MSDOS version is also supplied.

Several demonstration-type simulators have been published as source code in various journals. These are cited below:

Brown, Robert Jay. "An Artificial Neural Network Experiment." Dr. Dobb'sJournal (April, 1987) 16ff.

Colvin, Gregory. "Synapsys: A Neural Network." The C Users Journal (April, 1989) 59ff.

King, Todd. "Using Neural Networks for Pattern Recognition." Dr. Dobb'sJournal (January, 1989) 14ff.

Klimasauskas, Casey. "Neural Nets and Noise Filtering." Dr. Dobb's Journal (January, 1989) 32ff.

Jones, Wiliam P., and Josiah Hoskins. "Back-Propagation." Byte (October, 1987) 155ff.

APPENDIX B

Test results in Tabular form

Simulator: GENESIS Run Date: Wed Aug 8 12:35:33 EDT 1990

Input Bits	Total Layers	Total Nodes	Make List *	Setup Time [1]	Cycle Time	User Time	Kernal Time	Elapsed Time	Shared Memory	Unshared Memory	Page Faults
4	10	45	0.1	3.10	0.005	13.5	1.1	0:24 58%	208	304	80
8	12	81	0.2	5.12	0.005	19.6	0.7	0:27 73%	216	328	24
16	14	149	0.3	9.06	0.009	31.5	1.1	0:39 81%	216	360	24
32	16	281	0.9	17.04	0.019	54.9	1.6	1:04 87%	224	416	24
64	18	541	2.1	34.84	0.035	107.2	2.5	1:57 93%	224	512	24
128	20	1057	6.6	76.30	0.068	233.8	3.8	4:06 96%	224	712	24
256	22	2085	27.0	180.46	0.134	574.0	7.8	9:51 98%	224	1136	24
512	24	4137	145.2	470.26	0.271	1605.6	15.3	27:19 98%	200	1784	24
768	24	6177	423.1	847.54	0.396	3075.9	25.2	51:56 99%	152	2120	31
1024	26	8237	951.7	1367.82	0.531	5050.4	34.8	1:25:07 99%	128	2200	114
1280	26	10285	1787.2	1958.04	0.665	7432.8	43.5	2:05:09 99%	120	2216	189
1536	26	12333	2971.3	2646.24	0.810	10266.8	57.7	2:52:46 99%	112	2280	425
1792	26	14381	4636.1	3454.76	0.960	13559.2	72.8	3:48:38 99%	112	2296	519
2048	28	16433	6911.1	4443.64	1.090	17419.3	92.1	4:52:49 99%	112	2320	838

* - This is the user and kernal time of the network description file program.
1 - This time doe not include that used in producing the network description file.

Simulator: RCS Run Date: Mon Jul 23 18:03:52 EDT 1990

Input Bits	Total Layers	Total Nodes	Make List	Setup Time	Cycle Time	User Time	Kernal Time	Elapsed Time	Shared Memory	Unshared Memory	Page Faults
4	10	45	0.02	0.24	0.0165	0.9	0.3	0:03 42%	48	48	23
8	12	81	0.02	0.22	0.0345	1.7	0.2	0:02 73%	56	56	7
16	14	149	0.06	0.38	0.0755	3.7	0.2	0:04 80%	64	72	7
32	16	281	0.14	0.68	0.1680	7.9	0.3	0:09 91%	64	80	7
64	18	541	0.36	1.30	0.3755	17.4	0.2	0:18 95%	64	112	7
128	20	1057	1.04	2.54	0.8250	38.4	0.4	0:39 97%	64	160	7
256	22	2085	3.80	5.10	1.8235	85.1	0.3	1:26 98%	64	264	7
512	24	4137	15.80	10.28	4.0115	196.0	0.7	3:18 99%	64	480	7
768	24	6177	35.94	15.00	6.0045	305.6	0.8	5:08 99%	64	672	7
1024	26	8237	69.04	20.90	8.8030	462.4	0.9	7:45 99%	64	864	7
1280	26	10285	106.40	25.64	11.0230	596.0	1.3	10:00 99%	64	1040	7
1536	26	12333	159.48	30.48	13.2115	835.3	1.6	14:00 99%	64	1224	7
1792	26	14381	224.64	36.02	15.3825	909.5	1.3	15:14 99%	64	1344	7
2048	28	16433	313.12	42.50	19.2265	1169.8	1.7	19:35 99%	64	1528	7

Simulator: SFINX Run Date: ThJul 26 11:09:37 EDT 1990

Input Bits	Total Layers	Total Nodes	Make List	Setup Time	Cycle Time	User Time	Kernal Time	Elapsed Time	Shared Memory	Unshared Memory
4	10	45	0.5	0.06	0.0465	2.1	0.0	0:03 72%	56	40
8	12	81	0.7	0.06	0.1085	4.5	0.2	0:04 98%	56	48
16	14	149	1.3	0.08	0.2375	9.7	0.2	0:10 98%	56	48
32	16	281	2.3	0.12	0.5285	21.3	0.1	0:21 98%	56	48
64	18	541	4.5	0.18	1.1751	47.3	0.2	0:47 99%	56	64
128	20	1057	9.5	0.26	2.6236	105.6	0.2	1:46 99%	56	80
256	22	2085	24.1	0.46	5.8733	235.6	0.4	3:56 99%	56	120
512	24	4137	64.1	0.94	13.0532	523.9	0.6	8:46 99%	56	192
768	24	6177	121.7	1.58	19.5010	782.8	0.8	13:06 99%	56	272
1024	26	8237	208.2	1.84	28.9150	1159.8	0.9	19:23 99%	56	352
1280	26	10285	304.5	2.30	36.1238	1448.9	1.1	24:13 99%	56	432
1536	26	12333	415.1	2.60	43.3317	1737.7	1.5	29:04 99%	56	512
1792	26	14381	559.0	3.22	50.4670	2024.9	1.3	33:51 99%	56	584
2048	28	16433	758.6	3.96	63.7562	2556.4	2.0	47:28 89%	56	680

There were no page faults.
* - Total user + kernal time to build text file and assemble it into binary file.
1 - Does not include time required to build network file.

Simulator: custom Run Date: Tue Jul 24 16:58:06 EDT 1990

Input Bits	Total Layers	Total Nodes	Make List	Setup Time	Cycle Time	User Time	Kernal Time	Elapsed Time	Shared Memory	Unshared Memory
4	10	45	0.02	0	0.0090	0.4	0.1	0:00 68%	32	16
8	12	81	0.02	0	0.0175	0.8	0.0	0:00 91%	32	16
16	14	149	0.06	0	0.0345	1.5	0.1	0:01 97%	32	24
32	16	281	0.14	0	0.0690	3.1	0.1	0:03 98%	32	24
64	18	541	0.38	0	0.1365	6.5	0.0	0:06 98%	32	40
128	20	1057	1.14	0	0.2780	14.5	0.1	0:14 99%	32	64
256	22	2085	4.16	0	0.5680	31.3	0.3	0:31 99%	32	104
512	24	4137	17.38	0	1.1730	74.0	0.2	1:15 98%	32	184
768	24	6177	39.80	0	1.7325	126.6	0.2	2:07 99%	32	256
1024	26	8237	77.20	0	2.3845	201.5	0.4	3:22 99%	32	328
1280	26	10285	119.26	0	2.9760	277.5	0.5	4:38 99%	32	392
1536	26	12333	180.06	0	3.5780	366.4	0.8	6:08 99%	32	448
1792	26	14381	253.86	0	4.1605	484.7	0.7	8:06 99%	32	504
2048	28	16433	346.92	0	4.8995	718.0	1.1	12:00 99%	32	600

There were no page faults.

VITA

Charles Weldon Joyce, Jr. was born in Winston-Salem, North Carolina on December 21, 1951. His early degrees are in acting (BFA, University of North Carolina, 1972) and dramatic theory and criticism (MA, 1977). During this time he was employed as a professional actor and stage manager. Acquiring a third degree in arts administration (MA, University of Cincinnati, 1979) he was, for a time, employed in the management of theatre groups and arts councils. After being employed by a public radio station, he founded and managed Artsware, a firm which placed personal computers in small arts organizations and in the hands of individual artists and craftspersons. It was this business which convinced him to pursue the Master of Science degree in Computer Science at the University of Tennessee, which he expects to receive in the Fall of 1990.