Computer Science Department Technical Report
Machine Perception Laboratory
University of California
Los Angeles, CA 90024-1596

UCLA SFINX - NEURAL NETWORK SIMULATION ENVIRONMENT

Eugene Paik
Josef Skrzypek

December 1989
CSD-890076

UCLA SFINX - Neural Network Simulation Environment

Eugene Paik

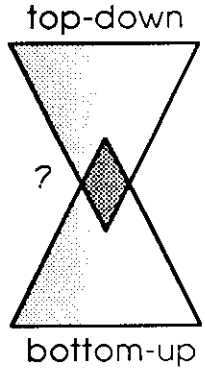
Josef Skrzypek

MPL

Machine Perception Lab

UCLA
Computer Science
Department

CRUMP
Institute for
Medical
Engineering



1 Introduction

Although the task of developing perception in a machine is a formidable endeavor, nature provides some clear directions on how to approach the problem. Evidence from neurophysiology and other related fields in neuroscience demonstrates that the basic organization of biological computing substrates is based on a highly distributed network of neurons. The advantages of this approach are:

real-time performance - Although the individual elements maybe relatively slow, the network produces the necessary computing power through massive parallelism. Improved performance does not result from an arbitrary interconnection scheme, but rather from the synthesis of functionality from specific connectivity patterns.

fault tolerance - The likelihood of component failure increases with the size and complexity of the system. Therefore, large complex systems must be designed to withstand and compensate for the possible malfunctions of individual components. The distributed nature of neural systems seems to naturally afford the redundancy of representation and plasticity needed to produce graceful degradation in cases of local system malfunctions or erroneous stimuli. This type of tolerance is not readily realized in centralized, Von Neumann style machines.

Furthermore, whereas traditional computer science approaches have separated hardware and software for engineering simplicity, nature has integrated them for performance and efficiency. Nature has coupled the structure and function of its hardware to maximize its performance, thereby improving odds of survival. This suggests that future solutions to perception should be based on a close match between function and underlying architecture. A desired function must dictate the optimal computing architecture. This paradigm of distributed representation and computation may provide not only improvements in computing speed, but also the conceptual tools necessary to realize perception in machines.

Autonomous robots operating in unconstrained environments need to perceive and interact with their surroundings in "real-time". To realize these perceptual tasks, the underlying computing substrates of these systems must be capable of continuously organizing and interpreting massive streams of multi-sensory data. Attempts to engineer systems using just a single sensory modality, for example artificial vision, have shown that current sequential and pipeline computers cannot meet the demands required by real-time segmentation of natural scenes.

What precisely is meant by a neural networks? Despite considerable progress, neuroscience does not as yet have a complete explanation of how the nervous system can produce cognition and perception. Indeed, the technical difficulties in recording neural activity, not to mention the the sheer complexity of the nervous system, pose a great challenge to neuroscience. It is becoming clear that one of the ways to accelerate the progress of neuroscience is to synthesize models of neural networks under the constraints of current neurophysiological data. In this mode of research, artificial neural networks are

developed and studied by simulating them on digital computers. The insights gained from simulations suggest future directions of experimentation in natural systems. The success of this approach depends on close interactions between researchers studying biological systems and those attempting to build artificial ones.

SFINX, Structure and Function In Neural Connections(X), is a neural network simulation environment designed to provide the investigative tools for studying the behavior of various neural structures. It was developed at the UCLA Machine Perception Laboratory.

1.1 Other Neural Simulators and SFINX

One of the central issues when designing a neural net simulator is to what resolution a neuron should be modelled. As with any physical phenomenon, there exists a continuum of abstraction levels. In the case of neural systems, it can range from neurotransmitters within the synaptic cleft to propagation of action potential in neural membrane to mathematical or stochastic models representing the overall behavior of a group of neurons[1].The appropriate level of abstraction must be based upon the goal of the modeler.

PABLO is an example of a simulator that provides precise modeling of neurons and their interactions[2]. Its environment allows for detailed modeling of many known properties of soma bodies, dendrites, and axons. A probabilistic error generator was included in each neuron to model the occasional failures of signal transmission through an axon. PABLO's similarities to neurophysiology allows one to determine the rough functionality of a neural network that was being analyzed physiologically. PABLO, written in Fortran, is a discrete-event simulator and considerable effort must be expended during a simulation cycle in determining the most critical, next event. Also, it is a batch oriented system, allowing only for the initial loading of a group of neurons and their interconnectivity patterns. Finally, debugging functional and/or connection problems is limited to printer plots of neuron activity.

BOSS, another discrete-event simulator implemented in Fortran, was designed to investigate large neural networks[3]. In contrast to PABLO, where each individual neuron was specified and interconnected, BOSS forms a statistical representation of the connectivity pattern. This allows for relatively fast simulation of large connectivity patterns. Another useful property of BOSS is that a large network could be generated by using a few parameters. Consequently, the architecture of connectivity is locally regular. This feature, while a time saver in synthesizing large networks, placed limits on the connectivity patterns that could be explored.

The above simulators were both of a batch type. In contrast, ISCON explored the advantage of an interpreted simulator as a network construction tool[4]. Written in LISP, users were able to dynamically change network connectivity and restart the simulation. Unfortunately, the penalty is that even small examples take prohibitively long to execute. ISCON also offers the clarity of a graphics output environment. Debugging interconnectivity problems is simplified through the use of time traces on a node's activity or excitation level. Extending this capability to display the two-dimensional activity of a layer of nodes as they are processing an image would help in debugging and verifying the correctness of a connectionist architecture.

To increase speed while maintaining flexibility, ISCON evolved into the Rochester Connectionist Simulator[6]. RCS is a run-time environment written in C that allows user written programs to access a library of connectionist type functions, e.g. building networks, setting potentials, examining nodes. Users must develop their own graphics interface and debugging tools. Also, since the network topology is formed by the same program that runs the simulation, changing the connectivity requires halting the simulator, editing a C program, and recompilation.

UCLA PUNNS[7] served as the foundation for the development of SFINX. PUNNS offers a multifaceted, interactive simulation environment. The underlying primitive was an idealized, lumped parameter model of a neuron. A language was developed that defined each individual node by specifying the input to that node and the C function that would process the converging input. Particular attention was given to maintaining the temporal consistency of node firing, so that the time varying properties of neural networks could be explored. After repeated experimentation on realistic, grey-level images, it was found that PUNNS' focus on individual node definitions resulted in prohibitively long network load times. The primary bottleneck of loading was in parsing of the network description language, which was implemented in YACC and LEX.

SFINX is an interactive simulation environment offers tools to investigate the behavior of various structures of interconnected digital computing elements that operate synchronously. SFINX accepts networks of arbitrary structures. In addition, the regularity in connectivity patterns that may exist in large multi-layered networks can be efficiently defined and simulated using special constructs. SFINX has been implemented in a manner which allows its user to systematically customize the simulation environment according to simulation need and the capabilities of available computing environment.

SFINX has been used to simulate neural networks for the segmentation of images using textural cues[7], color constancy in low level vision[8], lightness constancy, and shape recognition[9].

2 Overview of SFINX

SFINX is a simulation environment for studying computation with neural networks. In simple terms, a SFINX neural network can be viewed as a set of computing elements called **nodes** that are connected or **linked** together. The links define the flow of data among the nodes. A node is made up of the following components:

memory
: Each node has certain capacity to store information. The state of a node refers to the contents of its memory at a particular instance in time.

function
: The node function specifies the node output and how its memory is modified based upon the node's input and history. A node can receive input from other nodes to which it is linked or from sources outside the network.

a set of links
: A node may have connections to other nodes, including itself, from which it receives input. For example, if node **A** is linked to node **B** and **C**, then the outputs of nodes **B** and **C** at time t are the input to node **A** at time t+1.

Structure of a network refers to the memory capacity of each node and the connectivity pattern of the links. *State* of a network refers to the contents of all the memory at a particular instance in time.

2.1 Outline of the Design

SFINX was developed as a practical tool for investigating various neural structures. Hence, flexibility and efficiency have been of primary importance in the overall design. SFINX is organized in a modular fashion, so that additional modules may be systematically added. A module is a set of commands which make a logical group. For example, a particular form of a graphics interface maybe seen as a module. Currently, there are various paradigms in neural network investigations. Investigators with different points of emphasis may be able to utilize the basic SFINX environment by developing the necessary modules. Steps for installing additional modules are documented in Section 4.5.

Node functions are implemented as user provided C routines. The user must write the C routines representing each node function. These C routines are then linked into the SFINX program. This approach forces greater responsibility upon the user; however, it also provides maximum efficiency and flexibility. This approach eliminates arbitrary restrictions placed on the classes of networks available for SFINX. Steps for generating node functions are documented in Appendix IV and V).

Currently, SFINX supports two different network representations, each with a trade-off between efficiency and flexibility. The most flexible representation, termed **explicit networks** (EN), can encode arbitrary connectivity patterns, but require the most space and time for simulation. The counterpart, termed **function arrays** (FA), are suited for networks with high degree of regularity in their connectivity pattern. In theory, one form is not more general than the other; however, in practice, FA usually produce significant improvement in efficiency, but preclude networks with irregular connectivity patterns.

Node memory is implemented as a set of registers whose type, henceforth called **REG_TP**, is one of int, short, float, or double types of the C programming language. All registers in a network must be of same REG_TP. In order to change the REG_TP, one must recompile the simulator using an appropriate macro setting (Appendix II).

2.2 System Requirements

SFINX has been specifically implemented to provide clean portability for most Unix† operating systems, such as System V, Berkeley 4.2 or 4.3. The core module, which consists of the basic SFINX tools based on a textual interface, requires a standard C compiler, a paging program (e.g. /bin/more), and signal traps. The core module should be compilable with little difficulty under most Unix environments. For specific details on porting SFINX, see Appendix II. Currently, SFINX has been ported to HP 9000 320/350 workstations running HP-UX, Sun workstations running BSD 4.3 and IBM RT workstations running BSD 4.2.

2.3 Main Data Structures

This section introduces the basic data structures used in SFINX. Familiarity with these structures may help understanding the pages to follow.

The **buffer** is a two dimensional array of REG_TP and serves a dual purpose. First, they can be used as I/O buffers for networks. Second, they are used to represent function arrays. Commands **showbuf** and **setbuf** allow the user to display and set the contents of the buffers.

In addition to function arrays, SFINX has another network representation called explicit networks. In this scheme, each network link is explicitly represented in the data structure. This data structure is composed of several C structures that are highly interconnected. At this point, it is sufficient to say that there is a single array for each of these structures. All pointers between these structures are implemented using array indices; no absolute pointers are used. As a result, since run time dependent values are not used, a network state can be easily saved and reloaded into the simulator.

When the simulator is invoked, the user must specify the maximum dimension of the buffers and maximum number of nodes, links, and registers. This approach greatly simplifies the memory management scheme used in SFINX. In essence, during the initial

---

† Unix is a trademark of AT&T Bell Laboratories.

phase of the simulator, a large block of system memory is reserved for Explicit Networks and buffer space. If these parameters must be enlarged, SFINX must be reinvoked using appropriate values.

A list of **environment variables** is maintained by SFINX. Associated with each variable is an integer value. These variables reflect the current state of the simulator. Some variables modify the behavior of commands. The list of environment variables are fixed during compilation and cannot be modified during runtime. Their values may be displayed and modified by the command **set**.

SFINX also provides a separate list of variables called the **user variables**. Here, the user may instantiate arbitrary variables. Associated with each variable is a string value. These variables are intended to provide a flexible way of adjusting the parameters of node functions. Since it is usually unreasonable to recompile and relink a node function each time a parameter is changed, the user may write a generalized node function so that its parameters can be set by predesignated user variables. In essence, a node function would first check for the existence of certain variable. If it exists, it would override its default parameter value. This mechanism can be very effective in efficiently simulating a network with various parameters. The commands **setu** and **unsetu** are used to display and modify this list.

A library of C routines contains all the routines that represent node functions. Actually, there is a separate library for explicit networks and function array node functions. The functions in these libraries can be displayed by the commands **enfn** and **fafn**.

2.4 Simulation Environment

The basic organizational structure of SFINX is analogous to traditional programming language compilers (see Figure 1). The first step in SFINX simulations is to create a network representation. For Explicit Networks, this first step entails transforming a textual description of a network into an equivalent SFINX binary data structure. This structure need be generated only once, after which it can be loaded and reloaded into the simulator. The SFINX ASSEMBLER program (SASS), described in Appendix III, provides the transformation. Function Arrays can be created simply by placing the appropriate values in buffer locations.

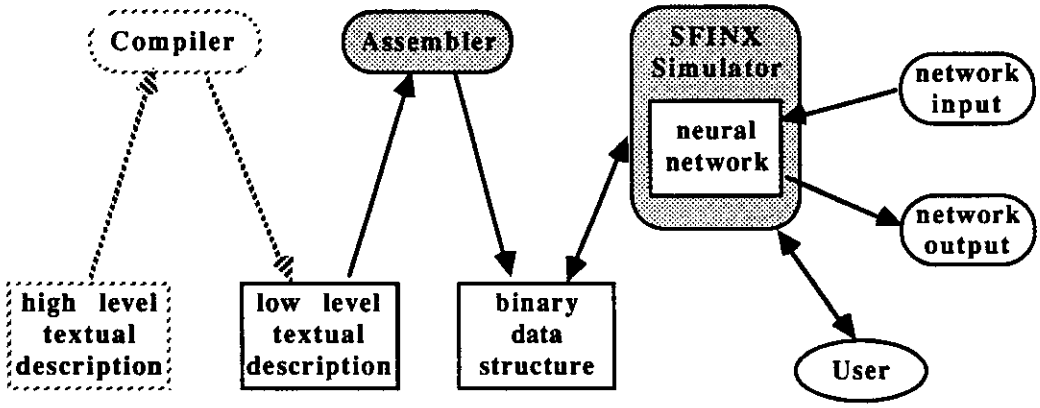


Figure 1. **SFINX Environment**

One of the goals at MPL is to develop a higher level language for describing neural networks. In this case, a Compiler would transform this higher level, perhaps functional, description of a system into a low level structural description of a network. Currently, no prototype exists. However, continued experimentation with network structures using low level descriptions may produce enough patterns and consistencies to develop such a language.

2.5 Basic Steps for Using SFINX

The SFINX user interface is handled by a command line interpreter which can be run in interactive or batch mode. By changing the initial state of the network, the same network structure can be re-executed with various parameters. The basic steps of using the SFINX simulator can be summarized as follows:

1. Define a SFINX network (see Network Representation).
2. Invoke the SFINX program (see Appendix I).
3. Load the network into the simulator.
4. If necessary, modify the initial state of the network.
5. If necessary, load the input to the network.
6. Execute the network.
7. Display and/or save the state of the resulting network.
8. If network is to be simulated with different parameters, goto 4.

Usually, a simulation session requires numerous commands and is often cumbersome to repeatedly retype similar commands. In such cases, the user may create a text file, called SFINX script file, containing the sequence of commands. Script files are executed by command **exec**. These shell scripts act as meta-commands representing a sequence of low level commands. Also, they are a natural way of recording the precise actions taken for a simulation.

3 Network Representation

This chapter describes the details of how networks are actually represented within SFINX. Two types of representations are supported, **Explicit Networks** (EN) and **Function Arrays** (FA). In theory, any network structure can be encoded in either form. However, in practice, FA format is suited only for those networks whose topology can be viewed as two dimensional layers of nodes which are interconnected by highly regular pattern of links. On the other hand, EN can handle arbitrary connectivity patterns, but they are also less efficient in terms of space and execution time. Thus, whenever feasible FA format is preferable to the EN counterpart. For networks that contain both regular and irregular connectivty patterns, it is also possible to create a hybrid network representation which uses both formats.

3.1 Explicit Networks

Explicit Networks use data structures which closely parallel the network topology. Each network link is explicitly represented by the data structure. The user should be aware of the following fields (see Figure 2):

node id
: Each node has a unique three integer identification number, which is assigned by the user when creating the network. It is often convenient to use these numbers as implicit pointers to buffer locations.

function register (FR)
: FR points to an entry in the node function library. For each node function, there are actually two C functions: the actual node function (nfn) and the initializing node function (infn). Infn is executed only once at the beginning of each simulation cycle, i.e. a single iteration of all the nodes. The primary duty of this function is to check the user variables for possibly overriding the default values of the node function parameters. The duty of the actual node function is to compute the node's output value. This value is the return value of the function. The node function may also modify the contents of GRs and LRs (see below) by side effect.

output register (OR)
: OR is a special register containing the value of the node's current output.

general registers (GR)
: During the creation of an explicit network, the user may reserve an arbitrary number of registers for each node. They are available to the node function and no restrictions are placed on their usage.

set of links

Each link points to a node from which input is received. It takes once simulation cycle for the output of the linked node to arrive at the linking node. Once an EN is created, links cannot be modified, removed, or added. To simulate the effect of adding and deleting links, one must first create a network whose connectivity pattern encompasses all the desirable links. LRs (see below) can then be used to signify opened and closed links.

link registers (LR)

During the creation of an explicit network, the user may reserve an arbitrary number of registers for each link. They are made available to the node function and no restrictions are placed on their usage. Since both GR and LR are general purpose memory location and are available to the node function, they could have been collapsed into a single class. However, this separation may help simplify the implementation of some node functions. A natural usage of LR is link weights. They can also be used to implement a queue for simulating delays.

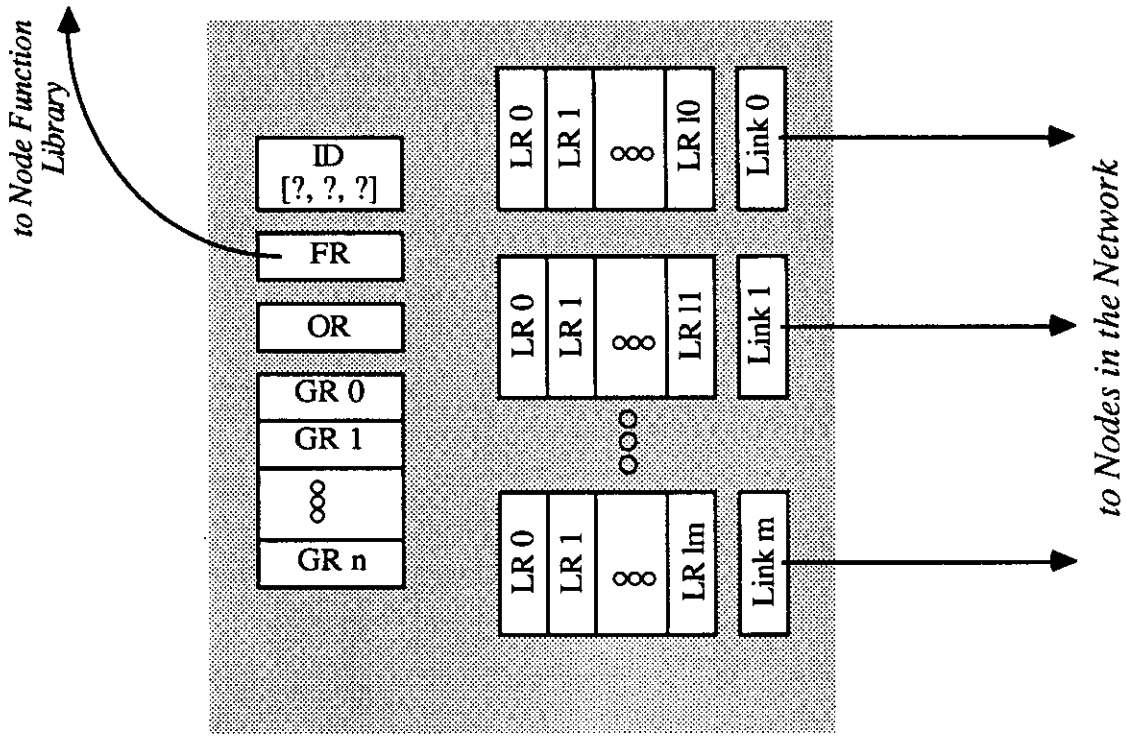


Figure 2. **Explicit Node**

3.1.1 Adding EN Node Functions

Steps for adding EN node functions into the simulator library are as follows:

1. Create a C source file for the node function (Appendix IV).

2. If needed, create a C source file for the initializing node function. It is important to remember that all initializing routines in the library are called for each simulation cycle or time step, even if the node function is not used in the network.

3. Add these functions to the list_enfn.c file, whose format is straightforward. If the initializer is unnecessary, then the ienfn field (see list_enfn.h and list_enfn.c files) can be set to NULL. Be sure to create a unique node function name and place the new function at the end of the list. Networks which have been saved contain merely the index within the library; therefore, the order of the preexisting functions should not be modified.

4. Add this file into the EN_FNS macro in the Makefile of the Working directory (see Appendix II).

5. Remake the program.

3.2 Function Arrays

Function Arrays are a very simple way of representing a network whose topology can be mapped on a two dimensional layer and whose connectivity pattern is highly regular. In this scheme, various buffer layers are designated as FR, GR, and LR. Whereas EN links are explicitly encoded in the network representation, FA links are implicitly defined within each node function. More specifically, the links are represented in terms of relative position with respect to the coordinates of the node.

For example, a simple 3x3 convolution can be implemented using an FA network, where the links can be encoded by 9 vectors: <-1, -1>, <-1, 0>, ..., <1, 1>. Thus, if a node is positioned at [z, 1, 2], then it would be linked to nodes at position [input_level, 0, 1], [input_level, 0, 2], ..., [input_level, 2, 3], where **z** and **input_level** are the z-coordinate of the node and input, respectively. Both of these values can be parameterized with user variables.

There is one more generalization. When a buffer layer is used to represent FA function registers, the command **runfa** is used. However, if an FA layer contains only one node function and the location of the nodes are evenly spaced throughout the layer, then the command can be used. This command generates a fictitious buffer layer of FRs. See **runlfa** command documentation for further details.

For EN, the simulator gathers the input from appropriate nodes. However, in the FA version, the node function must explicitly grab the input and save its output at predetermined buffer entries. In particular, extra care is required for FA with feedback, as the order of execution becomes critical in these cases. Furthermore, the user must carefully design the allocation of buffer space for various purposes, and this allocation scheme must then be well encoded in each FA node function. User variables can be used to parameterize these values (see Appendix V for an example).

3.2.1 Adding FA Node Functions

Steps for adding FA node functions into the simulator library are as follows:

1. Create a C source code for the node function (see Appendix V).

2. If needed, create a C source code for the initializing node function. It is important to remember that all initializing routines in the library are called for each simulation cycle, even if the node function is not used in the network.

3. Add these functions to the list_fafn.c file, whose format is straight forward. If the initializer is unnecessary, then the ifafn field (see list_fafn.h and list_fafn.c files) can be set to NULL. Be sure to create a unique node function name and place the new function at the end of the list. Networks which have been saved contain merely the index within the library; therefore, the order of the preexisting functions should not be modified.

4. Add this file into the FA_FNS macro in the Makefile of the Working directory (see Appendix II).

5. Remake the program.

4 SFINX Commands

4.1 Command Interpreter Syntax

The user interacts with a simulator through the command interpreter[1] which accepts a limited number of syntactic forms. The command interpreter takes a single command per line, except where the '\' character is present. The '\' character signifies that the command continues to the following line, and the rest of the line is considered as a comment. The ';' character begins a comment and also terminates the command line. These mechanisms are useful for commenting SFINX shell scripts. Each command can have zero or more arguments. Each argument is one of three types:

numeric value (val)
: This field must begin with '-', '.', or a numeric character.

string value (str)
: A string is a sequence of characters beginning with any printable character other than the characters mentioned above. It may also be surrounded by double quote characters to include spaces.

node id (nid)
: This field specifies a set of nodes of buffer entries. Note that both EN nodes and buffer locations are specified by three integers. This convention can be used for conveniently associating between nodes and buffer entries. Node and buffer specifications has the following syntactic format:

[first_range, second_range, third_range]

where each range can be of following forms:

empty	signifies all possible values.
start : end	signifies values between ***start*** and ***end***, inclusive.
start :	signifies values greater than or equal to ***start***.
: end	signifies values less than or equal to ***end***.
number	signifies a single value, the ***number***.

[1] A more user-friendly graphical front end based on X Window System is currently under development. X is a windowing system developed at Massachusetts Institute of Technology.

Therefore, [,,] represents all nodes. **[1, 0 : 127, 40:]** represents those nodes whose first id field equals 1; second field is between 0 and 127, inclusive; and the third greater than or equal to 40. Note that buffer entries are specified [z, y, x] and **not** [x, y, z]. This convention is used consistently throughout the system.

The list of SFINX commands are provided below. Optional arguments are enclosed in parentheses. Literals are printed in regular font. They maybe entered in upper or lower case. Italicized arguments, such as numeric values ***i, j,*** and ***val*** or string values ***str***, must be replaced by appropriate entries.

4.2 Core Commands

The following is a list of commands that makeup the core module of SFINX.

help

prints out the list of available commands with short descriptions.

exec ***str*** **(*val*)**

The contents of the file ***str*** is executed by the command interpreter ***val*** (default: 1) times. The exec files can be nested, but cannot be recursive.

load ***str***

str must be a SFINX network file created by the SASS program or the **save** command. This network is loaded into the simulator memory. The number of nodes and links must be less than or equal to the maximum specified by the environment variables MAX_NODES, MAX_LINKS, MAX_REGS. Also, the register type must be consistent with the REG_TP of the running simulator. If the network is too large for the current setting, the simulator must be reinvoked using appropriate **n**, **l**, and **r** arguments. See Appendix I.

save ***str***

saves the current state of the loaded network into a file ***str***, which can then be reloaded by command **load.**

loadb ***str val***

str must be an SFINX buffer file created by the **saveb** command. This file is loaded into the simulator buffer at layer ***val***. The size of the buffer in the file must be same as that defined by the environment variables MAX_BUFX, MAX_BUFY, and MAX_BUFZ. Also, the register type must be consistent with the REG_TP of the running simulator. If the buffer size does not match the current settings, the simulator must be reinvoked using the appropriate **x**, **y**, and **z** arguments. See Appendix I.

saveb ***str val***

saves the current values of the buffer layer ***val*** into a file ***str***, which can the be reloaded using the **loadb** command.

en ***nid*** **(L)**

displays the contents of the loaded network. If the literal argument **L** is given, then link information is also provided.

en *nid* FR *str*
sets the function register of nodes ***nid*** to point to the EN node function whose name is ***str***.

en *nid* OR *val*
sets the output register of nodes ***nid*** to ***val***.

en *nid* GR *i val*
sets the *i*th general register of nodes ***nid*** to ***val***.

en *nid* LR *i j val*
sets the *j*th link register of the *i*th link of nodes ***nid*** to ***val***.

buf (*nid*)
displays the contents of the buffer locations ***nid*** textually on the screen. If no argument is given, all buffer locations are displayed.

buf *nid val*
sets the buffer entries ***nid*** to ***val***.

buf *nid str*
sets the buffer entries ***nid*** to point to the FA node function whose name is ***str***.

enfn
displays the list of EN node functions.

fafn
displays the list of FA node functions.

runen (*val*)
simulates the currently loaded explicit network ***val*** (default: 1) cycles. If the environment variable SHOWRUN is set to a positive number, then the simulator notifies the user each time that that many nodes are simulated. This is useful for monitoring a simulation of large networks. All EN initializing node functions are called in each cycle.

runfa (*val*)
executes a FA of buffer layer defined by environment variable FA_LAYER ***val*** (default: 0) cycles. If the environment variable SHOWRUN is set to some non-zero number, then the simulator notifies the user every time that that many buffer nodes are simulated. All FA initializing node functions are called in each cycle.

runlfa *str* (*val*)
executes a single FA node function ***str*** as a Function Array for ***val*** (default: 1) cycles. This command creates a fictitious FA where all nodes of the layer have the same node function. All FA initializing node functions are called in each cycle.The command **fafn** displays the list of buffer functions in the library. The following environment variables are used:

FA_LAYER (default: 0)
defines the z coordinate of the fictitious function array.

FA_X0 (default: 0)

FA_Y0 (default: 0)
define the initial X and Y values.

FA_XX (default: 0)
FA_YY (default: 0)
define the maximum X and Y values.

FA_INCX (default: 1)
FA_INCY (default: 1)
define the incremental value of the valid FA node locations.

SHOWRUN
is set to some non-zero number, then the simulator notifies the user every time that that many buffer nodes are simulated.

For example, with the following setting:

FA_LAYER = 1
FA_X0 = 2
FA_Y0 = 3
FA_XX = 4
FA_YY = 7
FA_INCX = 2
FA_INCY = 3

would produce fictitious node id's of [1, 3, 2], [1, 3, 4], [1, 6, 2], and [1, 6, 4].

cpbuf ***(nid)***
copies the contents of the first Current Output register to a Buffer location corresponding to its node id offset by values of environment variables CPB_Z, CPB_Y, CPB_X (all defaults: 0).

set
Displays the current values of the environment variables.

set ***str*** ***(val)***
Sets the environment variable ***str*** to ***val*** (default: system default value).

setu
Displays the current values of the user variables.

setu ***str1*** ***(str2)***
Sets the user variable ***str1*** to ***str2*** (default: empty string). If the variable does not already exist, it is created. These variables are not used by the simulator; they are provided to allow the user to easily change the parameters of node and buffer functions. Typically, a these functions have default constants. The existence of a predesignated variable can be used to override these values.

unsetu ***str***
Deletes user variable ***str*** from the system. This is not the same as setting the value of the variable to a null string.

quit
Exits the simulator.

! ***str***

Execute a shell command ***str***.

4.3 Additional Modules

Two examples of SFINX modules are provided below. The basic organization of these modules can be used to implement other custom modules.

4.3.1 HIPS Commands

At the Machine Perception Laboratory, the HIPS[††] data format has been used to store digitized images. These images have been used as input and output to various networks using the following commands:

rdhips ***str*** **(*num*)**

Loads a HIPS image file ***str*** into the buffer layer ***num*** (default: 0). The image is positioned flush against the upper left corner.

wrhips ***str*** **(*num*)**

Saves the contents of the buffer layer ***num*** (default: 0) as a HIPS image file ***str***.

4.3.2 GRAPHICS Commands

The following is a graphics module for displaying network activation levels in the form of grey scale images. First, activation levels are placed in a buffer layer. Then, this buffer layer is displayed on a graphics display. Various spatial and intensity scaling options are available.

This module was first implemented on Hewlett-Packard workstations using HP Starbase graphics package. In the HP Starbase version, shell environment variables SB_OUTDEV and SB_OUTDRIVER are used to locate the graphics device and driver.

This module has also been ported to other graphics environments, including IBM RT using an Imagraph AGC-1010P card and SUN workstations using a Matrox VIP-1024 card.

clearw

Clears the graphics window.

draw **(*val*)**

Graphically displays a buffer ***val*** (default: 0). The following environment variables are in effect:

[††] HIPS, an acronymn for Human Information Processing Laboratory's Image Processing System, was developed at Human Information Processing Laboratory, Departrment of Psychology, New York University by Yoav Cohen, Michael S. Landy, M. Pavel, and George Sperling. HIPS must be obtained separately from New York Unversity.

W_STARTX, W_STARTY specifies the coordinate of the upper left corner of the display window on the screen.

PIXSIZE specifies the ratio of actual pixel to display pixel. With PIXSIZE set to 4, each buffer entry is represented by a 4-by-4 array of display pixels.

MINP_VAL, MAXP_VAL specifies the minimum and maximum thresholding of the actual image. The displayed image has 128 grey levels. Those actual image values less than MINP_VAL are mapped to the lowest intensity; those greater than MAXP_VAL to maximum intensity; those in between are linearly interpolated between these values.

4.5 Adding Commands

It may be advantageous on occasions to add customized commands. In fact, several commands maybe grouped as a logical module, as in the examples above. The basic steps for adding additional SFINX modules are as follows:

1. Generate a C source file for each command. It will be helpful to study one of the existing commands, such as **runen** and **en**. The convention for command source code file name is c_*cmnd*.c, where *cmnd* is the name of the command. This convention is not enforced; however, it is an effective way of organizing the numerous files of source code.

2. Create a *MODULE*_CMNDS macro for these c_*cmnd*.o files in the main Makefile, where *MODULE* is the name of the module.

3. Create a *MODULE*_LIBS macros for any necessary libraries which must be linked in for this module in the main Makefile.

4. Add the above two macro into the sfinx_core.o line in the Makefile.

5. Add a -D*MODULE* to the CFLAGS macro in the main Makefile.

6. Include the commands in the file list_cmnd.c, whose format is straightforward. Be sure to use a unique name.

6. Include any necessary environment variables in the file list_set.c. It is advised that conditional #ifdef and #endif be used to separate the environment variables of various modules.

If necessary, add appropriate productions reflecting the syntax of the new command in the file parse.y. The parsing for command line input is done command by command using yacc. The productions are straight forward and are available in the file parse.y. In essence, there are fixed syntactic patterns which are recognized by the parser. If new syntactic structure is required to implement a new command, corresponding production must be added in the parse.y and parse.h file.

All command source files should have the include files <stdio.h> and parse.h. If the command is to modify any SFINX global data structure, it should also include files

struct.h and extern.h, in that order. The main routine handling the command should look something similar to:

```
void  c_cmnd(cmnd)
CMND_TP *cmnd;
{
      /* declare local variables */

      case (cmnd.ctype) {

            case CMND_?:      /* a legal syntactic form     */

                        /* take appropriate action    */

                        break;

            case CMND_?:      /* another legal format       */

                        /* take appropriate action    */

                        break;

            default:          /* illegal syntactic format   */
                              /* for this command           */
                        /* print error message        */
                        return;
      }
}
```

The structure CMND_TP and the constants CMND_? are defined in the file parse.h. The ctype field defines the syntactic structure used to read the last command. Depending on the particular syntax structure, various other fields of cmnd structure are set to the values of corresponding command arguments.

Appendix I: Invoking SFINX

The simulator is invoked by typing **sfinx**, plus any additional arguments to override system's default values. The following arguments are supported:

n=*num*
l=*num*
r=*num*

sets the value of the environment variable MAX_NODES, MAX_LINKS, and MAX_REGS, respectively, to ***num***. These numbers correspond to the maximum size of an Explicit Network in terms of number of nodes, links and registers registers that can be loaded into the simulator. The default value of this variable can only be overridden during invocation of SFINX.

z=*num*
y=*num*
x=*num*

sets the value of the environment variable MAX_BX, MAX_BY, MAX_BZ to ***num***, respectively. This number corresponds to the size of the simulator buffer. The default value of this variable can only be overridden during invocation of SFINX.

i*str*

run the **exec** command of the file ***str*** before prompting the user. When more than one i arguments is given, the order of execution corresponds to the order of the arguments. However, no duplicate files are allowed. After the file is completed, the user is prompted using stdin. See command **exec** for further details.

Appendix II: Porting SFINX

Two directories are used in the distribution of source code for SFINX version 1.0. The main directory contains most of the parts including all of the commands. The **Makefile** in this directory makes the program **sass** and a file called **sfinx_core.o**. This file is generated by linking all of the components of the simulator, except for the node functions.

Within the main directory there is a subdirectory called **Working**, with its own Makefile. This Makefile links in the node function with sfinx_core.o to create a running version of the simulator. By combining most of the modules of sfinx in sfinx_core.o, the user's working directory does not need to contain the numerous SFINX source files. Usually, each user has his/her own Working directory with a private set of node function libraries.

The user can define the type for REG_TP by including in the CFLAGS macro one of the following:

-DI_REG	int	(the default)
-DS_REG	short	
-DF_REG	float	
-DD_REG	double	

***** It is important that this REG_TP definition of the Makefile in the main directory and Working directory are identical. *****

Furthermore, if REG_TP is modified, then all the files must be remade.

Appendix III: SASS

SASS, short for SFINX ASSEMBLER, converts a textual description of network structure into a SFINX Explicit Network data format. It receives the textual description through standard input. The first argument is used as the filename of the SFINX EN file. Following optional arguments are available:

n=*num*
l=*num*
r=*num*
: These numbers must be larger than the number of nodes, links and registers in the network being compiled. These options must be used when the default settings are too small.

The format of the network description is as follows:

for each node:

[node id] (initial OR) (# of GR) (values for each GR)

(# of links)

for each link:

[source node id]

(# of LR) (values for each LR)

The following is an example of 2 node network. The first node has id [1, 2, 3]; initial output value of 4, no general registers, and no links. The second node has id [3, 2, 1]; initial output value of 10, two general registers whose values are 11 and 22; and two links. The first link is to the first node and has 1 link register of value 21. The second link is to itself, and has no link registers.

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

            0


[3, 2, 1]   10

            2      11     12

            2      [1, 2, 3]   1      21

                   [3, 2, 1]   0
```

Appendix IV: Sample EN Node Function

The following is a sample En node function. This code demonstrates the use of most of the utility macros and functions available. This function is provided in the distribution file **enfn_sample.c** under the **Working** directory.

```
/*****************************************************************/
/*                                                               */
/* This is a pedagogical EN node function. The output of nodes   */
/* with this node function is "constant" times the sum of input.*/
/* The default value of the constant is 1; however, it may be    */
/* overridden by the user variable "enfn_sample_constant".       */
/* Also, various values which are available to the node function*/
/* are printed out.                                              */
/*                                                               */
/*****************************************************************/

#include <stdio.h>
#include "struct.h"
#include "extern.h"
#include <math.h>

static double     constant;

/* ============================================================ */
void  ienfn_sample() /* ================================== */
/* ============================================================ */
{
      char  buffer[128];

      constant = get_uvar_val("enfn_sample_constant", buffer) ?

                  atof(buffer) :    /* returned value */

                  1;                /* default value  */

      printf("enfn_sample constant is %d\n", constant);
}


/* ============================================================ */
REG_TP  enfn_sample(node, POR, input) /* ==================== */
/* ============================================================ */
      NODE_TP     *node;
      REG_TP      POR;
      REG_TP      *input;
{
      int   numl,       /* number of links in this node     */
            lcnt;       /* counter variable for links       */
      int   numr,       /* number of regs in this node      */
            rcnt;       /* counter variable for regs        */
```

```
int   numlr,              /* number of regs in this link*/
      lrcnt;              /* counter for link regs      */
REG_TP  total;            /* for totaling input   */


                  /* print the node id          */
printf("\n\nin nf_sample: ");
pr_id(N_Id(node));        /* a utility routine    */
printf("\n");

                  /* print the following:       */
                  /*   previous output          */
                  /*   number of links          */
                  /*   number of registers      */
printf("   POR %d; #link %d; #regs %d\n",
            POR,
            numl = N_NumL(node),
            numr = N_NumR(node));

                  /* print register contents    */
printf("   REGS: ");
for (rcnt = 0; rcnt < numr; rcnt++) {

                  /* assuming REG_TP = int/short*/
      printf("%d ", N_IthReg(node, rcnt));
}
printf("\n");

                  /* for each link print ...    */
for (total = lcnt = 0; lcnt < numl; lcnt++) {

                  /* print id of source node    */
      printf("          %d: ", lcnt);
      pr_id(N_IthSrcN(node, lcnt));

                  /* print input value and      */
                  /* number of link registers   */
      printf("  input %d;  %d LRegs: ",
                  input[lcnt],
                  numlr = N_NumLR(node, lcnt));

                  /* print register values      */
      for (lrcnt = 0; lrcnt < numlr; lrcnt++) {
            printf("%d ", N_IthLReg(node, lcnt, lrcnt));
      }
      printf("\n");

                  /* keep total of input values */
      total += input[lcnt];
}

            /* return the output value of this node   */
return (constant * total);
}
```

First, the initializing function **ienfn_sample** demonstates the basic method of writing general node functions whose parameters can be dynamically adjusted by the user. This function checks the user variable list for the existience of the variable **enfn_sample_constant**. If it does not exist, the function get_uvar_val returns FALSE, in which case the default value of 1 will be assigned to the variable **constant**. If it does exist, then the function returns TRUE; furthermore, the char array **buffer** will be set to the string value associated with the variable **enfn_sample_constant**. In this case, the string is converted to a floating point value via **atof()** and the resulting value is assigned to the variable **constant**.

Note also that the variable constant is declared as **static**. This serves to hide the variable from other source files.

The actual node function **enfn_sample** generates the output whose value is the **constant** times the sum of its input. This node output value is returned as the return value of the node function. The return type of all node functions must be REG_TP.

Each node function is passed three parameters: **node**, **POR**, and **input**. The **node** is a pointer to a special C struct used to represent a node in an Explicit Network. The user does not need to know the details of how it is defined, as macros provide a convenient interface. POR is of type REG_TP and its value is that of the node's previous output. The **input** is an array of REG_TP and the values correspond to the input from the nodes it is linked to. The order of the input values are that of the links. Thus, the first input value, **input[0]**, is from the first link.

As a side effect, it prints out various values available to node functions. The output of this node is similar to that of the **en** display command. Note that there are three very similar for-loops: one for going down the list of node registers, another for links, and finally one for link registers.

Appendix V: Sample FA Node Function

The following is a sample FA node function. This code demonstrates the use of most of the utility macros and functions relevant for FA function. This function is provided in the distribution file **fafn_sample.c** under the **Working** directory.

```
/*****************************************************************/
/*                                                               */
/* This is a pedagogical FA node function. There is a single     */
/* input located at the input layer of the Buffer with the same  */
/* x-y coordinate. The output location is also at the same x-y   */
/* coordinate of the output layer. The output value is constant  */
/* times the input value. The input level and output level and   */
/* constant make up the three parameters to this function and    */
/* can be set by user variables fafn_sample_input,               */
/* fafn_sample_output and fafn_sample_constant, respectively.    */
/*                                                               */
/*****************************************************************/

#include <stdio.h>
#include "struct.h"
#include "extern.h"
#include <math.h>

static      int         input, output;
static      double      constant;

/* ============================================================= */
void  ifafn_sample() /* ======================================== */
/* ============================================================= */
{
      char  buffer[128];

                        /* define constant              */
      constant = get_uvar_val("fafn_sample_constant", buffer) ?
                  atof(buffer) : 1;

                        /* define input layer           */
      input = get_uvar_val("fafn_sample_input", buffer) ?
                  atoi(buffer) : 0;

                        /* define output layer          */
      output = get_uvar_val("fafn_sample_output", buffer) ?
                  atoi(buffer) : 1;

      printf("ifafn: input (%d)  output (%d)  constant (%f)\n",
                  input, output, constant);
}


/* ============================================================= */
```

```
void  fafn_sample(idz, idy, idx) /* ========================== */
/* ============================================================ */
      int   idz, idy, idx;     /* position of the FA node        */
{
                               /* print the coordinate of ndoe   */
      printf("in fafn [%d, %d, %d]\n", idz, idy, idx);

                               /* output = constant * input      */
      Buffer(output, idy, idx) =
                         constant * Buffer(input, idy, idx);
}
```

First, the initializing function **ifafn_sample** demonstates the basic method of writing general node functions whose parameters can be dynamically adjusted by the user. This function checks the user variable list for the existience of the variable **fafn_sample_constant**. If it does not exist, the function get_uvar_val returns FALSE, in which case the default value of 1 will be assigned to the variable **constant**. If it does exist, then the function returns TRUE; furthermore, the char array **buffer** will be set to the string value associated with the variable **fafn_sample_constant**. In this case, the string is converted to a floating point value via **atof()** and the resulting value is assigned to the variable **constant**. Parameters **input** and **output** are handled similarly.

Note also that the variable constant is declared as **static**. This serves to hide the variable from other source files.

The actual node function **fafn_sample** generates the output whose value is the **constant** times its input, which is located at the buffer location of z-coord = **input**, and y-coord and x-coord is same as the node's. This output value is assigned to the buffer location [output, idy, idx].

Note that **Buffer()** is a macro. ***There is no index boundry checking done.*** It is the responsibility of the user to make sure that index values are within the Buffer size.

Appendix VI: SFINX Macros and Routines

This section provides a list of global variables, macros and utility functions needed for developing node function routines. Macros are used to hide much of the messy details of implementations. They can be used both at the left and right hand side of an assignment statement. However, some values, such as the number of links in a node N_NumL(), should not be modified as they will create unpredictable results.

VI.1 Global Variables

`Max_Z`
`Max_Y`
`Max_X`

These variables contain the current size of the buffer. Their values should not be modified.

VI.2 Macros

VI.2.1 Modifiable macros

The contents referred to by the following macros may be modified. Namely, these macros may appear on the left side of an assignment statement.

`Buffer(z, y, x)`

z, y, and x should be arithematic expressions. This provides the buffer entry of coordinate value (z, y, x). Note that through the sfinx simulator, the order of coordinates used is (z, y, x) and NOT (x, y, z).

`N_IthReg(node, rg)`

node is a NODE_TP *, and r should be an arithematic expression whose value is less than N_NumR(node). This macro gives the rg*th* register of this node. The registers are number starting from 0 up to (N_NumR(node) - 1).

`N_IthLReg(node, ln, rg)`

node is a NODE_TP *, ln and rg should be arithemtic expressions whose values are less than N_NumL(node) and N_NumLR(node), respectively. This macro gives the rg*th* register of ln*th* link of the node.

VI.2.2 Unmodifiable macros

The contents referred to by the following macros must not be modified. They must not be used on the left side of an assignment statement. If their values are modified, the behavior of the simulator is undefined.

`N_Id(node)`
node is a NODE_TP *. This macro gives the node id, which is defined as a 3 element array of short. Thus N_Id(node)[0] is the z coordinate, N_Id(node)[1] the y coordinate, and N_Id(node)[2] the x coordinate.

`N_NumL(node)`
node is a NODE_TP *. This macro gives the number of links in this node.

`N_NumR(node)`
node is a NODE_TP *. This macro gives the number of node registers in this node.

`N_NumLR(node, ln)`
node is a NODE_TP *, ln should be an arithematic expression whose values are less than N_NumL(node).. This macro gives the number of registers for the ln*th* link.

`N_IthSrcN(node, ln)`
node is a NODE_TP *, ln should be arithemtic expression whose value is less than N_NumL(node), respectively. This macro gives the id of the node associated with the ln*th* link.

VI.3 SFINX Utility Functions

The following are a set of utility functions. They are often helpful in writing node functions.

```
void  fpr_id(fp, id)
      FILE          *fp;
      NODE_DI_TP    id;

#define pr_id(id) fpr_id(stdout, id)
```

The values returned by N_Id() or N_IthSrcN() macro are of type NODE_ID_TP. This function prints the id in standard notation to fp.

```
void  fpr_regs(fp, regs, numr)
      FILE        *fp;
      REG_TP      *regs;
      int         numr;

#define pr_regs(regs, numr)    fpr_regs(stdout, regs, numr)
```

The values returned by N_IthReg() or N_IthLReg() macros are of REG_TP. This function prints the numr register values beginning from the one pointed to by regs. This routine can handle any of the defined REG_TPs. The output goes to fp. To print out all node registers, use the statement:

```
pr_regs( & N_IthReg(node, 0), N_NumR(node))
```

To print out all registers of a link, use the statement:

```
pr_regs( & N_IthLReg(node, ln, 0), N_NumLR(node, ln))
```

To print a buffer entry, use the statement:

```
pr_regs( & Buffer(z, y, x), 1)
```

```
int   get_uvar_val(var_name, ret_val)
      char  *var_name;
      char  *ret_val;
```

This function returns the value associated with the user variable var_name. If the variable exists, then the function returns TRUE and the value is strcpy()'ed to ret_val, which must point to a character array with enough space. Otherwise, the function returns FALSE, and ret_val is untouched.

```
void  setu_val(var_name, var_val)
      char  *var_name;
      char  *var_val;
```

This function sets the value associated with the user variable var_name to var_val. If the variable did not pre-exists, then it is created.

```
int   get_var_val(var_name, ret_val)
      char  *var_name;
      int   *ret_val;
```

This function returns the value associated with the environment variable var_name. If the variable exists, then the function returns TRUE and the value is assigned to *ret_val, which must point to an integer variable. Otherwise, the function returns FALSE, and ret_val is untouched. The file list_set.c contain the list of environment variables.

Appendix VII: Sample Explicit Network (EN) Simulation

This section describes a SFINX simulation of a pedagogical network using Explicit Representation. The network is a modified Hartline_Ratliff [10] backward inhibitor model of size 1-by-4 (Figure 3). This model exhibits Mach band effects of lateral inhibition.

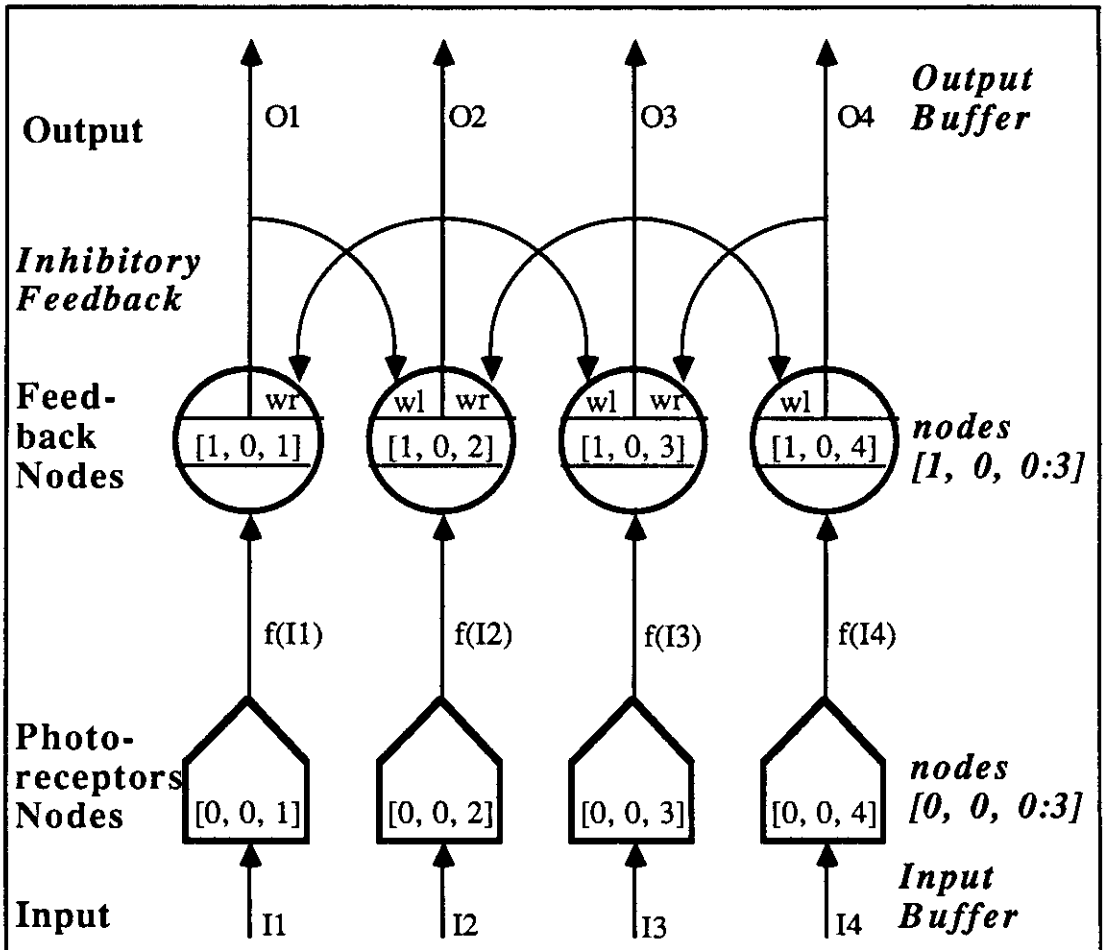


Figure 3. Modified Hartline-Ratliff backward inhibitor model

The input to this network is a 1-by-4 image intensities. This input image will be stored in buffer location [*input_layer*, 0, 0 : 3]. "input_layer" will be defined as a user variable.

There are 4 photoreceptor nodes, whose id's are in the range [0, 0, 0 : 3]. Each photoreceptor node receives a single input from a buffer location. The z component of this location is defined by the user variable "input_layer". The x and y coordinates are the values of third and second node id fields, respectively. For example, a photoreceptor node [, y, x] will receive its input from buffer location [input_layer, y, x].

The behavior of photoreceptor nodes can be described by the following algorithm:

if ($I_t > O_{t-1}$)
then
$O_t = I_t$
$O_0 = I_t$
$t = 0$
else
$O_t = O_0 e^{-\alpha t \Delta t}$
$t = t + 1$
endif

Where I_t is the current input value; O_{t-1} the previous output value; O_t the current output value being computed. O_0 is the last input that exceeded the output of the previous cycle. Δt is the inter-cycle duration and t the number of cycles since O_0. Therefore, (t x Δt) is the total amount of time since O_0. α determines the rate of decay. Δt and α will be set globally using user variables. For each photoreceptor node, O_0 and t will be stored in first and second general registers, respectively. The following C code implements the phtoreceptor node function.

```
#include <stdio.h>
#include "struct.h"
#include "extern.h"
#include <math.h>

static
double      alpha, delta_t;

static
int   input_layer;


/* ============================================================ */
void  ienfn_pr() /* ========================================== */
/* ============================================================ */
{
      char  buffer[128];

      alpha = get_uvar_val("alpha", buffer) ?
                  atof(buffer) :    /* returned value */
                  0.9;        /* default value  */

      delta_t = get_uvar_val("delta_t", buffer) ?
                  atof(buffer) :    /* returned value */
                  0.9;        /* default value  */

      input_layer = get_uvar_val("input_layer", buffer) ?
                  atoi(buffer) :    /* returned value */
```

```
                0;          /* default value  */

    if (get_uvar_val("pr_debug", buffer)) {
        printf("ienfn_pr: i_l(%d) a(%f) d_t(%f)\n",
                input_layer, alpha, delta_t);
    }
}


/* ============================================================ */
REG_TP      enfn_pr(node, POR, input) /* ====================== */
/* ============================================================ */
    NODE_TP     *node;            /* pointer to node structure
    */
    REG_TP      POR;        /* previous output value      */
    REG_TP      *input;     /* current input values       */
{
    REG_TP      intensity,  /* input intensity from buffer
    */
          beta;       /* initial value prior to decay     */
    int   num_cycles; /* of decay             */


    intensity = Buffer(input_layer,     /* z-coordinate   */
                N_Id(node)[1],          /* y-coordinate   */
                N_Id(node)[2]);         /* x-coordinate   */


                            /* check if new input exceeds */
                            /* previous output.           */
    if (intensity > POR) {

                            /* initial decaying value     */
          N_IthReg(node, 0) = intensity;

                            /* initialize decay clock     */
          N_IthReg(node, 1) = 0;

          return(intensity);
    }


                      /* retrieve initial decay value     */
    beta = N_IthReg(node, 0);

                      /* retrieve and increment decay     */
                      /* clock.                 */
    num_cycles = ++N_IthReg(node, 1);

    return (0.5 + beta*exp(-alpha*num_cycles*delta_t));
}
```

Actually, the above implementation of exponential decay may be streamlined in two ways. First, by computing the product of `alpha` and `delta_t` in the initializing function this multiplication operation can be done only once per simulation cycle. Second, the

exponential decay can be implemented by $O_t = O_{t-1} e^{-\alpha\Delta t}$. This method eliminates the need for maintaining O_0 and t separately. Furthermore, since O_{t-1} is available through the POR parameter and $e^{-\alpha\Delta t}$ is simply a constant, no general registers are needed. However, the original implementation is provided to explicitly demonstrate a usage of general registers.

There are 4 feedback nodes, whose id's are in the range [1, 0, 0 : 3]. The output of a feedback node is the input from the photoreceptor node minus the weighted sum of the input from its immediate neighbors. By representing the photoreceptor weight as 1 and the lateral weights as negative numbers, the function becomes simply the weighted sum of its inputs. The following C code implements the feedback node function.

```
#include <stdio.h>
#include "struct.h"
#include "extern.h"
#include <math.h>

                /* the following parameter is used to     */
                /* scale the weights                      */
static
int   weight_factor;


/* ============================================================ */
void  ienfn_fb() /* ========================================== */
/* ============================================================ */
{
      char  buffer[128];

      weight_factor = get_uvar_val("weight_factor", buffer) ?
                atoi(buffer) :    /* returned value */
                100;        /* default value  */


      if (get_uvar_val("bd_debug", buffer)) {
          printf("ienfn_bd: output_layer(%d) weight_factor(%d)\n",
                output_layer, weight_factor);
      }
}


/* ============================================================ */
REG_TP      enfn_fb(node, POR, input) /* ===================== */
/* ============================================================ */
      NODE_TP     *node;      /* pointer to node structure  */
      REG_TP      POR;        /* previous output value      */
      REG_TP      *input;     /* current input values       */
{
      int   lcnt;             /* counter for links          */
      int   numl;             /* number of links            */
      REG_TP      total = 0;  /* totaling weighted input    */


      numl = N_NumL(node);

                              /* accumulate weighted input  */
      for (lcnt = 0; lcnt < numl; lcnt++) {
```

```
            total += input[lcnt] * N_IthLReg(node, lcnt, 0);
        }

                    /* place the output value in the          */
                    /* appropriate buffer location            */
                    /* weight_factor/2 term is for rounding   */
        return ((total + weight_factor/2)/weight_factor);
}
```

The structure of the network is defined by the following SASS input file:

```
[0, 0, 0]   0
            2       0       0
            0

[0, 0, 1]   0
            2       0       0
            0

[0, 0, 2]   0
            2       0       0
            0

[0, 0, 3]   0
            2       0       0
            0

[1, 0, 0]   0
            0
            2       [0, 0, 0]   1       0
                    [1, 0, 1]   1       0

[1, 0, 1]   0
            0
            3       [0, 0, 1]   1       0
                    [1, 0, 0]   1       0
                    [1, 0, 2]   1       0

[1, 0, 2]   0
            0
            3       [0, 0, 2]   1       0
                    [1, 0, 1]   1       0
                    [1, 0, 3]   1       0

[1, 0, 3]   0
            0
            2       [0, 0, 3]   1       0
                    [1, 0, 2]   1       0
```

For small networks, such as the one above, the SASS files may be created manually on a wordprocessor. However, this process is usually too time consuming and mistake prone for networks with many nodes. In such cases, it is often invaluable to develop a utility program to generate the SASS files. For example, the **gen_hr** program below produces arbitrary length network structures of the modified Hartline-Ratliff model described above. The length of the network is specified by the command line argument. The above SASS file was produced by the gen_hr program. Normally, the output of the utility program can be piped directly into the SASS program, thereby eliminating the creation of the intermediary textfile. For example the unix command **`gen_hr 4 | sass hr.s`** will create a file called "hr.s" containing the explicit representation of the modified Hartline-Ratliff model of length 4.

```
main(argc, argv)
      int   argc;
      char  **argv;
{
      int   i;
      int   numn = atoi(argv[1]);

      for (i = 0; i < numn; i++) {
            printf("\[0, 0, %d\] 0 2 0 0\n", i);
            printf("              0\n");
            printf("\n");
      }

      for (i = 0; i < numn; i++) {
            printf("\[1, 0, %d\] 0 0\n", i);

            printf("     %d  \[0, 0, %d\] 1 0\n",
                  ((i-1<0) || (i+1>=numn)) ? 2 : 3, i);

            if ((i-1) >= 0)
                  printf("        \[1, 0, %d\] 1 0\n", i-1);

            if ((i+1) < numn)
                  printf("        \[1, 0, %d\] 1 0\n", i+1);
            printf("\n");
      }
}
```

Once the network representation is made and the node functions have been linked into the simulator, the network can be loaded into the simulator and executed. The following SFINX script file contains the basic series of commands used to simulate the Hartline-Ratliff network. Note that the script file has been written to work with networks of arbitrary size. This example assumes that the graphics commands **clearw** and **draw** are available. If graphics modules are not available, then these commands must be commented out. Nevertheless, the results may be displayed textually with the **en** command.

```
load hr.s

exec load_hr_image            ; load image into the input_layer
```

```
setu input_layer "0"            ; set user variables for
setu alpha "10"                 ; photoreceptor nodes
setu delta_t ".001"

setu output_layer "0"           ; set user variables for
setu weight_factor "100"        ; feedback nodes
setu

en [0, , ] FR photoreceptor     ; set function pointers of nodes
en [1, , ] FR feedback


en [0, , ] GR 0 0               ; initialize photoreceptor registers
en [0, , ] GR 1 0               ; for exponential decay

en [1, , ] LR 0 0 100           ; set the weights for feedback links
en [1, , ] LR 1 0 -20
en [1, , ] LR 2 0 -20


clearw                          ; clear the graphics display
set pixsize 8                   ; set drawing scale

set w_starty 100                ; draw input image
set w_startx 100
draw 0

set w_starty 140                ; set the window position
runen 1                         ; simulate one cycle and
cpbuf                           ; display result
draw 1

set w_starty 160                ; continue simulation and display
runen 1
cpbuf
set w_starty 180
exec one_cycle

set w_starty 200
exec one_cycle
```

This file also refers to a script files "load_hr_image". This file will be dependent on the size of the network. This file is of the form:

```
buf [0, , :1] 96                ; enter input intensity into
buf [0, , 2:] 160               ; input area of the buffer
```

Finally, to run the simulation, invoke SFINX and simply issue the command exec **hr.sfinx**.

Appendix VIII: Sample Function Array (FA) Simulation

This section describes a SFINX simulation of a pedagogical network using Function Arrays. The network is a simple feedforward network which performs a 2-dimensional convolution on an image.

The code below is a general convolution node function, where the size of the convolution mask may be dynamically set. As usual, the initializing function sets the global parameters, such as input and output buffer layers. In this implementation, when the user variable "convolve_wfile" is set, its value is used to open an external file which contains the description of the rectangular convolution mask. The first two integers of this file defines the y and x length of the mask. Following these integers, (y length) times (x length) floating point numbers are read in as the weights in row-major (or y-major) order.

If the user variable "convolve_wfile" is not set, then the built-in default mask is used. Note that the dimensions of the mask size must be odd in order to insure that the center of the kernel is well defined.

```
#include <stdio.h>
#include "struct.h"
#include "extern.h"
#include <math.h>

#define MAX_M (64)              /* define maximum mask size    */
static
int   input_layer,              /* input output layers         */
      output_layer,
      bug_flag,                 /* flag to indicate error      */
      ylen,                     /* for size of the mask        */
      xlen;

static
double      mask[MAX_M][MAX_M];      /* mask matrix            */

                                    /* default mask           */
static
int   def_ylen = 3,
      def_xlen = 3;
static
double      def_mask[3][3] = {
                  -1.0, -1.0, -1.0,
                  -1.0,  8.0, -1.0,
                  -1.0, -1.0, -1.0
      };


/* ========================================================== */
void  ifafn_conv() /* ======================================== */
/* ========================================================== */
{
      int   i, j;
```

```
    char  buffer[128];
    char  fname[128];

                              /* define input layer          */
    input_layer = get_uvar_val("convolve_input", buffer) ?
                atoi(buffer) : 0;

                              /* define output layer         */
    output_layer = get_uvar_val("convolve_output", buffer) ?
                atoi(buffer) : 1;

    bug_flag = FALSE;

                              /* see if weights are defined */
                              /* in an external file.       */
    if (!get_uvar_val("convolve_wfile", fname)) {

                              /* use default mask            */
          strcpy(fname, "");

          ylen = def_ylen;
          xlen = def_xlen;

          for (i = 0; i < ylen; i++)
                for (j = 0; j < xlen; j++)
                      mask[i][j] = def_mask[i][j];


                              /* read weights from file      */
    } else if (!readweights(fname)) {
          printf("convolve: error in reading file (%s).\n",
                fname);
          bug_flag = TRUE;
    }


    if (!get_uvar_val("convolve_debug", buffer))
          return;


    printf("convolve: in(%d)  out(%d)  wfile(%s) bug(%s)\n",
                input_layer, output_layer, fname,
                bug_flag ? "ON" : "OFF");

    if (bug_flag)
          return;

    printf("MASK y(%d) x(%d):\n\n", ylen, xlen);
    for (i = 0; i < ylen; i++) {
          for (j = 0; j < xlen; j++) {
                printf("%10.3f ", mask[i][j]);
          }
          printf("\n");
    }
    printf("\n");
}
```

```
/* ============================================================ */
static
int   readweights(fname) /* ================================== */
/* ============================================================ */
      char  *fname;
{
      int   i, j;
      FILE  *fp = fopen(fname, "r");

      if (!fp) {
            printf("convolve: cannot open file (%s).\n",
                  fname);
            return FALSE;
      }
                              /* read the size of the mask  */
      fscanf(fp, "%d %d", &ylen, &xlen);

      if ((ylen >= MAX_M) || (xlen >= MAX_M)) {
            printf("convolve: ylen(%d) and xlen(%d) %s %d.\n",
                  ylen, xlen,
                  "must be less than", MAX_M);
            return FALSE;
      }

      if ((ylen < 1) || (xlen < 1)) {
            printf("convolve: ylen(%d) and xlen(%d) %s.\n",
                  ylen, xlen,
                  "must be greater than zero");
            return FALSE;
      }

      if (!(ylen % 2) || !(xlen % 2)) {
            printf("convolve: ylen(%d) and xlen(%d) %s.\n",
                  ylen, xlen,
                  "must be odd numbers");
            return FALSE;
      }
                              /* read the mask        */
      for (i = 0; i < ylen; i++) {
            for (j = 0; j < xlen; j++) {
                  if (fscanf(fp,"%lf", mask[i]+j) == EOF) {
                        printf("convolve: %s.\n",
                              "premature EOF");
                        return FALSE;
                  }
            }
      }
      return TRUE;
}


/* ============================================================ */
void  fafn_conv(idz, idy, idx) /* ============================ */
/* ============================================================ */
      int   idz, idy, idx;
{
      int   x, y;
      double      tot = 0.0;
```

```
        int    xwidth = xlen /2,
               ywidth = ylen /2;

                            /* if error occured, do nothing      */
        if (bug_flag)
               return;
                            /* check if node position is         */
                            /* out of boundry                    */
                            /* if yes, just return zero.         */
        if ((idx < xwidth) || ((Max_X-idx) <= xwidth) ||
            (idy < ywidth) || ((Max_Y-idy) <= ywidth)) {
              Buffer(output_layer, idy, idx) = 0;
              return;
        }
                            /* compute the weighted sum          */
        for (y = -ywidth; y <= ywidth; y++)
              for (x = -xwidth; x <= xwidth; x++)
                    tot += mask[y+ywidth][x+xwidth] *
                            Buffer(input_layer, idy+y, idx+x);

                            /* if REG_TP is int or short,        */
                            /* then return value should be       */
                            /* rounded.                          */
#ifdef      S_REG
      Buffer(output_layer, idy, idx) = tot + 0.5;
#else
#ifdef      I_REG
      Buffer(output_layer, idy, idx) = tot + 0.5;
#else
      Buffer(output_layer, idy, idx) = tot;
#endif
#endif
}
```

Note that the node function checks for the boundry condition in terms of the image location. The simulation may run significantly faster if this portion of the code is eliminated. To do so, the user must ensure that this function is not called with an illegal **idy** and **idx** parameter. This can be acheived by properly setting the environment variables `fa_x0`, `set fa_y0`, `set fa_xx` and `set fa_yy` when using **runlfa** command. When using **runfa**, the buffer layer used for node function must be set to 0 for illegal boundry positions.

Once the above function is linked into the SFINX simulator, then the following script file can be used to simulate the convolution network. This script file uses graphics commands; therefore, those systems without the graphics module must first comment out the graphics commands.

```
setu convolve_input "0"        ; define input layer
setu convolve_output "1"       ; define output layer
setu convolve_wfile "mask"     ; define weight file
setu convolve_debug            ; turn debug mode ON
setu
```

```
buf[0, :15, :15]  64          ; enter input image
buf[0, :15, 16:]  96
buf[0, 16:, 16:]  128
buf[0, 16:, :15]  160

clearw                        ; clear display window
set pixsize 4                 ; set drawing size
set w_starty 200              ; position display window
set w_startx 100
draw 0                        ; draw the input image


set fa_x0 0                   ; define node tesselation
set fa_y0 0
set fa_xx 30
set fa_yy 30
set fa_incx 2
set fa_incy 2

run1fa convolve               ; simulate network

set minp_val -256             ; adjust drawing parameter
set maxp_val  256
set w_startx 400              ; reposition window
draw 1                        ; draw result
```

The above script file assumes that the buffer size and hence the input image size is 32 by 32. This can be done by invoking the SFINX simulator with the parameters **y=32** and **x=32**. The command **run1fa** is used to create a ficticious layer of nodes. The tesselation of the network can be easily modified by adjust the 6 environment variables associated with the **run1fa** command.

Acknowledgements

We are grateful to all the members of MPL for their helpful suggestions and continued encouragements, often manifested in the form of greater and greater demands, during the development of SFINX. We are specially indebted to David Gungner for his numerous help, among them laying the foundation of SFINX in PUNNS and giving birth to the idea of Function Arrays and the name SFINX. We would also like to thank Edmond Mesrobian, Paul Lin and Inge Heisey for offering themselves as guinea pigs during the early stages of this experiment and Michael Stiber, Douglas Trainor and David Lee for their critical reviews of the manuscript.

References

[1] Kuffler, S. W., and Nichols, J. G., *From Neuron to Brain*, Sinauer Associates, Massachusetts, 1976.

[2] Perkel, D. H., A Computer Program for Simulating a Network of Interacting Neurons, *Computers and Biomedical Research*, Vol. 9, pp. 31 - 43, 1976.

[3] Wittie, L. D., Large-scale Simulation of Brain Cortices, *Simulation*, Vol. 31, Num. 3, September, 1978.

[4] Small, S. L., Shastri, L., Brucks, M. L., Kaufman, S. G., Cottrell, G. W., Addanki, S., ISCON: A Network Construction Aid and Simulation for Connectionist Models, *University of Rochester, Department of computer Science Technical Report*, TR 109, April, 1983.

[5] Fanty, M., Goddard, N., User's Manual Rochester Connection Simulator - Draft, *University of Rochester, Department of Computer Science Technical Report*, September, 1986.

[6] Gungner, D. and Skrzypek, J., UCLA PUNNS - A Neural Network Machine for Computer Vision, *SPIE Conference on Electro-Optic Systems and Devices*, SPIE Proceedings Vol. 755, January 1987.

[7] Mesrobian, E., Skrzypek, J., Discrimination of Natural Textures: A Neural Network Architecture, *Proceedings of the IEEE First Annual International Conference on Neural Networks*, June, 1987.

[8] Heisey, I., Skrzypek, J., Color Constancy in Early Vision: A Connectionist Model, *Proceedings of the IEEE First Annual International Conference on Neural Networks*, June, 1987.

[9] Oyster, J. M., Skrzypek, J., Computing Shape with Neural Networks: A Proposal, *Proceedings of the IEEE First Annual International Conference on Neural Networks*, June, 1987.

[10] Hartline, H. R., Ratliff, F., Spatial Summation of Inhibitory Influences in the Eye of the Limulus, *Science*, Vol 120, no. 3124, 1954, p.781