Abstract

Specifying, constructing and simulating structured connectionist networks requires significant programming effort. System tools can greatly reduce the effort required, and by providing a conceptual structure within which to work, make large and complex network simulations possible. The Rochester Connectionist Simulator is a system tool designed to aid specification, construction and simulation of connectionist networks. This report describes this tool in detail: the facilities provided and how to use them, as well as details of the implementation. Through this we hope not only to make designing and verifying connectionist networks easier, but also to encourage the development and refinement of connectionist research tools themselves.
## Contents

1 Introduction

1.1 Overview ........................................... 1
1.2 How to use this report ............................... 2
1.3 Version 4.2 changes .................................. 3
1.4 Version 4.1 changes .................................. 3
1.4.1 Simulator enhancements .............................. 4
1.4.2 Graphics Interface enhancements ................. 5
1.4.3 Back propagation enhancements .................... 6
1.5 Installation ............................................. 6
1.6 Obtaining the simulator software ..................... 6
1.7 Acknowledgments ...................................... 7

2 Simulator User Manual

2.1 Introduction ............................................ 9
2.1.1 Preliminaries ........................................ 11
2.1.2 Graphics Interface ................................ 11
2.2 Network Construction ................................ 11
2.2.1 Creating space for units ........................... 11
2.2.2 Making units ........................................ 12
2.2.3 Adding sites ........................................ 13
2.2.4 Making links ........................................ 13
2.2.5 Naming units ........................................ 14
2.2.6 State names ......................................... 14
2.2.7 A simple example .................................. 15
2.2.8 Flags ................................................ 15
2.2.9 Sets ................................................. 16
2.2.10 Activation Functions ............................... 17
2.2.11 Library functions .................................. 19
2.2.12 Another simple example ......................... 19
2.2.13 Modular construction ............................. 21

2.3 Making an executable Simulator ...................... 21

2.4 Simulation ........................................... 22
  2.4.1 Help ............................................... 22
  2.4.2 Building the network ............................. 23
  2.4.3 Debugging during network construction .......... 23
  2.4.4 Calling functions ............................... 26
  2.4.5 Unit specification in commands .................. 27
  2.4.6 Simulation commands ............................. 27
  2.4.7 Examination commands ............................ 30
  2.4.8 Modification commands ........................... 33
  2.4.9 Set commands .................................... 39
  2.4.10 File commands ................................... 40
  2.4.11 Compilation and Code-loading commands ......... 44
  2.4.12 Memory Management commands .................... 49
  2.4.13 Miscellaneous commands .......................... 50
  2.4.14 Abbreviated commands ........................... 52
  2.4.15 Constructing a network from the command interface 53

2.5 Simulating propagation delays ....................... 53
  2.5.1 Installing the delay simulator ................... 53
  2.5.2 Using the delay simulator ........................ 54
  2.5.3 Delay command ................................... 54
  2.5.4 MakeDelayLink primitive ........................ 54
  2.5.5 MakeDelayUnit primitive ........................ 55
  2.5.6 SetDelay and GetDelay primitives ................. 55
  2.5.7 ChangeDelayBuffer primitive ...................... 56

2.6 Advanced Programming Features ..................... 56
  2.6.1 Simulator Variables ................................ 56
  2.6.2 Network Access Functions ........................ 56
2.6.3 Saving and reloading user data structures ........................................ 59
2.6.4 Customizing unit, site and link data structures ................................. 59
2.6.5 Customizing the simulator command interface .................................. 60
2.6.6 Modeling Time .................................................................................. 60
2.7 An Extended Example of network construction ..................................... 60
  2.7.1 Designing a network ........................................................................ 60
  2.7.2 The Problem: four coloring a map .................................................... 60
  2.7.3 Designing the build program ............................................................ 61
  2.7.4 Implementing in C ........................................................................... 62
  2.7.5 A command script to demonstrate the network ............................... 68
2.8 Floating Point version ........................................................................... 70

3 Advanced Programming Manual ................................................................ 73
  3.1 Introduction ......................................................................................... 73
  3.2 The Network Data Structure ............................................................... 73
    3.2.1 Creating space for units ............................................................... 73
    3.2.2 Making units ............................................................................... 75
    3.2.3 Adding sites ................................................................................. 75
    3.2.4 Making links ............................................................................... 76
    3.2.5 Deleting sites and links ............................................................... 77
    3.2.6 Deleting the network ................................................................. 78
  3.3 Naming ............................................................................................... 78
    3.3.1 Unit names ................................................................................... 78
    3.3.2 Function names ........................................................................... 79
    3.3.3 Set and State names ..................................................................... 80
    3.3.4 Miscellaneous name functions .................................................... 80
  3.4 Display functions ................................................................................ 80
  3.5 Set functions and macros ..................................................................... 82
    3.5.1 Adding units to a set .................................................................... 82
    3.5.2 Removing units from a set ........................................................... 83
    3.5.3 Creating and deleting sets ............................................................ 83
    3.5.4 Set theoretic functions .................................................................. 84
    3.5.5 Set membership tests ................................................................... 84
    3.5.6 Miscellaneous set functions ....................................................... 85
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.6 Modifying and Accessing network values</td>
<td>85</td>
</tr>
<tr>
<td>3.7 Library functions</td>
<td>86</td>
</tr>
<tr>
<td>3.7.1 Unit functions</td>
<td>86</td>
</tr>
<tr>
<td>3.7.2 Site functions</td>
<td>87</td>
</tr>
<tr>
<td>3.7.3 Link functions</td>
<td>87</td>
</tr>
<tr>
<td>3.7.4 Weight scaling</td>
<td>87</td>
</tr>
<tr>
<td>3.8 Unit index and pointer macros</td>
<td>88</td>
</tr>
<tr>
<td>3.9 Flag Macros</td>
<td>88</td>
</tr>
<tr>
<td>3.9.1 Setting a flag</td>
<td>88</td>
</tr>
<tr>
<td>3.9.2 Clearing a flag</td>
<td>89</td>
</tr>
<tr>
<td>3.9.3 Testing a flag</td>
<td>89</td>
</tr>
<tr>
<td>3.9.4 An example</td>
<td>89</td>
</tr>
<tr>
<td>3.10 Simulating</td>
<td>90</td>
</tr>
<tr>
<td>3.11 File functions</td>
<td>90</td>
</tr>
<tr>
<td>3.11.1 Network saving functions</td>
<td>90</td>
</tr>
<tr>
<td>3.11.2 Network checkpointing functions</td>
<td>91</td>
</tr>
<tr>
<td>3.11.3 Logging functions</td>
<td>92</td>
</tr>
<tr>
<td>3.11.4 Other file functions</td>
<td>93</td>
</tr>
<tr>
<td>3.12 Calling the Command Interface</td>
<td>94</td>
</tr>
<tr>
<td>3.12.1 Calling the debug interface</td>
<td>94</td>
</tr>
<tr>
<td>3.12.2 Running another interface on top of the standard one</td>
<td>94</td>
</tr>
<tr>
<td>3.13 Parsing command lines</td>
<td>95</td>
</tr>
<tr>
<td>3.13.1 Lexical functions</td>
<td>95</td>
</tr>
<tr>
<td>3.13.2 Parsing unit specifications</td>
<td>97</td>
</tr>
<tr>
<td>3.13.3 An example of a simulator command function</td>
<td>98</td>
</tr>
<tr>
<td>3.14 Guarded code functions</td>
<td>98</td>
</tr>
<tr>
<td>3.15 Miscellaneous functions</td>
<td>98</td>
</tr>
<tr>
<td>3.16 Simulator Variables</td>
<td>100</td>
</tr>
<tr>
<td>3.17 The Name Table</td>
<td>102</td>
</tr>
<tr>
<td>3.18 Customizing unit, site and link data structures</td>
<td>103</td>
</tr>
<tr>
<td>3.18.1 Redefining the data type</td>
<td>104</td>
</tr>
<tr>
<td>3.18.2 Making a link</td>
<td>104</td>
</tr>
<tr>
<td>3.18.3 The link function</td>
<td>105</td>
</tr>
</tbody>
</table>
CONTENTS

3.18.4 The site function ........................................ 106
3.18.5 Linking with simulator code .............................. 106
3.18.6 Displaying, saving, loading, etc. ....................... 106
3.18.7 Displaying units ........................................ 108
3.18.8 Listing links .......................................... 108
3.18.9 Checkpointing and Restoring ......................... 109
3.18.10 Saving and Loading .................................. 110
3.18.11 Unit and Site functions ............................... 111
3.18.12 Completing the example ............................... 111
3.19 Customizing the simulator command interface .......... 112
3.20 Adding to the Help information .......................... 114
3.21 Avoiding name clashes .................................. 115
3.22 Floating Point version .................................. 115
3.23 Linking user and simulator code ......................... 115
3.24 The simulator as a subroutine ........................... 116
   3.24.1 Controlling the simulator from another program .... 116
   3.24.2 Return command .................................... 117
   3.24.3 Integration with MIT Scheme and Kyoto Common Lisp 117
3.25 Implementation Details ................................ 118
   3.25.1 Main Data Structure and Execution Cycle ........ 118
   3.25.2 NameTable ......................................... 121
   3.25.3 Command Interface ................................ 122
   3.25.4 Debugging package ................................ 122
   3.25.5 Dynamic Code Loading and the Item Table .......... 123
   3.25.6 Simulating Propagation Delay ....................... 127
   3.25.7 Memory Management ................................ 127
   3.25.8 User defined network extensions .................. 129
   3.25.9 The shell script *makesim* ........................ 130
   3.25.10 Control flow with other packages ............... 130
   3.25.11 Saving, Loading, Checkpointing and Restoring .... 130
CONTENTS

4.14.1 Structural Organization ........................................... 167
4.14.2 External Variables and Functions ................................. 168
4.14.3 Data Structures ..................................................... 168
4.14.4 Processing Strategy .................................................. 169
4.14.5 Link Mode processing ............................................... 171
4.14.6 Floating Point version ............................................. 171
4.15 XGI: The X11 Interface ................................................. 172
  4.15.1 Control Window .................................................... 172
  4.15.2 Display Windows ................................................... 174
  4.15.3 Layout Window ...................................................... 175
  4.15.4 Custom Menus ....................................................... 177
  4.15.5 Examples ............................................................ 178
  4.15.6 Interacting with XGI from Program Code ....................... 179
  4.15.7 Integrating User X Code with XGI ............................... 189
  4.15.8 Implementation ..................................................... 189
  4.15.9 Known Bugs ........................................................ 191
  4.15.10 Deficiencies ...................................................... 192
5 Back Propagation Package ................................................. 193
  5.1 Introduction ........................................................... 193
  5.2 Building a back-propagation module ................................ 194
    5.2.1 BPmodule ......................................................... 195
    5.2.2 BPinput .......................................................... 195
    5.2.3 BPhidden ......................................................... 195
    5.2.4 BPoutput ........................................................ 196
    5.2.5 BPteach .......................................................... 196
    5.2.6 BPfire ............................................................. 197
    5.2.7 BPlink ............................................................ 197
    5.2.8 BPendmod ......................................................... 197
  5.3 Simulating the module ................................................. 198
    5.3.1 BPsetinput ....................................................... 198
    5.3.2 BPsetteach ........................................................ 198
    5.3.3 BPCycle ........................................................... 199
  5.4 Example: learning 8-3-8 encoding ................................... 199
5.4.1 Constructing the network ............................................. 199
5.4.2 Running a simulation ................................................ 201
5.5 Activation and error-propagation functions ...................... 202
  5.5.1 Writing error propagation functions ......................... 203
5.6 Module unit layout and operation ................................... 205
List of Tables

2.1 User accessible simulator variables ........................................ 57
2.2 Example code for a build function ........................................ 64
2.3 Example code for making a region .......................................... 66
2.4 Example code for makecolor and mux functions ......................... 67
2.5 Example code for making borders .......................................... 68
2.6 Example code for unit function ........................................... 69

3.1 Simulator network data structure ........................................... 74
3.2 Code for the disp command .................................................. 99
3.3 Example command interface code ......................................... 113

4.1 Parameter Scrollbar Widget code ......................................... 190
4.2 Creating a Scrollbar Control for a Variable ............................... 191

5.1 Back propagation build code ............................................... 200
5.2 Code for running a back propagation simulation ....................... 202
5.3 Code for back propagation error function ................................ 204
List of Figures

2.1 Units, sites and links ............................................ 10
2.2 Linked blocks of links ........................................... 49

3.1 Main Data Structure ............................................. 119
3.2 Dynamic Code Loading .......................................... 124
3.3 Item Table with Chain .......................................... 126
3.4 Simulating Propagation Delay .................................. 128

4.1 Initial GI tool window .......................................... 135
4.2 GI tool window: Main mode .................................... 140
4.3 GI tool window: Link mode ..................................... 144
4.4 GI tool window: Text mode ..................................... 146
4.5 GI tool window: Draw mode ..................................... 148
4.6 GI tool window: Custom mode .................................. 150
4.7 XGI Control Window ............................................ 173
4.8 XGI Display Window ............................................. 174
4.9 XGI Information Window ........................................ 175
4.10 XGI Layout Window: Main mode ............................... 176
4.11 XGI Layout Window: Text mode ............................... 176
4.12 XGI Custom Menu .............................................. 177
4.13 XGI Custom Menu Editing ..................................... 178
4.14 Structured Display and Object Oriented Graphics ............. 179
Chapter 1

Introduction

1.1 Overview

This report describes the features and operation of the Rochester Connectionist Simulator (RCS) version 4.2, which is a flexible and powerful tool for simulating networks of highly interconnected information processing units. The computational paradigm it supports is known by a number of different names: "Parallel Distributed Processing", "Neural Networks", "Neural Computers", "Massively Parallel Computation" and others. Here at the University of Rochester we tend to use the term "connectionist" to describe this class of computational model; hence the name we've given to this tool.

RCS is designed to operate within the Unix environment and is written in the C programming language. It consists of a number of pieces and options which can be combined in various ways that will hopefully suit your needs. 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. It is simply too early to predict with accuracy what the future will decide are the most useful approaches. 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. Much of this flexibility results from making it easy — and sometimes necessary — for you, the user, to integrate your own functions and data structures into the simulator code. Although you can use RCS without writing C code, this will severely limit what you will be able to do. On the other hand, the ability to do a little C programming will give you great power to make RCS do your bidding.

Although RCS is a fairly well integrated package today, its evolution has taken a number of years and several different turns. The original implementation was in LISP. Gradually it became more C-based (mainly for performance reasons), but for a while one was forced to specify the network topology with LISP functions while the actual unit functions had to be coded in C. However in the spring of 1987 a major effort was completed to rewrite and enhance the simulator and it was dubbed version 4.0. It contained a considerable amount of
new function as well as a graphics interface for Sun Microsystems workstations. The winter
of 1988 saw version 4.1 make its appearance with still more enhancements (see Section 1.4)
as well as some fixes to version 4.0. Version 4.2 was released in October, 1989.

1.2 How to use this report

This document consists of what was originally a set of separate manuals that sprang up over
the period that RCS was being developed. Each one of these manuals makes up a separate
chapter of this report. Some information is repeated within several chapters because of this.
However the documentation is fairly complete.

In fact the documentation might be a little too complete in that you may be put off by
the sheer amount of it. However do not be intimidated by the vast number of functions and
options: any one person will probably ever use only a small fraction of these, and it would
be a mistake for you to try and read and absorb them all before attempting to build your
first network. If you are a new user you can probably get by quite nicely for now just by
reading the first several sections of Chapter 2. If you have a Sun Microsystems workstation,
or a X11 workstation or terminal, you might also want to read the some of Chapter 4 as
well. Once you are comfortable with the "basic" simulator functions, you can gradually
increase the scope of your familiarity with the others as your need arises.

Following is a list of the other chapters in this report and a brief description of what
they contain:

- **Chapter 2** is the User's Guide to the simulator itself. It will tell you how to build,
  run, modify, change and delete basic connectionist networks. It may be the only
  chapter you ever need depending on well the networks and functions already built
  into RCS suits your needs.

- **Chapter 3** contains advanced material for programming the simulator. It will be
  necessary for you to consult this if you plan to extensively control and monitor your
  network through your own C code, or if you wish to modify the simulator or its
  data structures, or if you wish to interface the simulator to other programs or other
  languages.

- **Chapter 4** is the User's Guide to the Graphics Interface to the simulator for running
  on Sun Microsystems workstations, or any other workstation or terminal capable
  of running the X window system. If you have access to one of these you'll almost
certainly want to check this out. It allows you to visually display the structure and
  activity of your network as it runs. It is almost a necessity for the debugging of
  large or complex networks — and it allows you to visually document your work for
  publication purposes.

- **Chapter 5** shows how to integrate and use a package specially designed to create and
  run back propagation networks. If these are the types of networks you want to build,
1.3 Version 4.2 changes

Version 4.2 contains a number of bug fixes and improvements.

- The most noticeable addition is the X11 Graphics Interface.

- This version now works under the SunOS 4.0 operating system, on Sun-3s and Sun-4s. It probably works on Sun-2s. It is also known to work on Vaxes (running BSD 4.3), and on the DECstation 3100 (running Ultrix). It may work on other MIPS-chipset machines with little effort. It does not work, as-is, on the Sun 386i, which uses a different object file format. It can probably be made to work on any Unix machine, if you try hard enough. The biggest impediment is the object file format of your machine, if it’s different from Berkeley a.out format.

- Various minor bug fixes have been made (e.g. to the back-propagation package). Unfortunately, the bug which prevents the saving and restoring of back-propagation networks is still present. That bug has been made less fatal, however.

- An Apple Macintosh port of the version 4.1 simulator has been contributed, and is included (as-is, untested) in the 4.2 distribution. It includes the Linda-C source code for a parallel version of the simulator. The code is in MPW C, so if you don’t have the MPW package you cannot use it. This Macintosh port was demonstrated at the IJCNN '89 Washington conference.

- The Simulator is now distributed under the terms of the Free Software Foundation General Public License, which, in a nutshell, can be summarized as follows: The software is copyrighted by the University of Rochester, and we offer a license which gives you legal permission to copy, distribute, and/or modify the software, subject to certain conditions. This free software is distributed in the hope that it will be useful, but without any warranty; without even the implied warranty of merchantability or fitness for a particular purpose. The precise terms of the license are in the file COPYING in the simulator distribution. If you cannot find a copy of the license, write to the Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

1.4 Version 4.1 changes

The documentation for all the enhancements added for version 4.1 has been seamlessly integrated into this report. However, for those of you who are already familiar with version 4.0, the next few sections contain a sketch of what these enhancements were and where to
look elsewhere in this report for the details. If your favorite does not appear in the list, all we can say is “maybe next time”.

In addition, a number of bugs in version 4.0 have been fixed. In particular saving, loading, checkpointing and restoring floating point networks should now work. Quitting the debug interface in the floating point version should work. Makesim should not get confused about whether you compiled for floating point or integer any more. It is now possible to turn debugging off from the debug interface; the error that put you in the debug interface must still be fixed to continue, but the simulator will ignore (or crash) further errors. The back propagation library now correctly allows the building of single layer back propagation networks. Also, a string copy bug in the back propagation construction library has been fixed. In the Graphics Interface, a problem with loading in user-defined icons has been fixed as well as the annoyance of having multiple “phantom” markers at a time appearing on the display.

1.4.1 Simulator enhancements

The following commands and functions have been added to the simulator itself:

- A set of commands and functions for deleting links and sites from an existing network are now available. See Sections 2.4.8.4, 2.4.8.6 and 3.2.5.

- A command and function to clean out the network data structures and restart the simulator from scratch has been implemented. The details can be found in Sections 2.4.8.16 and 3.2.6.

- There are three new commands to manipulate the unit, site and link functions. Sections 2.4.8.12, 2.4.8.13 and 2.4.8.14 contain the details.

- There are now commands to facilitate the dynamic compilation and loading of code into an existing network. Refer to Section 2.4.11 for more information.

- Some commands and functions for recovering free space within the simulator have been implemented. See Section 2.4.12 for details.

- A facility that allows the simulation of propagation delays along links (as real neural systems do) is now available. The specifics can be found in Section 2.5.

- It is now possible to call the simulator as a subroutine from another program. Included is support for doing this from Kyoto Common Lisp and MIT’s Scheme. Refer to Section 3.24.1 for more information.

- There is now a whatis command to report to what kind of object a particular name in the Name Table refers. See Section 2.4.13.4.
1.4. VERSION 4.1 CHANGES

• You no longer need to allocate space for your whole network ahead of time. In fact you don’t necessarily have to ever allocate space; the simulator will do it for you using a default size. See Section 2.4.8.1 for details.

• There is now an abort command that signals IOT. See Section 2.4.13.5.

• The -fpa switch is no longer given to the compiler when installing the simulator and compiling user files. You should use the environment variable FLOAT_OPTION instead; see Section 2.8 for more details.

1.4.2 Graphics Interface enhancements

The following changes and enhancements have been made to the Graphics Interface:

• A legitimate floating point version of the Graphics Interface exists. That is, output, potential, data and link weights fields are treated and displayed as true floating point numbers just as the floating point version of the simulator does. Section 2.8 describes how to create a floating point version of the simulator.

• In addition to lines and regular boxes, you can now draw “bounding boxes” which, when move, will also move anything inside of them as well as other overlapping bounding boxes. See Section 4.7.5 for a further description.

• A restart command has been implemented which allows the graphics interface (and, optionally, the simulator) to be “wiped clean” of all data structures so that you can begin building or displaying your network from scratch. Section 4.10.9 contains more details.

• Reading in of a command file through the Graphics Interface has been enhanced to allow for the inclusion of comments and blank lines. See Section 4.11.3 for further explanation.

• The marker that indicates the current coordinates where the next unit icons will be shown is not made visible unless the left mouse button is clicked on the display screen first. Thus unless you specifically want it displayed, the marker will not appear.

• The default action of the mouse buttons in custom mode is the same as they would be in main mode. See Section 4.7.6.

• The logging of commands to the Graphics Interface log file can now be controlled through a command. Section 4.10.8 contains the details.
1.4.3 Back propagation enhancements

The back propagation package has added one new parameter into each of the back prop­
agation unit activation function and the error propagation function, called, respectively
BPtemperature and BPlearn for parameterizing the rate of learning. See section 5.5 for
more details.

1.5 Installation

Once you have the tar tape, the installation of the simulator requires moving the directory
structure to your file system. Use the tar command explained in the cover letter that
you should have received with the tape. In the top level directory there is a file called
README which contains further instructions for preparing and tailoring the simulator
code to your needs.

1.6 Obtaining the simulator software

The software described in this document is available for free by “anonymous FTP” via the
Arpa Internet, from the host “cayuga.cs.rochester.edu”, in the directory “pub/simulator”.
(Don’t forget to use “type binary” to retrieve compressed files!) Official bug patches also
reside in that directory.

If you are unable to access the Simulator distribution via the Internet, the Simulator
is available on magnetic media. Send a check for US$150 (payable to the University of
Rochester) to

Department of Computer Science
University of Rochester
Rochester, NY 14627

and specify whether you want a 1600BPI 1/2" magnetic reel, or a QIC-24 1/4" (SUN)
cartridge. (The tape will be in Unix “tar” format in either case.) You will receive the tape
of your choice and one copy of this manual. The manual alone can be ordered for US$10
per manual.

There is a <simulator-users@cs.rochester.edu> mailing list. On this list you will find
the other users of the Rochester Connectionist Simulator. To join (or drop yourself from)
the mailing list, send a note to <simulator-request@cs.rochester.edu>. We urge all users
to join the list to keep up with the latest news and bug fixes.

Please send bug reports to <simulator-bugs@cs.rochester.edu>. We are interested in
fixing bugs, but can’t make any promises! Please make your bug reports as specific as
possible.
1.7 Acknowledgments

A number of people associated with the University of Rochester have had a hand in the development of RCS. The original simulator was written by Stephen Small, Lokendra Shastri, Gary Cottrell and others. Mark Fanty later significantly enhanced it, weaned it away from LISP and designed the heart of the simulator as it exists today. He also ported the simulator to the BBN Butterfly parallel processor. Nigel Goddard was responsible for the extensive simulator rewrite that became version 4.0 and also for the version 4.1 enhancements. Kenton Lynne conceived of and wrote the Graphics Interface as well as its version 4.1 enhancements. Toby Mintz wrote the back propagation package. Michael McInerny wrote the first version of the X11 graphics interface, which was subsequently beaten into shape by Mark Fanty and Nigel Goddard.

As far as credit for authorship of this report: Nigel Goddard authored Chapters 2 and 3, co-authored Chapter 5 with Toby Mintz, and wrote the X11 sections of Chapter 4. Kenton Lynne wrote Chapter 1 (this introduction) and most of Chapter 4. Rose Peet put together the original version of this integrated report (TR 233) from the scraps and pieces of documentation mentioned above. Liudvikas Bukys did another pass in 1989 to remove many of the spelling errors and typographical inconsistencies that had accumulated over the years.

Financial support for this work came from ONR/DARPA research contracts N00014-82-K-0193 and N00014-84-K-0655, and from the IBM Corporation, which supported Kenton Lynne at the University of Rochester through the IBM Resident Study program. Finally, much of the credit for this entire undertaking belongs to Dr. Jerome Feldman for his constant inspiration, encouragement and timely advice during all aspects of this project.
Chapter 2

Simulator User Manual

2.1 Introduction

Connectionist networks consist of simple computational elements (units) which communicate by sending their level of activation via links to other elements. The units have a small number of states, and compute simple functions of their inputs. Associated with each link is a weight, indicating the "significance" of activation arriving over that link. The behavior of the network is determined by the pattern of connections, the weights on the links, and the unit functions.

The Rochester Connectionist Simulator supports construction and simulation of a wide variety of networks. The main design criterion has been flexibility. Each unit can compute a different function, any amount of data may be associated with each unit, and an arbitrary connection pattern may be specified.

The particular network paradigm supported by the simulator is, in brief, as follows. Each unit has a number of sites at which the incoming links arrive. The provision of sites allows differential treatment of inputs, since the links themselves do not indicate their origin at the destination unit. Figure 2.1(a) shows a small network with the links arriving at sites.

Simulations may be run synchronously or asynchronously. During synchronous simulation all units use the output values computed during the previous step as their input. The order of simulation is unimportant, the network behaving as though all units update simultaneously. During asynchronous simulation, at each step a fraction of the units are updated, in pseudo-random order, with the new output value immediately transmitted to the other units. It is guaranteed that after a limiting number of steps every unit will have been updated at least once.

There are a number of example networks in the "/example directory. It would be a good idea to check these out before writing any code.
(a) Units with sites

(b) Unit vector with linked lists

Figure 2.1: Units, sites and links
2.1.1 Preliminaries

It will aid in understanding the following sections if the representation of units, sites and links is discussed here. The main data structure is an array of unit structures (of type Unit), with a linked list of site structures (of type Site) attached to each unit structure, and a linked list of incoming link structures (of type Link) attached to each site structure. Figure 2.1(b) illustrates these structures. Unit, site and link structures contain various pieces of data. Units have a potential, corresponding to the level of activation, a state which can be used to vary the unit function, and an output which is transmitted along the outgoing links. Each site has a value, which is set by the site function. Each link has a weight which may be modified by the link function, a pointer value to the incoming value, and the index from unit holding the index of the source unit for the link. Each unit, site and link structure contains a pointer to a function which is called by the simulator to simulate the action of the unit, site or link. In addition, each unit, site and link structure contains a data field which is for general use by the user. Further details may be found in the Advanced Programming Manual.

2.1.2 Graphics Interface

The simulator command interface (see Section 2.4) was designed for simple terminal operation. This document describes network construction and simulation on the terminal interface.

The Graphics Interface is a package that runs on top of the simulator described in this document, and is a powerful network debugging aid. See Chapter 4 for details. If the Graphics Interface is included when making a simulator (see Section 2.3), it is automatically invoked when the simulator is run. All the commands described in section 2.4 are available from the Graphics Interface, as well as many more.

2.2 Network Construction

Although it is possible to construct a network from the command interface (see Section 2.4.15), it is a time consuming process and really only suited to novice users. One of the example networks is constructed this way, and is described in section 2.4.15.

Generally a network is built in the simulator by a user program written in C. The simulator provides primitive functions, the major ones being to make units, add sites, make links, and associate a name with one or more units. Many other primitives are also provided to access parts of the network data structure.

2.2.1 Creating space for units

Before any units can be made, you should try to specify the total number of units in the network that will be needed. This is done through the AllocateUnits function:
AllocateUnits(number)
    int number;

If AllocateUnits has not been called before the first call to MakeUnit, then 1000 units are automatically allocated. If you try to construct more units than have been allocated, the simulator allocates a new and larger unit array, copies the already made units into the new array, and deletes the old array. The extra number of units allocated whenever the simulator runs out of allocated units is the same as the most recent call to AllocateUnits. You may call AllocateUnits as many times as necessary. Each call increments the number of units allocated by the argument to AllocateUnits. Thus if you initially call AllocateUnits(2000) you will get 2000 units. If later you call AllocateUnits(100) you will now have 2100 units allocated. If later on you try to make units past 2100, the simulator will increase the number of allocated units in increments of 100.

2.2.2 Making units

Now units may be made with a calls to MakeUnit. This function builds a new unit, using space allocated by AllocateUnits, for example:

    int MakeUnit(type, func, ipot, potential, data, output, istate, state)
    char *type;
    func_ptr func;
    int ipot, potential, data, output, istate, state;

*type* is a pointer to a character string, and is simply used for display purposes. *func* is a pointer to the function used to simulate the unit's action. *potential* is the activation level for the unit. *data* is a four byte value for the unit *data* field described above. *output* is the initial output of the unit. *state* is a short integer representing the initial state value. *ipot* and *istate* are the values to set the unit potential and state when the network is reset. MakeUnit returns the index in the unit array of the unit created. The first call to MakeUnit builds the unit with index 0, and consecutive calls to MakeUnit will return consecutive indices. An example of a call to MakeUnit would be:

    unit_index = MakeUnit("retinal",UFsum,500,500,0,50,1,1);

The function pointer may be NULL, in which case a function which does nothing will be called by the simulator to simulate the unit action. If not NULL, the function must be either one you have written, or one of the library functions. For simple networks the library functions (see Section 2.2.11) should be sufficient.
2.2. NETWORK CONSTRUCTION

2.2.3 Adding sites

Once a unit has been created, one or more sites may be attached to it with calls to AddSite:

```c
Site * AddSite(index, name, func, data)
    int index, data;
    char *name;
    func_ptr func;
```

`index` is the index of the unit to which the site is to be attached. `name` is a pointer to a character string which will be the name of the site. `func` is a pointer to the function to be called to simulate the action of the site. `data` is the four byte value to be placed in the site `data` field described above.

Links to the unit cannot be made until there is a site attached to the unit to which they may go. A call to AddSite might look like:

```c
AddSite(unit_index,"excite",SFweightedsum,0);
```

AddSite returns a pointer to the newly created site structure. As with units, the function may be NULL, one of your functions, or one of the library functions.

2.2.4 Making links

A link from a unit to a site on another unit is created with a call to MakeLink:

```c
Link * MakeLink(from,to,site,weight,data,func)
    int from,to,weight,data;
    char *site;
    func_ptr func;
```

`from` is the index of the unit where the link originates. `to` is the index of the unit to which the link is going. `site` is a pointer to a character string which is the name of the site on the destination unit at which the link is to arrive. `weight` is the weight to put on the link, and should be within range of a short integer. By convention weights are scaled down by a factor of 1000, thus a specified weight of 500 will be treated as a weight of 0.5. This is to allow weights in the range 0 to 1 without having to use floating point arithmetic. Weights may be negative. MakeLink returns a pointer to the link structure created. An example of a call to MakeLink might be:

```c
MakeLink(unit_index, unit_index, "excite", -500, 0, LFsimple);
```

This would make a link from the unit to itself, to be attached at the site "excite", with a weight of -500 (meaning -0.5), and function LFsimple. Such a link could be used to provide exponential decay. As with units, the function may be NULL, one of your functions, or one of the library functions.
2.2.5 Naming units

Each unit may be given a name with a call to NameUnit:

\[
\text{NameUnit(name, type, index, length, depth)}
\]

\[
\text{char *name;}
\text{int type, index, length, depth;}
\]

As well as naming a single unit, this function can name a vector or 2-D array of units. The name may then be used during simulation from the command interface (see Section 2.4), and may also be used during network construction. \text{name} is a pointer to the character string name to be given. \text{type} is the type of name – SCALAR, VECTOR, or ARRAY. \text{index} is the index of the unit to be named, or the first unit in the vector or array. \text{length} is the number of units if it is a VECTOR, and the number of columns if it is an ARRAY, and is undefined for SCALAR. \text{depth} is the number of rows for an ARRAY, and is undefined for SCALAR and VECTOR.

A name which is specified as VECTOR or ARRAY will apply that name to the unit with the index specified, and to the requisite number of units following it in the unit array. Thus the call:

\[
\text{NameUnit("Vertex", ARRAY, 100, 4, 2);}
\]

will apply the name “Vertex” to units 100 through 107, making an array of 2 rows of 4 units. Now, for example, unit 107 will be displayed by the simulator with the name “Vertex[1][3]”.

All names must be unique. This applies to state, site, function, type and set names, as well as unit names.

2.2.6 State names

A name may be given to a state. Internally the state is just a short integer, but for display purposes it is much clearer if the state of a unit is printed as a name rather than merely a number. A state name is declared with, for example:

\[
\text{DeclareState("active",1);}
\]

which assigns the name “active” to the state represented by number 1. Now whenever a unit is displayed, if its state field has value 1 the simulator will print the name “active”, otherwise it will print the state value as an integer. A maximum of 100 state names may be declared, corresponding to state values 0 to 99.
2.2. NETWORK CONSTRUCTION

2.2.7 A simple example

The following sample program will build a network of 10 units, each one linked to all the others with inhibitory links. This kind of structure is known as a winner-takes-all network.

```c
#include "sim.h"
build()
{
    int i, j, index;
    AllocateUnits(10);
    for (i = 0; i < 10; i++)
    {
        index = MakeUnit("competing", UFsum, 0, 0, 0, 0, 1, 1);
        AddSite(index, "inhibit", SFweightedsum, 0);
    }
    for (i = 0; i < 10; i++)
    for (j = 0; j < 10; j++)
        if (i != j)
            MakeLink(i, j, "inhibit", 0-(random()%1000), 0, NULL);
    DeclareState("active", 1);
    NameUnit("W-T-A", VECTOR, 0, 10, 0);
}
```

First space for the 10 units is allocated. Then, in the first for loop the 10 units of type "competing" are constructed, and a site with name "inhibit" is added to each one. The functions UFsum and SFweightedsum are library functions. UFsum simply sets the unit potential and output to be the sum of the values of all the units sites (in this case, just one). SFweightedsum sets the value of the site to be the weighted sum of the incoming link values. All the initial values for the unit fields are set to zero, apart from the states, which are set to 1.

The second for loop constructs all the links: one from each of the other units, all attached to the sites "inhibit". The weights are set to a random value in the range 0 to 1000 (representing 0 to 1). The name "active" is associated with state value 1, and the 10 units are named as a vector "W-T-A". The first unit built is always unit 0, so the index in the call to NameUnit can safely be assumed to be 0.

2.2.8 Flags

Each unit has 32 flags associated with it. Currently flags 0 to 6 are used by the simulator, and flags 7 to 11 are reserved for future simulator use. Flags 12 to 19 should be used for library packages, and so user code should be restricted to flags 20 through 31, preferably working from 31 down. Some of the simulator reserved flags may be set by the user for one or more units.
CHAPTER 2. SIMULATOR USER MANUAL

The user-settable flags are as follows:

- **SHOW_FLAG** if set then the unit is in the Show set (see Section 2.4.7.3).
- **LIST_FLAG** if set then the unit is in the List set (see Section 2.4.7.2).
- **NO_LINK_FUNC_FLAG** if set then no functions are called for the links into a unit. This will result in speed up.
- **NO_SITE_FUNC_FLAG** if set then no site or link functions are called for the unit.
- **NO_UNIT_FUNC_FLAG** if set then no unit, site or link functions are called for the unit. The output of the unit remains the same.

Flags may be set when the network is constructed (i.e. in the build program) or during simulation (i.e. by unit functions), or in fact by any user function. The macros used to set, clear, and test flags are:

- `SetFlag(unit_index, flag)`
- `UnsetFlag(unit_index, flag)`
- `TestFlag(unit_index, flag)`

`TestFlag` computes TRUE if the flag is set for the unit, FALSE otherwise. For efficiency purposes, macros which use pointers to the unit (to avoid indexing into the unit array) are also available: `SetFlagP`, `UnsetFlagP`, `TestFlagP` (unit_pointer, flag). These enable flag setting in a tight loop in an efficient manner.

### 2.2.9 Sets

The user may create, modify and delete sets of units. The maximum number of sets at any instant is 32. Unless otherwise stated, the set functions return TRUE if the function succeeded, and FALSE otherwise. In general the return value will only be FALSE if any of the sets specified do not exist, or cannot be created. The set functions are:

- **DeclareSet(name)** - creates a set.
- **DeleteSet(name)** - deletes a set.
- **AddToSet(name, low, high)** - adds unit with indices low through high to the set.
- **RemFromSet(name, low, high)** - removes units low through high from the set.
- **UnionSet(name3, name1, name2)** - assigns the union of sets name1 and name2 to the set name3. Creates set name3 if it does not already exist.
- **IntersectSet(name3, name1, name2)** - assigns the intersection of sets name1 and name2 to the set name3. Creates set name3 if it does not already exist.
**2.2. NETWORK CONSTRUCTION**

DifferenceSet (name3, name1, name2) – assigns all units in set name1 but not in set name2 to the name3. Creates set name3 if it does not already exist.

InverseSet(name1, name2) – assigns all units not in set name2 to be in set name1. Creates set name1 if it does not already exist.

MemberSet (name, unit_index) – returns TRUE if the unit is in the set with the given name, FALSE otherwise.

IsSet(name) – returns TRUE if the name is the name of a set, FALSE otherwise.

Sets are a useful way to impose some structure on an otherwise amorphous mass of units. Unit functions may add or subtract a unit from a set. The sets are known to the command interface (see Section 2.4) by name when simulating, and so simulation commands can be applied to them.

### 2.2.10 Activation Functions

The library contains some standard unit, site and link functions, but user-written functions may be used. Unit, site and link functions are called by the simulator during each time step.

#### 2.2.10.1 Link functions

Let us look at the library function *LFsimple* for a sample link function.

```
LFsimple(up,sp,lp)
    Unit *up;
    Site *sp;
    Link *lp;
{
    lp->data = *(lp->value);
}
```

Three pointers are passed as parameters when the simulator calls a link function: a pointer to the unit to which the link is attached; a pointer to the site at which the link arrives; and a pointer to the link itself. *LFsimple* uses only the link pointer, and simply stores the incoming value in the link data field. In effect the data field is being used as a one-simulation-step memory. Check the Advanced Programming Manual for complete details of Link structure. Note that the value field in the Link structure is a pointer.
2.2.10.2 Site functions

A commonly used site function, from the library, is \textit{SFweightedsum}:

\begin{verbatim}
SFweightedsum(up, sp)
    Unit *up;
    Site *sp;
{
    int sum;
    Link *lp;
    for(lp = sp->inputs, sum = 0; lp != NULL; lp = lp->next)
        sum += (*(lp->value) * lp->weight);
    sp->value = sum/1000;
}
\end{verbatim}

Pointers to the unit to which the site is attached, and to the site itself, are passed in as parameters to site functions. \textit{SFweightedsum} simply trips down the linked list of incoming links, accumulating the weighted sum of the incoming values. The end of the list is terminated with a NULL \textit{next} field in the final Link structure. The weighted sum is divided by 1000 (the weight scaling factor), and the result set in the site \textit{value} field.

2.2.10.3 Unit functions

One of the simplest possible unit functions is \textit{UFsum}, which simply sums the site values:

\begin{verbatim}
UFsum(up)
    Unit *up;
{
    int sum;
    Site *sp;
    for(sp = up->sites, sum = 0; sp != NULL; sp = sp->next)
        sum += sp->value;
    up->output = up->potential = sum;
}
\end{verbatim}

Unit functions are passed a pointer to the unit structure when called by the simulator. This function trips down the linked list of sites attached to the unit, summing up the values. The final sum is set in the \textit{potential} and \textit{output} fields of the unit structure, thus setting the value that will be transmitted along outgoing links.
2.2.11 Library functions

2.2.11.1 Unit functions

UFsum is a unit function which sets output and potential to the sum of all site values.

2.2.11.2 Site functions

SFmax sets the site value to the maximum input value.

SFmin sets the site value to the minimum input value.

SFsum sets the site value to the sum of the input values.

SFweightedmax sets the site value to the maximum weighted input value. A weight of 1000 is treated as unity; the input value is multiplied by its weight and the result divided by 1000.

SFweightedmin(up,sp) sets the site value to the minimum weighted input value. A weight of 1000 is treated as unity.

SFweightedsum sets the site value to the sum of the weighted input values. A weight of 1000 is treated as unity.

SFand returns 1 if all its inputs are positive, otherwise 0.

SFxor(up,sp) returns 1 if exactly one of its inputs is nonzero, otherwise 0.

SFprod returns product of inputs.

2.2.11.3 Link functions

LFsimple sets the data field of the link to be the input value (unweighted). This does not affect the behavior of the network, but does help with debugging.

2.2.12 Another simple example

Let us modify the winner-takes-all example to demonstrate the use of sets and flags.

Since the link functions were specified to be NULL, i.e. a function which does nothing, we could equally well set the NO_LINK_FUNC_FLAG for each unit, saving the time taken to call the null function for each link.

We shall also create a set “still-competing” which will contain all the units whose potential is greater than 0. This set could then be displayed during simulation to view the winner-takes-all inhibition process.
```c
#include "sim.h"
build()
{
    int i, j, index;

    AllocateUnits(10);
    for (i = 0; i < 10; i++)
    {
        index = MakeUnit("competing", UFmysum, 0, 0, 1, 1);
        AddSite(index, "inhibit", SFweightedsum, 0);
        SetFlag(i, NO_LINK_FUNC_FLAG);
    }
    for (i = 0; i < 10; i++)
    for (j = 0; j < 10; j++)
        if (i != j)
            MakeLink(i, j, "inhibit", 0-(random()%1000), 0, lULL);
    DeclareState("active", 1);
    NameUnit("W-T-A", VECTOR, 0, 10, 0);
    DeclareSet("still-competing");
    AddToSet("still-competing", 0, 9);
}

The network building program has changed very little. The NO_LINK_FUNC_FLAG is set
for each unit as it is made, and the unit's data value is set to be the index of the unit. At
the end of the program we declare the set "still-competing" and add the 10 units to it. But
the major change is we no longer use the library function UFsum. Instead we write our own
function, UFmysum, which in addition to setting the output and potential, adds or removes
the unit from the set.

UFmysum(up)
    Unit *up;
{
    int sum;
    Site *sp;

    for(sp = up->sites,sum = 0;sp != NULL;sp = sp->next)
        sum += sp->value;
    up->output = up->potential = sum;
    if (sum <= 0)
        RemFromSet("still-competing",up->data,up->data);
}

As each unit is simulated, it checks if it is still in the competition; if not, it removes itself
from the "still-competing" set, using its index which was stored in the data field when
2.3. MAKING AN EXECUTABLE SIMULATOR

the unit was created. Now if the set “still-competing” is displayed at every step during simulation, the winner-takes-all process will become apparent.

2.2.13 Modular construction

One of the most important aspects of network construction is a modular approach. The actual size and configuration of network can be specified at the highest level in a data file, containing an abstract version of the problem being modeled. At the next level, the build function is used to control the gross aspects of network construction. Another level down, separate functions can be written to construct the different types of units and links. This corresponds to a hierarchy of descriptive levels and is crucial for building large networks. Section 2.7 gives an example of this approach.

2.3 Making an executable Simulator

Before a network can be simulated, the program to build it must be compiled and linked in with the simulator object files. A shell script makes this task simple. The name of the shell script is makesim, and it is normally found in the `/bin` directory – but this is site dependent. There are a number of flags which may be specified, if you specify none it will assume you are running on a SUN workstation and will create an integer simulator with the graphics interface. To create such a simulator, assuming the network building program is in the file `build.c`, simply type:

```
makesim build.c
```

This will create an executable binary file `sim` which is the simulator. When `sim` is run, the graphics interface window will appear and simulation commands, described in the next section, may be executed.

The complete specification for `makesim` is given in the man page. The simulator man directory should be on your manpath. The most commonly used flags are:

```
-ng do not load graphics interface.
-g compile user code for debugging.
-r run the executable when it has been created.
-o <file> file name for the executable.
-f compile for floating point version of simulator
-t integrate ‘propagation delay’ version of simulator
-v verbose: display the shell commands used to create the simulator
```

Multiple C files ending with `.c` and object files ending with `.o` are allowed. For instance, the line:
makesim build.c initialize.c test.o print.o

would compile the files build.c and initialize.c and load the resulting object files with test.o and print.o and the simulator object files, to make the executable.

Libraries can also be included. The line:

makesim build.c mylib.a neurolib.a

would compile the file build.c and load the resulting object file with the simulator object file, together with appropriate code extracted from the library files mylib.a and neurolib.a.

The standard `cc` switches `-ldirectory` (for included files), and `-Ldirectory` and `-llibrary` (for libraries) are passed through to the compiler or the loader.

### 2.4 Simulation

When a simulator executable is run, the startup message will be printed and commands can be typed when the prompt appears. The startup message will be something like:

Rochester Connectionist Simulator (version 4.2 patchlevel 0)

    Copyright (C) 1989 University of Rochester
    This software comes with ABSOLUTELY NO WARRANTY; for details, type 'status warranty'. This is free software, and you are welcome to redistribute it under certain conditions; type 'status copying' for details.

integer version
no propagation delay

Debugging turned on, not in Auto-Fix mode
->

The command interface prompt is ->. The startup message indicates what kind of simulator is running: integer or floating point. Debugging and Auto-Fix are described below. The simplest command is ?. Typing this to the prompt will result in all the command names being listed.

#### 2.4.1 Help

The `help` command has the syntax:
2.4. Simulation

help [<item>]
<item> ?

*help* alone will print the standard help message, describing the types of commands available, and how to exit the simulator. *help name* and *name ?* are equivalent, and print help information about *name*. This should be a command name or the special item *UnitId*. For instance, the command *help display* will print information as to what the *display* command does. Many of the commands require a specification of a unit or range of units, which can involve unit indices, unit names, set names or a combination of these. *help UnitId* will print information as to how to specify units for commands.

2.4.2 Building the network

The network building program was linked in with the simulator to produce the executable which is now running. Before the network can be simulated, it must be constructed by running this network building program. By convention, the top level function in the network building program is *build*. To execute this function, the *call* command (see Section 2.4.4) is used, in this case:

```
call build
``` 

2.4.3 Debugging during network construction

It is often the case that network building programs contain errors, for instance specifying a link between two units, one of which does not exist. The simulator can be put in *debug* mode, in which case these kinds of errors do not cause a core dump, but rather are automatically fixed, or cause a *debug interface* to be entered.

2.4.3.1 Debug command

**Syntax:**    debug [[ auto ] <on | off>]

**Example:**  debug auto on

The debug command is used to switch network construction debugging on and off, and to switch automatic error correction on and off. *debug on* and *debug off* will control whether debugging is operative. *debug auto on* and *debug auto off* will control whether automatic error correction is in operation. If automatic error correction is switched on and a log file is being maintained, then the user may accumulate a list of errors to be fixed, just as a conventional compiler produces a list of errors before aborting.
Debugging only applies to network construction, not to simulation. If debugging is switched on, then any time the simulator tries to make a unit, site, or link, the simulator will check that the values specified are suitable. For instance, if a function for a unit that is not known to the simulator is specified, the simulator will issue an error message. If automatic correction (Auto-Fix) is switched on, the simulator will substitute what it thinks is a suitable value, and continues.

If Auto-Fix is off, the simulator will enter a debug interface (see Section 2.4.3.2) and ask for the errors to be fixed. The set command (see Section 2.4.3.3) is used to change incorrect values. Once all the errors have been fixed, the quit command may be used to return to the normal network construction process. Continuation is not possible until the errors have been fixed, unless the ignore command (see Section 2.4.3.4) is issued.

If debugging is turned off, absolutely NO CHECKING is done – a core dump will occur if anything is wrong during construction. It is best to build and test a network with debug switched on until it is clear that the build function is correct. Building without checking appears to be approximately twice as fast as with checking.

If debugging and Auto-Fix are switched on, and errors or warnings are occurring rapidly, typing control.C which will introduce the interrupt interface (see Section 2.4.3.5) whence the network may be examined, Auto-Fix switched off, etc. If control.C is typed when the network is not in a safe state, the interrupt will be delayed.

2.4.3.2 Debug interface

The debug interface is signaled by a different prompt:

```
  debug[n]>
```

where n is an integer indicating the interface level. Level 0 is the normal command interface. Most of the commands available at level 0 are available to the debug interface, but none of the commands that cause a simulation step, such as go or read. When the debug interface is entered, a list of errors that need fixing, and for which unit, site or link, is displayed. The list of errors may be displayed at any time with the set command. When all the errors have been fixed with the set command, exiting the debug interface with quit will cause the building process to continue. Exiting with ignore will cause the building process to continue, but the unit, site or link will not have been made.

The way in which debug levels may accumulate, is that on being put in the debug interface, say level 1, the user may decide that a unit should be made on the fly, with the MakeUnit command. If the specification given for this unit is incorrect, the next level debug interface will be entered to fix this specification. On exit from a debug interface, if one returns to another lower level debug interface then the set command may be used to recall what the errors were that caused this interface to be entered originally.
2.4.3.3 Set command

Syntax:  
set <pot|out|state|ipot|istate|unit|to|from|weight> <value>
set <func|type|site> <name>
set all default
set
Example: set func SFweightedsum

The *set* command, which is only available at the debug interface, is used to correct errors in unit, site or link specifications. Errors may be fixed with specific values, or the simulator default values may be used: *set all default*. The list of outstanding errors will be printed in response to: *set*. It will also be printed if the user attempts to *quit* before all the errors have been corrected. For example, suppose one tried to build a unit with the *MakeUnit* command (see Section 2.4.8.2):

```plaintext
-> MakeUnit mytype NullFunc 1 2 3 4 1234567 1234567
   The following errors were encountered while trying to
   make unit 1 of type mytype and function NullFunc:
   - initial state value 1234567 out of range -/+32767
   - state value 1234567 out of range -/+32767
debug[2]> quit
   There are still errors in the unit definition:
   - initial state value 1234567 out of range -/+32767
   - state value 1234567 out of range -/+32767
debug[2]> set
   Current values for MakeUnit are:
   - type = mytype
   - function = NullFunc
   - state = 1234567
   - init state = 1234567
   - potential = 2
   - init potential = 1
   - output = 4
   - data = 3
   remaining errors are;
   - initial state value 1234567 out of range -/+32767
   - state value 1234567 out of range -/+32767
   fix and quit, or type ignore
debug[2]> set all default
default[2]> quit
Made unit 1
->
```
2.4.3.4 Ignore command

Syntax: ignore

The ignore command, which is only available at the debug interface, is used to continue the construction process without constructing the unit, site or link whose specification was in error. This may be expedient, but may also cause further errors in the construction process at a later time.

2.4.3.5 Interrupt interface

The interrupt interface is entered whenever the user types control.C. The interrupt interface is signaled by a different prompt:

interrupt[n]>

where n is an integer indicating the interface level. Level 0 is the normal command interface. Most of the commands available at level 0 are available to the interrupt interface, but none of the commands that cause a simulation step, such as go or read. If control.C is typed during a piece of guarded code, a message will be printed and entry to the interrupt interface will be delayed until the guarded code is exited.

The intention behind the interrupt interface is that the user can first of all manipulate the debug settings (whether on or off, whether automatic or manual fixing), and secondly can interrupt a potentially catastrophic situation to save or examine the network before it is corrupted or destroyed.

2.4.4 Calling functions

Any function in the user code can be called from the interface, using the call command, so long as the function was not declared static in the code (see the C manual for an explanation of this). The call command has the syntax:

Syntax: call function-name [<arg1> <arg2> ....]
Example: call Initialize image.inp.1

This will call the function with the optional arguments as parameters. The only required use of this command is to construct the network, as described in the preceding section. An example, as above, of an additional use would be to call a function which initializes a set of units representing a retinal image array from file data.

Parameters are passed into your function in a manner similar to the way C passes command line arguments into the mainline: via an argv, argc-type formal parameter list, where
2.4. SIMULATION

(argc) is an integer containing the number of arguments and (argv) is an array of char pointers to these arguments. The argument list includes all tokens following the call command. So in the example above, argc would be set to 2, with argv[0] pointing to the string "Initialize" and argv[1] pointing to the string "image.inp.1". For an example of how to write your function to extract parameters, examine the build function in the map-coloring network shown in section 2.7.4.

2.4.5 Unit specification in commands

Many commands expect one or a range of units to be given, to which the command will be applied. A single unit may be specified by index or by name. A range of units may be specified by the low and high units separated by – (with mandatory space either side of the –), by a set name, by a VECTOR or ARRAY name, or by the token all.

For example, suppose units 0 to 9 have names ZERO, ONE, TWO, ..., NINE, units 10 to 19 are a vector with name MyVector, units 20 to 39 are a 4 by 5 array of units with name MyArray, set FirstSet consists of all the odd-index units, and these 40 units are all that have been made.

Then the following unit specifications are valid and indicate these units:

<table>
<thead>
<tr>
<th>Specification</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ONE</td>
<td>1</td>
</tr>
<tr>
<td>0 - 3</td>
<td>0,1,2,3</td>
</tr>
<tr>
<td>THREE - 5</td>
<td>3,4,5</td>
</tr>
<tr>
<td>MyVector</td>
<td>10,11,12,...,19</td>
</tr>
<tr>
<td>MyArray</td>
<td>20,21,22,...,39</td>
</tr>
<tr>
<td>MyVector[4] - 19</td>
<td>14,15,16,17,18,19</td>
</tr>
<tr>
<td>FirstSet</td>
<td>1,3,5,...,39</td>
</tr>
<tr>
<td>all</td>
<td>0,1,2,...,39</td>
</tr>
</tbody>
</table>

As can be seen, mixed modes are allowed.

2.4.6 Simulation commands

The simulation commands are used to modify aspects of the simulation, and to cause one or more steps to be simulated.
2.4.6.1 Sync command

Syntax:  sync

The sync command sets the simulation to be synchronous. This means that at each simulation step, every unit will be updated, using the output values calculated from the previous step. The network behaves as if all units update simultaneously. This is the default setting for the update protocol.

2.4.6.2 Fsync command

Syntax:  fsync <execfrac> <execlimit> [<random seed>]
Example:  fsync 10 100 2039

The fsync command sets the simulation style to be asynchronous. This means that at each time step a percentage of units (given by execfrac) picked pseudo-randomly from all the units are simulated, and that after a limiting number of steps (given by execlimit) all units will have been simulated at least once. The new output of a unit is available immediately the unit is simulated, for other units to use. Thus the network should not be sensitive to the order in which units are simulated. The point by which all units must have been simulated at least once (execlimit) is determined with reference to the simulator Clock (see Section 2.6.1). When the clock is an exact multiple of execlimit, any unit that has not been simulated is simulated.

If an integer number is given for random seed then the random number generator will be seeded with this number. Seeding with the same number on separate occasions will cause the generator to produce the same sequence of random numbers, thus by specifying the same seed a session involving asynchronous simulation can be repeated exactly. If no seed is specified, the UNIX system time will be used.

This style of execution becomes more inefficient as the execution fraction increases (for implementation reasons). For an execution fraction of 100%, the async command should be used.

2.4.6.3 Async command

Syntax:  async [<random seed>]
Example:  async 2039

This command is a special case of the fsync command. Logically it has the same effect as if fsync 100 1 were given, i.e. that at each time step all units will be simulated in pseudo-
random order. For implementation reasons it much better to use this command than the equivalent \texttt{fsync} command.

2.4.6.4 Go command

Syntax: \texttt{go [clock] [StepCount]}

Example: \texttt{go clock 100}

The \texttt{go} command causes one or more simulation steps to be run. \textit{StepCount} specifies the number of steps to run, and defaults to 1. If the \textit{clock} option is used, the simulator will time how long it takes to simulate the number of steps, in increments of 1 second. This can be used to get a precise idea of network code efficiency. Since the time quantum is 1 second, a significant number of steps must be simulated to get reliable timing data.

2.4.6.5 Echo command

Syntax: \texttt{echo [StepCount | on | off]}

Example: \texttt{echo 10}

The \texttt{echo} command sets how often the simulator prints the echo message:

\texttt{finished }\textit{x} \texttt{out of }\textit{n} \texttt{steps}

Echoing occurs (if it is switched on) during execution of a \texttt{go} command. \textit{n} is the number of steps specified in the \texttt{go} command, and \textit{x} is a multiple of the \textit{StepCount} specified in the \texttt{echo} command. For example:

\texttt{-> echo 5}
\texttt{-> go 20}
\texttt{finished 5 out of 20 steps}
\texttt{finished 10 out of 20 steps}
\texttt{finished 15 out of 20 steps}
\texttt{finished 20 out of 20 steps}
\texttt{->}

Echoing is switched off with: \texttt{echo off}; and on with: \texttt{echo on}. At start up echo is switched on. Issuing the \texttt{echo} command without any of the options will cause the current setting to be displayed.
2.4.7 Examination commands

The examination commands are used to print out details of units and links, either automatically or at the user's request.

2.4.7.1 Display command

Syntax: disp unit <UnitID>
Example: disp unit 33

The disp command is used to display the values associated with one or more units, for instance the potential, output, state, functions, site names and values, link weights and values. The name and index of the unit where the link originates is shown in each link display. For example:

```
-> disp unit 0
Unit:0 Name:W-T-A[0] Type:competing function:UFsum
    potential:10 output:10 state:active data:0
Set memberships: still-competing
    sitename:inhibit function:SFweightedsum value:0 data:0
```

2.4.7.2 List command

Syntax: list <link | set | unit UnitId>
Example: list unit 3 - 5

The list command is used to display information about links, sets or units. list link displays all the links. list set displays all the set names and the number of remaining slots for sets. list unit UnitId will display in compact form the units specified by UnitId. For example:
2.4. SIMULATION

-> list unit 3 - 5

<table>
<thead>
<tr>
<th>Index</th>
<th>Name</th>
<th>Type</th>
<th>Potential</th>
<th>Output</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td><strong>NO NAME</strong></td>
<td>vertex</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td><strong>NO NAME</strong></td>
<td>vertex</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td><strong>NO NAME</strong></td>
<td>vertex</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

2.4.7.3 Show command

Syntax: show [on | off]
        show <step | pot > <value>
        show +|- <UnitId>
        show set [(+|-) <set name>]

Example: show set + still-competing
        show + 3 - 5

The show command is used to control what information is displayed during simulation. Showing is turned on or off with show on or show off. If showing is turned on, then every n steps the set of units selected for showing (the Show set) is displayed in compact form, where n is set with show step n.

The Show set is controlled in various ways. If a unit has a potential greater than or equal to the Show potential, it is in the show set. The Show potential is set with show pot value, and is initially a very large number. A unit or range of units may be added to the Show set with show + UnitId, and removed from the Show set with show - UnitId. A set of units may be added to or removed from the Show set with show set +/- set-name. Although show + set-name is equivalent to show set + set-name, the latter is more efficient. For example:
-> show + 3 - 5
-> show on
-> show step 2
-> go 2
finished 1 out of 2 steps
finished 2 out of 2 steps

<table>
<thead>
<tr>
<th>Index</th>
<th>Name</th>
<th>Type</th>
<th>Potential</th>
<th>Output</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td><strong>NO NAME</strong></td>
<td>vertex</td>
<td>0</td>
<td>-16643</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td><strong>NO NAME</strong></td>
<td>vertex</td>
<td>0</td>
<td>-31767</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td><strong>NO NAME</strong></td>
<td>vertex</td>
<td>0</td>
<td>-24072</td>
<td>0</td>
</tr>
</tbody>
</table>

2.4.7.4 Pipe command

Syntax: pipe < on | off | command >
Example: pipe /usr/ucb/more

Output from display, list, and show commands can be fed into a pipe rather than directly to the screen. The default pipe is the UNIX more command; this simply avoids large displays scrolling off the screen. Another use might be to save displays to a file, with for example:

```
pipe cat >> save.file
```

If piping is turned on (pipe on) then at every display, list or show, the output is sent to the pipe. Piping may be turned off with pipe off.

2.4.7.5 Pause command

Syntax: pause < on | off >
Example: pause on

The pause command is used to avoid displays scrolling off the screen. If pausing is switched on, then after every show the simulator waits for user indication to continue.

2.4.7.6 Status command

Syntax: status
2.4. SIMULATION

The status command displays the state of the simulator. For example:

```
-> status
Clock: 5  Show is on
NoUnits: 10  ShowPot: 3
NoLinks: 30  NoSets: 0
Echo every 1 steps  Pause is on
Pipe is on  PipeCommand is /usr/ucb/more
Simulation is synchronous
->
```

2.4.8 Modification commands

There are two types of modification commands: those that alter the network structure; and those that alter the state of the network. The former commands are those to make a unit, site, or link. In addition the commands to allocate data space for units and to give a name to one or more units are included in this section.

2.4.8.1 AllocateUnits command

Syntax: AllocateUnits <number>
Example: AllocateUnits 200

AllocateUnits is used to create data space for the units. You may call AllocateUnits as many times as you wish to add more space later on. You should normally call AllocateUnits before your first call to MakeUnit. If you don't, the simulator will automatically allocate enough space for 1000 units. If you try to construct more units than there is space for, the simulator will allocate more units in blocks the same size that it last allocated.

2.4.8.2 MakeUnit command

Syntax: MakeUnit <type> <function> <ipot> <pot> <data> <out> <istate> <state>
Example: MakeUnit competitor UFsum 0 0 0 0 1 1

MakeUnit is the simulator command to make one unit. It takes the same arguments as the simulator function to make a unit (see Section 2.2.2), namely type, function, initial potential, potential, data, output, initial state, and state, in that order. The first parameter, type, is simply stored in the name table by the simulator for display purposes. Any name that is not already in use (i.e. not a name of a set, state, function or unit(s)), may be used here. The second parameter, function, is the unit function, which must be known to the
simulator \textit{(i.e.} a library function or a function in user files, or NULL for the function which does nothing). The remaining arguments are integer values for the various fields of the unit. Initial potential is the unit potential after a reset. Potential is the current unit potential. Data is for you to use as you wish. Output is the current unit output. Initial state is the unit state after a reset. State is the current unit state.

Defaults for the numeric arguments are zero. Default for the function is the function NullFunc (does nothing). Default for type is the name NullType. Debugging is automatically switched on for the duration of this command.

2.4.8.3 AddSite command

\textbf{Syntax:} AddSite \textless unit\textgreater{} \textless sitename\textgreater{} \textless function\textgreater{} \textless data\textgreater{}

\textbf{Example:} AddSite 9 "excite" SFweightedsum 0

\textit{AddSite} is the simulator command to add a site to a unit. It takes the same arguments as the simulator function to add a site \textit{(see Section 2.2.3)}, namely: unit index, site name, site function, and site data. A site with the given name and function is attached to the unit with the given index, and the data field is set as given. Any name that is not already in use \textit{(i.e.} not a name of a set, state, function, type or unit(s)) may be used. The site function must be known to the simulator \textit{(i.e.} a library function or a function in user code or NULL for the function that does nothing). The unit index must be that of an existing unit. Data is for general use.

The defaults for site name, site function and data are NullSite, NullFunc and 0, respectively. The unit to which a site is to be attached MUST be specified. Debugging is automatically switched on for the duration of this command.

2.4.8.4 Deletesites command

\textbf{Syntax:} deletesites \textless UnitID\textgreater{} \textless sitename\textgreater{}

\textbf{Example:} deletesites 0 - 20 excite

The deletesites command deletes one or more sites. Deleted sites are returned to the free list for use by the next AddSite command. Links arriving at the site(s) are also deleted. The unit index is that of the unit to which the site is attached. The sitename is the name of the site.

Many sites may be deleted with one command. The unit specifications may be given in any of the usual ways \textit{("help UnitId" for details). The site name may be "all"}, meaning all sites on the unit(s). To delete all the sites on all units, use \textit{"deletesites all all"}. To delete all sites named "excite" on units 35 through 70, use \textit{"deletesites 35 - 70 excite"}. 

2.4.8.5 MakeLink command

Syntax: MakeLink <from> <to> <site> <weight> <data> <function>
Example: MakeLink 3 5 "excite" 500 0 LFsimple

MakeLink is the simulator command to make a link from one unit to another. It takes the same arguments as the simulator function to make a link (see Section 2.2.4), namely: source unit index, target unit index, site name (on target unit), link weight, link data and link function. A link from the source unit to the named site on the target unit is made with function, weight and data as given. The link function must be known to the simulator (i.e. a library function or a function in your files, or NULL for the function which does nothing).

The specified weight is scaled down by the simulator by a factor of 1000, so 500 corresponds to a real weight of 0.5. data is for general use. Defaults exist for function (NullFunc), weight (0) and data (0). The source and target unit indices, and the name of the site on the target unit MUST be specified. Debugging is automatically switched on for the duration of this command.

2.4.8.6 Deletelinks command

Syntax: deletelinks <From-UnitID> <To-UnitID> <To-site>
Example: deletelinks 0 - 20 firstset excite

The deletelinks command deletes one or more links. Deleted links are put on the free list and used by the next MakeLink command. The first unit index is the originating unit. The second unit index is the receiving unit, and the sitename is the name of the site on the receiving unit to which the link is attached.

Many links may be deleted with one command. The unit specifications may be given in any of the usual ways ("help UnitId" for details). The site name may be "all", meaning all sites at the destination unit(s). To delete all the links, use "deletelinks all all all". To delete all links from units in set "source" to site "excite" on units 24 through 48, use "deletelinks source 24 - 48 excite".

2.4.8.7 NameUnit command

Syntax: NameUnit <scalar|vector|array> <index> [<width> [<depth>]]
Example: NameUnit array 10 4 5

NameUnit is used to give a name to one or more units. It takes the same arguments as the simulator function of the same name, i.e. a name, a type, the index of the first unit to
which the name is to be applied, the width (if a vector or array), and depth if an array. The possible types are: scalar, vector, array. For a scalar name, the name is applied to a single unit. For a vector name, the name is applied to width units starting with index. For an array name, the name is applied to width*depth units starting with index.

### 2.4.8.8 Out command

**Syntax:** \( \text{out} \ <\text{UnitID}> \ <\text{value}> \ [\text{<UnitID> <value>}]* \)

**Example:** \( \text{out} \ 3 \ 100 \ 4 \ 200 \ 5 \ 300 \)

The *out* command is used to set the output of one or more units. It expects one or more unit-identifier/output-value pairs. The unit identifiers may be specified in any of the usual ways (see Section 2.4.5).

### 2.4.8.9 Pot command

**Syntax:** \( \text{pot} \ <\text{UnitID}> \ <\text{value}> \ [\text{<UnitID> <value>}]* \)

**Example:** \( \text{pot} \ 3 \ 100 \ 4 \ 200 \ 5 \ 300 \)

The *pot* command is used to set the potential of one or more units. It expects one or more unit-identifier/potential-value pairs. The unit identifiers may be specified in any of the usual ways (see Section 2.4.5).

### 2.4.8.10 State command

**Syntax:** \( \text{state} \ <\text{UnitID}> \ <\text{value}> \ [\text{<UnitID> <value>}]* \)

**Example:** \( \text{state} \ 3 \ 10 \ 4 \ 20 \ 5 \ 30 \)

The *state* command is used to set the state of one or more units. It expects one or more unit-identifier/state-value pairs. The unit identifiers may be specified in any of the usual ways (see Section 2.4.5).

### 2.4.8.11 Weight command

**Syntax:** \( \text{weight} \ [\text{<From> <To> <sitename> <value | random [<mean> <deviation>]}>]+ \)

**Example:** \( \text{weight} \ 3 \ 5 \ \text{excite} \ 200 \ 3 \ 6 \ \text{excite} \ 100 \)

The *weight* command sets the value of a weight on a link. The first unit index is the originating unit. The second unit index is the receiving unit, and the sitename is the name
of the site on the receiving unit to which the link is attached. The weight is scaled up by a factor of 1000. Thus a weight of 500 indicates a real weight of 1/2. In the floating point simulator you may use floating point values as well as integers. Random weights may be assigned, e.g. the command:

```
weight 0 1 excite random 500 100
```

will assign a weight picked randomly from the range 400 to 600.

Multiple weight settings may be given with one command, as with the `out`, `pot`, and `state` commands. The unit specifications may be given in any of the usual ways ("help UnitId" for details). The site name may be "all", meaning all sites at the destination unit(s).

### 2.4.8.12 Ufunc command

**Syntax:** `ufunc <UnitID> <function> [<UnitID> <function>]`

**Example:** `ufunc 0 UFsum 1 UFproduct 2 - 9 UFmean`

The `ufunc` command is used to set the unit function of one or more units, or alternatively to turn the unit function on or off for one or more units. It expects one or more unit-identifier/function-name pairs. The unit identifiers may be specified in any of the usual ways. The functions may be any global user function or a library function, or one of the keywords "on" or "off". If one of the keywords is given, the effect is to set the `NO_UNIT_FUNC_FLAG` on or off.

### 2.4.8.13 Sfunc command

**Syntax:** `sfunc <UnitID> <site> <func|on|off> [<UnitID> <site> <func|on|off>]`

**Example:** `sfunc 1 excite SFsum 2 all SFmean 3 - 9 all off`

The `sfunc` command is used to set the site function of one or all sites on one or more units, or alternatively to turn on or off all the site functions on one or more units. It expects one or more unit-identifier/site-name/function-name triples. The unit identifiers may be specified in any of the usual ways. The site name may be the name of any site on the unit, or "all" meaning all sites. The functions may be any global user function or a library function, or the keywords "on" or "off". If one of the keywords is given, the sitename must be "all". The effect then is to set the `NO_SITE_FUNC_FLAG` on or off.
2.4.8.14 Lfunc command

Syntax: `lfunc [<SourceUnitId> <DestinationUnitID> <site> <func|on|off>]+`

Example: `lfunc 0 - 10 all excite LFsimple`

The `lfunc` command is used to set the link function of links from one or more units arriving at one or all sites on one or more units, or alternatively to turn on or off all the link functions on one or more units. It expects one or more unit-identifier/unit-identifier/site-name/function-name quadruples. The unit identifiers may be specified in any of the usual ways (type "help UnitId" for information).

The first unit-identifier specifies the units at which the affected links originate. The second unit-identifier specifies the units at which the affected links arrive. The site name is that of a site on the destination unit(s), or "all" meaning all sites. The function may be any global user function or a library function, or the keywords "on" or "off".

If one of the keywords is given, the sitename must be "all" and the source units (the first unit-identifier) must be all the units. The effect then is to set the NO_LINK_FUNC_FLAG on or off for the destination units.

2.4.8.15 Reset command

Syntax: `reset`

The `reset` command is used to reset the network to some initial state. It resets the potential and state of all units to their original values (as specified when they were made with `MakeUnit`). It also resets the outputs of all units to be zero. It does not reset weights or any other parameters.

This command has been somewhat superseded by the `checkpoint` and `restore` commands (see Sections 2.4.10.1 and 2.4.10.2).

2.4.8.16 Restart command

Syntax: `restart [<gi>]`

Example: `restart`

This command cleans out all the simulator data structures except the item table and the state names, so network construction can begin again from scratch. If you give this command from the Graphics Interface, it too will be "restarted" by clearing out all its representations of the current network. Appending the "gi" parameter to the `restart` command will "restart" just the Graphics Interface; the actual network constructed by the simulator remains unaffected (assuming the command was given from the Graphics Interface).
2.4.9 Set commands

Sets may be manipulated from the command interface as well as from within user code (see Section 2.2.9). Sets provide a way of imposing some structure on what is essentially an amorphous mass of units.

2.4.9.1 Addset command

Syntax: `addset <set name> <UnitId>`
Example: `addset still-competing W-T-A`

The `addset` command adds one or more units to a set. The units may be specified using any of the normal methods (see Section 2.4.5). If the set does not already exist it is created.

2.4.9.2 Remset command

Syntax: `remset <set name> <UnitId>`

The `remset` command removes one or more units from a set. The units may be specified using any of the normal methods (see Section 2.4.5).

2.4.9.3 Deleteset command

Syntax: `deleteset <set name>`
Example: `deleteset still-competing`

The `remset` command deletes a set. All units in the set are removed from the set and the name is reset to be unused.

2.4.9.4 Unionset command

Syntax: `unionset <answer set name> <set A> <set B>`
Example: `unionset winners still-competing-A still-competing-B`

The `unionset` command assigns the union of the second and third sets to the first set. All units which are in either the second set or the third set or both are put in the first (answer) set. If the answer set does not yet exist, it is created. If the answer set exists, then any units in the set which are not in the union of the second and third sets are removed from it.
2.4.9.5 Intersectset command

Syntax: intersectset <answer set name> <set A> <set B>
Example: intersectset winners still-competing-A still-competing-B

The intersectset command assigns the intersection of the second and third sets to the first set. All units which are in the second set and the third set are put in the first (answer) set. If the answer set does not yet exist, it is created. If the answer set exists, then any units in the set which are not in the intersection of the second and third sets are removed from it.

2.4.9.6 Diffset command

Syntax: diffset <answer set name> <set A> <set B>
Example: diffset winners still-competing-A still-competing-B

The diffset command assigns the difference of the second and third sets to the first set. All units which are in the second set but not the third set are put in the first (answer) set. If the answer set does not yet exist, it is created. If the answer set exists, then any units in the set which are not in the difference of the second and third sets are removed from it.

2.4.9.7 Inverseset command

Syntax: inverseset <answer set name> <set name>
Example: inverseset losers still-competing-A

The inverseset command assigns the inverse of the second set to the first set. All units which are not in the second set are put in the first (answer) set. If the answer set does not yet exist, it is created. If the answer set exists, then any units in the set which are not in the inverse of the second sets are removed from it.

2.4.10 File commands

It is possible to save the structure and/or state of a network to file, and load the saved file in later. It is also possible to make a log file of the session, both user input and simulator response, and to read in a file of simulator commands. The simulator automatically makes a log file of the commands typed in for each session, and at the end of the session (see Section 2.4.13.1) asks if it should be saved.

NOTE: Due to the implementation details of the back-propagation library (see Chapter 5), which uses (abuses) the link function pointer as a back-pointer to a
2.4. SIMULATION

It is not possible to save and restore back-propagation networks. Simulator versions prior to 4.1 would print an error message about not being able to find the name of the link function, and exit. Version 4.2 will print an error message, but will not exit. Perhaps future versions will actually fix this so that it works.

2.4.10.1 Checkpoint command

Syntax: checkpoint [file name]
Example: checkpoint conceptlearn

The checkpoint command is used to save the state of a network to file. The state consists of the values of unit parameters (e.g., weights, potentials, etc), the current sets and unit state names. The network state may be restored with the restore command (see Section 2.4.10.2). Related commands are save and load (see Sections 2.4.10.3 and 2.4.10.4), which save and reload the structure (pattern of units and links) of the network as well.

A name does not have to be specified for the checkpoint file, if it is not the simulator will construct one involving the UNIX process id number. The convention is that all checkpoint file names are of the form name.chk.n, where n is an automatically incremented integer.

For example:

-> checkpoint conceptlearn
chk file name [default: conceptlearn.chk.1] >
saving state
...............................
->

Here the simulator appended .chk.1 to the specified file name, and asked for confirmation. The row of dots indicate the checkpoint process, each dot representing ten units having been checkpointed.

2.4.10.2 Restore command

Syntax: restore <file name>
Example: restore conceptlearn.chk.1

The restore command restores the state of a network from a file made with the checkpoint command. This command simply restores the state of the network (i.e., the weights, potentials, etc), it does NOT rebuild the network. That means the network must already have been built. The command will check that the file used for restoration was made from the same simulator with the same network. If the simulator has been recompiled, but has
had the same network constructed in it, the warning message may be safely ignored. The simulator will also issue a warning if the checkpoint file was made during a previous session.

For example:

```
-> restore conceptlearn.chk.1
restoring unit state.............................
restoring set names
restoring state names
state restored.
->
```

2.4.10.3 Save command

**Syntax:** `save [<file name>]`

**Example:** `save conceptlearn`

The `save` command is used to save a the structure and state of a network to file. The structure of the network is the units, sites and links and associate names and functions. The state consists of the values of unit parameters (e.g. weights, potentials, etc), the current sets and unit state names. The saved network may be reloaded into an empty simulator with the `load` command (see Section 2.4.10.4). Related commands are `checkpoint` and `restore` (see Sections 2.4.10.1 and 2.4.10.2), which save and reload the state of the network only.

A name does not have to be specified for the save file, if it is not the simulator will construct one involving the UNIX process id number. The convention is that all save file names are of the form `name.net.n`, where `n` is an automatically incremented integer.

For example:

```
-> save conceptlearn
net file name [default: conceptlearn.net.1] >
saving name table ...
saving units.................................
saving state
...........................................
->
```

Here the simulator appended `.net.1` to the specified file name, and asked for confirmation. The row of dots indicate the save process, each dot representing ten units having been saved.
2.4.10.4  Load command

Syntax:  load <file name>
Example:  load conceptlearn.net.1

The load command loads the structure and state of a network from a file made with the save command. This command constructs the network (i.e. makes units, sites and links) and restores the state of the network (i.e. the weights, potentials, etc). The simulator should be empty: no network building functions can have been called. The command will check that the file used for loading was made from the same simulator executable, and issue a warning message if the executables are different. If the simulator has been recompiled, with the same functions available, the warning message may be safely ignored.

For example:

-> load conceptlearn.net.1
loading units................................. units loaded
restoring unit state..........................
restoring set names
restoring state names
state restored.
Done!
->

2.4.10.5  Log command

Syntax:  log [on|off]
Example:  log on

The log command is used to switch logging on and off. If logging is on, then everything typed at the keyboard, and everything sent to the screen by the simulator is saved in a log file. When logging is switched off, the current log file is closed. If logging is switched on again, the previous log file may be appended to, or a new log file may be started. For example:

-> log on
log file name [default: run2462.log.1] >
-> log off
-> log on
log file name [default: run2462.log.1] >
Overwrite file run2462.log.1 (y,n,a[pend]) ? a
->
The convention is that all log files are of the form name.log.n where n is an automatically incremented integer. The simulator constructs a logfile name from the UNIX process id number (as above), and asks for confirmation or another file name. If a network is constructed with debugging and Auto-Fix switched on, and a log file is kept, then errors may be accumulated in the file for one-time correction, just as compilers for conventional languages accumulate errors before aborting the compilation.

2.4.10.6 Read command

Syntax:  read <file name>
Example: read conceptlearn.cmd.1

The read command causes the simulator to read commands from a file. The commands in the file should have exactly the same format that commands typed on the keyboard have. read commands may be nested - in other words a file of commands that is being read may contain a read command to commence reading a different file, and return to reading the current one when the end of the new one is reached.

A command file containing the commands typed at each session is created, and at the end of the session the simulator asks if it should be deleted or kept (see Section 2.4.13.1). This enables easy rerun of a session. For instance, the following command script is used to control the simulation for the example in section 2.7.

```
call build map
show on
show pot 1
show set + change
pipe off
pause on
async
print pause are you ready
go 10
```

2.4.11 Compilation and Code-loading commands

The dynamic code loading features use the concept of a code unit. A code unit corresponds to an object file produced by the C compiler when given the -c flag. Object files produced in other ways (for example, by other compilers) may or may not be suitable for the dynamic loading facility. When a code unit is loaded into the simulator, the names and addresses of all global functions and global 4 byte uninitialized variables are put in the Item Table. The functions may then be called and the variables examined and modified from the interface. However no other code unit can reference any of the functions or variables loaded with the
2.4. SIMULATION

dynamic loading facility. Code units may only reference functions and variables that are in
the simulator itself, in the same code unit, in the backpropagation or regular libraries, or
in user-specified libraries.

A code unit may be compiled and loaded, its source code modified and then compiled
and loaded again. Only the latest version of the code unit is kept. If a given name is used
in two code units (for example, each contains a function called “build”), then only the item
with the given name in the most recently loaded code unit is accessible from the interface.
However, code in the older code unit may still reference the older item. Code units are
automatically deleted if they contain no accessible functions. Code units are specified at
the interface by the name of the object file without the .o extension. Internally the simulator
uses the full file name including extension.

Loading a code unit that contains a function with the same name as one currently used
as a unit, site or link function causes the new function to be used by all the appropriate
units, sites and/or links.

2.4.11.1 Compile command

Syntax:  compile <filename> ["<switches>"
Example: compile myfuncs "-I/mydir -O"

This command compiles a file of C code. The filename should be given without the .c
extension (so to compile “myfuncts.c” use the command: “compile myfuncts”). The resulting
object file will be named “filename.o”.

You may give your own switches to the compilation in a string as the third parameter.
For example, if you want to specify an include directory, you might use: “compile myfile
"-I /mydir -O””.

2.4.11.2 Combine command

Syntax:  combine <file> <file> [<file>]* - <dest file> ["<switches>"
Example: combine build funcs check - total

This command combines two or more object (.o) files into a single object file, so as to be
able to use the “loadcode” command with network construction and activation functions
in different files. Switches may be provide to the shell “ld” command as an optional last
parameter.
2.4.11.3 Loadcode command

Syntax: loadcode <filename> ["<libraries>"] [all]
Example: loadcode total "-L /lib -lneuro"

This command dynamically loads an object file (.o) into the running system. The filename should be given without the .o extension. The command is intended for loading of user functions, in particular unit, site and link functions, and parameters. Libraries to be searched may be given as a third parameter, in the usual format for the UNIX "ld" command. See below for examples.

The unit of code is an object file, NOT a function. Thus each code unit is known by the name of the object file (.o). Loading a code unit will cause any previously loaded code unit with the same name to be destroyed. Thus functions and data loaded with this command may be redefined. References in the new code unit to functions in a code unit loaded or reloaded previously CANNOT be resolved. Therefore all code units must be completely self contained except for references to functions which are part of the simulator or which were loaded when the simulation executable was made (i.e. which were loaded by the "makesim" UNIX level command).

All unit, site and link functions are updated to the latest version of their functions. All global functions and 4 byte UNINITIALIZED variables are put in the item table, so you can "call" the functions and modify the variables (see "value" command). Generally integers and floats are 4 byte variables.

You may specify library directories and libraries to be searched as a third parameter, a string. The -L and -l switches to "ld" may be given in this string (or in fact any other switches). For example, you might issue the command "loadcode mycode "-L jlib -lneuro"."

To see the shell commands issued to load your code, include the keyword "all" at the end of your command, i.e. after the codefile or the switches if you used any.

Example: you have an empty simulator, a file of unit functions ("func.c") and a file of network building functions ("build.c"). Compile these files using the "compile" command. The network building code makes reference to the unit functions, so they must be combined with the "combine" command before they can be loaded. Combine them into file "comb.o" (use "combine build func - comb"). Load "comb.o" into the simulator - "loadcode comb". Now you can call your network building function(s).

If after experimenting you need to change your unit functions, simply edit "funcs.c", recompile ("compile funcs") and reload ("loadcode funcs"). The new unit functions will be installed. If you want to rebuild the network, or add a piece of network, the unit functions must again be combined with the network building code before loading.
2.4. SIMULATION

2.4.11.4 Value command

Syntax: value <data name> <value | ? [float]>
Example: value decayrate 50

This command shows or sets the value of a C global variable in a section of code loaded with the “loadcode” command. Note that only 4 byte UNINITIALIZED variables are known, i.e. integers and floats. “value x 3” or “value x 3.0” will set x to 3 or 3.0 depending on whether it is a floating point or integer parameter. You MUST use the correct form, or the value will be set incorrectly. “value x?” prints the value of x as an integer, “value x? float” prints the value as a floating point number. Type conversion is NOT automatic, the default is integer and you must indicate floating point explicitly if required. Any variables which have an initializer will not be known (see the C manual for a definition of “initializer”).

2.4.11.5 Whereis command

Syntax: whereis <function-name | variable-name>
Example: whereis buildnet

This command looks for the given function or variable name and finds out which code unit, or the simulator executable, contains it. Note that for code units loaded at simulator creation time (i.e. with the makesim command) only functions are accessible. If you want a variable to be modifiable from the interface, you must either load its code unit with the “loadcode” command, or write a function yourself which you can call from the interface to modify it. If a function is a basic simulator function, or was loaded with the makesim command, the whereis command will report it as being in the base simulator code. Otherwise it will report which code unit loaded with the “loadcode” command contains it.

2.4.11.6 How to use the dynamic loading facilities

Any functions loaded when a simulator is made (i.e. with makesim) cannot be changed by dynamically reloading a code unit. Therefore, it is advisable to begin by making an empty simulator (use the makesim command without specifying any .c or .o files), and to load in all the network building and activation function code with loadcode. Typically network building code and activation function code will be in separate files, say build.c and activation.c. A sample session might go like this:
virtuous % /usr/connect/bin/makesim -X
simulator type is integer
loading function names
X11 graphics interface
creating simulator sim
virtuous % sim

Rochester Connectionist Simulator (version 4.2 patchlevel 0)

Copyright (C) 1989 University of Rochester
This software comes with ABSOLUTELY NO WARRANTY; for details, type 'status warranty'. This is free software, and you are welcome to redistribute it under certain conditions; type 'status copying' for details.

integer version
no propagation delay

Debugging turned on, not in Auto-Fix mode

... building Graphic Interface tool -- please wait ...

Now in the graphics command window, you might type these commands:

compile build
compile activation
combine build activation - total
loadcode total
call buildnet

Then after displaying the network on the graphics interface, and simulating for a while, you might decide you want to change one of the activation functions in activation.c. First make the changes. Then recompile and reload.

compile activation
loadcode activation

You can expect some warning messages about the previous versions of the functions in the total code unit. Suppose after further simulation you decide you want to rebuild the network with a slightly different structure. First edit the network building functions in build.c. Then the activation and build code units need to be recombined once more.
2.4. Memory Management commands

The simulator recovers freed space when appropriate. Space can be freed because links or sites were deleted, because the network is restarted, because a code unit is reloaded, or because names are deleted. Deleted links and sites are placed on a free list. Unless the scavenge command is given, or the simulator runs out of space, these free lists are not recovered, and in fact the free list of sites is never recovered. However the links and sites on the free lists are reused if more network construction takes place. When a network is restarted, all freed space is recovered. When a code unit is reloaded, freed space is recovered. Figure 2.2 illustrates three blocks of links chained forwards and backwards. Note that redefining a function is not enough to free up any space: the unit of code is an object file, not a function. The version which simulates link propagation delays (see Section 2.5) may also be able to recover space in the output buffers which is unused.

![Figure 2.2: Linked blocks of links](image)

If the simulator runs out of space in a controlled way, it first attempts to scavenge unused space on the free link list and in the name table. If there is still not enough space,
it requests the user to free up some space if possible. If that cannot be done, the core is dumped and the simulator exits.

2.4.12.1 Scavenge command

Syntax: scavenge
Example: scavenge

This command attempts to recover unused space. It can take a while if your network is large. First it recovers unused space in the nametable. Then it recovers links on the free list if possible. Finally, in the propagation delay simulator (see Section 2.5) it recovers unused space in the output buffers, if possible.

2.4.12.2 Scavenging functions

When links and sites are deleted or names become undefined, the space associated with them is not immediately recovered (see Section 2.4.12). It is possible to call the space recovery functions from within a user program (for example, immediately after calling one of the deletion commands). These functions are ScavengeLinks() which attempts to recover space from the list of free links (which can take some time), ScavengeNames() which attempts to recover unused space in the name table, and ScavengeBuffers() which shortens the output buffers in the propagation delay simulator (see Section 2.5) to the minimum length required. This last function may cause simulation time to be substantially speeded up.

2.4.13 Miscellaneous commands

2.4.13.1 Quit command

Syntax: quit

The quit command is used to exit the simulator (or to exit a higher level interface – see Section 2.4.3.2). When exiting to the UNIX shell, the simulator will ask if the file of commands typed during the session should be kept. If the answer is yes, it will ask for a file name. The convention is that command file names are of the form name.cmd.n. For example:

-> quit
Save command file (y,n) ? y
cmd file name [default: run2462.cmd] > conceptlearn.cmd.1
%
2.4. SIMULATION

From higher level interfaces, the *quit* command will return to the next lowest interface (if all errors have been fixed – see Section 2.4.3). Typing control-D to the prompt at any level interface will result in immediate exit to the UNIX shell (after possibly saving the command file).

2.4.13.2 Print command

**Syntax:** print <message>

**Example:** print finished reading in concepts

The *print* command simply prints the message it is given. For example:

```
-> print finished reading the message
finished reading the message
->
```

This command is intended for use in command files. When a command file is being *read*, the commands are not echoed on the screen. This command can be used to indicate significant stages in the simulation specified in the command file.

2.4.13.3 Printpause command

**Syntax:** printpause <message>

**Example:** printpause finished reading in concepts

The *printpause* command is a combination of the *print* and *pause* commands. It prints the message and then the simulator pauses for the user to indicate to continue. For example:

```
-> printpause finished reading in concepts
finished reading in concepts
PAUSE - any char to continue <user hits a key here>
->
```

Like the *print* command, this command is intended for use in command files. The command file can display interesting units, and then *printpause* while the user examines the data, before continuing. During the pause, the user can even type control-C, thus entering the interrupt interface (see Section 2.4.3.5) whence the network may be examined in detail, and when the interrupt interface is exited, the command file will continue to be read.
2.4.13.4 Whatis command

Syntax: whatis <name>
Example: whatis buildnet

This command reports what kind of name the given name is – whether that of a single unit, a vector of units, an array of units, a site, a unit type, a function, a variable, a code unit, an unused name, or of an unknown type.

2.4.13.5 Abort command

Syntax: abort
Example: abort

This command signals IOT, normally resulting in core being dumped and the program exited.

2.4.14 Abbreviated commands

Several commands have abbreviations:

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>disp</td>
</tr>
<tr>
<td>e</td>
<td>echo</td>
</tr>
<tr>
<td>g</td>
<td>go</td>
</tr>
<tr>
<td>l</td>
<td>list</td>
</tr>
<tr>
<td>o</td>
<td>out</td>
</tr>
<tr>
<td>p</td>
<td>pot</td>
</tr>
<tr>
<td>q</td>
<td>quit</td>
</tr>
<tr>
<td>s</td>
<td>state</td>
</tr>
<tr>
<td>sh</td>
<td>show</td>
</tr>
<tr>
<td>w</td>
<td>weight</td>
</tr>
</tbody>
</table>

In addition some of the terms used in commands have abbreviations:

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Term</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>all</td>
</tr>
<tr>
<td>c</td>
<td>connections (equivalent to link)</td>
</tr>
<tr>
<td>def</td>
<td>default</td>
</tr>
<tr>
<td>u</td>
<td>unit</td>
</tr>
</tbody>
</table>
2.4.15 Constructing a network from the command interface

Using the MakeUnit, AddSite, and MakeLink commands it is possible to construct small simple networks from the command interface. The first MakeUnit command will cause space for 200 units to be allocated, or the AllocateUnits command may be used to explicitly create the space. For example, the following script file makes a network of four units, linked in a ring, and names them as a vector.

```
MakeUnit mytype UFsum
MakeUnit mytype UFsum
MakeUnit mytype UFsum
MakeUnit mytype UFsum
AddSite 0 mysite SFweightedsum
AddSite 1 mysite SFweightedsum
AddSite 2 mysite SFweightedsum
AddSite 3 mysite SFweightedsum
MakeLink 0 1 mysite 1000 0 NULL
MakeLink 1 2 mysite 1000 0 NULL
MakeLink 2 3 mysite 1000 0 NULL
MakeLink 3 0 mysite 1000 0 NULL
NameUnit NOVICE vector 0 4
```

2.5 Simulating propagation delays

In real neural networks (i.e. nervous systems) propagation delay along fibers and within dendritic trees plays a significant role in computation. To make the simulator more useful to researchers in neuroscience, a facility to simulate propagation delay along links has been added. To use this facility the delay version of the simulator must be installed. This version will then be used by the makesim command when the -t flag is given. A new primitive (C function) is provided to specify links with an associated delay. Propagation delay times are specified as a number of simulation steps. Since it takes at least one simulation step to communicate between two units, the minimum delay is 1 step. The maximum is 255 steps.

2.5.1 Installing the delay simulator

This should be done automatically when the top level makefile(s) are executed. The delay versions are in lib and are called tsim.o and tsimf.o (floating point). If these are not present, the makefile in src/uniproc should be used to make them. make delay will make the integer delay version, make fdelay will make the floating point delay version.
2.5.2 Using the delay simulator

All existing network construction programs should work with the delay version – all links will have one step delay. However the network update will take longer. To make a propagation delay simulator, give the `-t` flag to the `makesim` command. Any C files you provide will be compiled with the symbol `TSIM` defined, so your code can contain compilation conditionals.

2.5.3 Delay command

Usage: `delay [<SourceUnitId> <DestinationUnitID> <site> <delay-value>]+`
Example: `delay all 0 - 40 excite 10`

The `delay` command is used to set the propagation delay on links from one or more units arriving at one or all sites on one or more units. It expects one or more unit-identifier/unit-identifier/site-name/delay-value quadruples. The unit identifiers may be specified in any of the usual ways (type “help UnitId” for information).

The first unit-identifier specifies the units at which the affected links originate. The second unit-identifier specifies the units at which the affected links arrive. The site name is that of a site on the destination unit(s), or “all” meaning all sites. The delay may be any integer value from 1 to 255. Note that increasing the delay on a link can take considerable time if your network is large.

When units are displayed or links listed, the delay associated with each link is displayed along with all other usual information.

2.5.4 MakeDelayLink primitive

The `MakeLink` primitive will make a link with delay one. The new `MakeDelayLink` will make a link with any delay from one to 255 time steps. `MakeDelayLink` has the following specification:

```c
Link * MakeDelayLink(source,destination,site,weight,data,function,delay)
    int source,destination;
    int weight, data, delay;
    char *site;
    func_ptr function;
```

which is exactly the same as `MakeLink` (see Section 2.2.4), except that there is an extra parameter, `delay` which specifies the delay on the link. `MakeDelayLink` returns a pointer to the link structure created. The time unit is one simulation step, so a delay of 15 means a delay of 15 simulation steps.
2.5. SIMULATING PROPAGATION DELAYS

2.5.5 MakeDelayUnit primitive

When making units it is advisable to use the new primitive MakeDelayUnit which has the following specification:

```c
int MakeDelayUnit(type, func, init_pot, pot, data, output, init_state, state, delay)
    char *type;
    func_ptr func;
    int init_state, state, init_pot, potential, output, data, delay;
```

This function is just like the usual MakeUnit except that it has an extra parameter delay which specifies the maximum delay on links arriving at the unit. Using the usual MakeUnit primitive is not incorrect, but will usually result in the network taking a much longer time to build. The reason for this is that each unit has a buffer of output values along which the outputs are pipelined. If a new link is specified to arrive at the unit with a delay longer than the current buffer, then a longer buffer is made and the complete set of links searched to remap any that pointed to the old buffer. This search of the links can take a considerable amount of time. If the maximum length of the buffer is given with the MakeDelayUnit primitive when the unit is first made, then this search should never take place. It is not essential that the delay value given to MakeDelayUnit be correct: if it is too large, some space is wasted and simulation could be slowed down a little; if it is too small, the set of links will be searched, possibly many times, wasting time during network construction. The buffers will be shortened to the smallest length required if you use the scavenge command (see Section 2.4.12.1).

2.5.6 SetDelay and GetDelay primitives

These primitives set or return the delay associated with a link. SetDelay can take considerable time if the delay buffer has to be extended. They have the following specifications:

```c
Link * SetDelay(from, to, site, delay)
    int from, to, delay;
    char * site;

int GetDelay(from, to, site)
    int from, to, delay;
    char * site;
```

Both take the source unit index, destination unit index, and destination site name as parameters. SetDelay also takes the new delay as a parameter, and returns a pointer to the link structure. GetDelay returns the current value associated with the link. It is assumed as elsewhere in the simulator that there is at most one link from any given unit to a particular site on another unit, so that source index, destination index and destination site specify a link uniquely.
2.5.7 ChangeDelayBuffer primitive

ChangeDelayBuffer(from, delay)
    int from, delay;

This function changes the size of the buffer associated with unit \textit{from} to the new value \textit{delay}. The delay must be in the range 1 to 255. No action is taken if the buffer has the requested size already.

2.6 Advanced Programming Features

We have tried to ensure that the functions and facilities described in the preceding sections cannot corrupt the network structure, or cause the simulator to dump core, even if misused. However, such security imposes limitations that experienced users will find irksome. Consequently the simulator has purposely been designed so that an advanced programmer is limited as little as possible. This means that all internal data structures apart from the Name Table are available to user code. The following sections give an overview of unsecured features, which are described in detail in the Advanced Programming Manual. Details of features that are commonly used in user code appear here.

2.6.1 Simulator Variables

Many variables used by the simulator are accessible to user code. Modifying these should be done with care. The variables which are often used in user programs are shown in Table 2.1. The use of these variables is documented in the Advanced Programming Manual.

2.6.2 Network Access Functions

There are a large number of access functions and macros that allow user code to retrieve and set values in the network data structure. The following subsections describe the kinds of facilities available, and detail functions and macros which are commonly used in user code. A much more complete description is given in the Advanced Programming Manual.

2.6.2.1 Display functions

The simulator functions used to display, list and show units and links can be called from user code. The piping mechanism through which these functions usually print is also available.
2.6. ADVANCED PROGRAMMING FEATURES

Table 2.1: User accessible simulator variables

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unit * UnitList</td>
<td>pointer to the unit array</td>
</tr>
<tr>
<td>int NoUnits;</td>
<td>number of units made so far</td>
</tr>
<tr>
<td>int LastUnit;</td>
<td>index of last unit that there is space for</td>
</tr>
<tr>
<td>char ** StateNames;</td>
<td>array of state names, indexed by state value</td>
</tr>
<tr>
<td>int NoStates;</td>
<td>maximum number of states with names</td>
</tr>
<tr>
<td>int StateCount;</td>
<td>actual number of states with names</td>
</tr>
<tr>
<td>char ** SetNames;</td>
<td>array of set names, indexed by set number</td>
</tr>
<tr>
<td>int LastSet;</td>
<td>maximum number of sets allowed (currently 32)</td>
</tr>
<tr>
<td>int NoSets;</td>
<td>actual number of sets used</td>
</tr>
<tr>
<td>int NoLinks;</td>
<td>number of links made so far</td>
</tr>
<tr>
<td>int Clock;</td>
<td>the system clock</td>
</tr>
<tr>
<td>int Pause;</td>
<td>if TRUE, Pause after every Show</td>
</tr>
<tr>
<td>int EchoStep;</td>
<td>print message after this number of steps</td>
</tr>
<tr>
<td>int Logging;</td>
<td>TRUE if a log file is being created</td>
</tr>
<tr>
<td>FILE * LogFile;</td>
<td>file pointer for Log file</td>
</tr>
<tr>
<td>int Show;</td>
<td>TRUE if showing is switched on</td>
</tr>
<tr>
<td>int ShowStep;</td>
<td>display units at this number of steps</td>
</tr>
<tr>
<td>int ShowPot;</td>
<td>display units with potential over this limit</td>
</tr>
<tr>
<td>unsigned int ShowSets;</td>
<td>bit vector for sets to be shown</td>
</tr>
</tbody>
</table>

2.6.2.2 Naming

The Name Table is available for user code access, to add, delete, and modify names. In addition to the basic functions which are used to access the Name Table, the following functions are common in user code, particularly activation functions.

char * IndToName(u)
int u;

returns the name of the unit with index u, or **NO NAME** if the unit has not been given a name. If the name is that of a VECTOR or ARRAY, the name has the form name[offset] or name[row][column].

int NameToInd(name, column, row)
char * name;
int column, row;

returns the index of the unit with the given name. If the name is that of a VECTOR, then column gives offset of the unit within the vector. If the name is that of an ARRAY, then column and row give the column and row of the unit within the array. If the name is not that of a unit, or either of the indices are out of range, then the function returns -1.

2.6.2.3 Simulating

Several simulation functions which closely correspond to simulation interface commands can be called from user code.

\texttt{Reset()}

resets the network: sets the system Clock to zero; sets the potential and state of each unit to \texttt{init.potential} and \texttt{init.state} respectively; sets the output of each unit to zero.

\texttt{Step(count)}

\begin{verbatim}
int count;
\end{verbatim}

simulates \texttt{count} steps. Echoes and shows will be done if appropriate.

2.6.2.4 Modifying and Accessing the Network

There are a large number of functions which can be used to modify values in the network data structure. The ones often used are:

\texttt{SetOutput(index, value)}
\texttt{SetPotential(index, value)}
\texttt{SetState(index, value)}
\texttt{SetData(index, value)}

\begin{verbatim}
int index, value;
\end{verbatim}

Unit \texttt{index} is given output, potential, state, or data \texttt{value}.

\texttt{int GetOutput(index)}
\texttt{int GetPotential(index)}
\texttt{int GetState(index)}
\texttt{int GetData(index)}

\begin{verbatim}
int index;
\end{verbatim}

Output, potential, state or data of unit \texttt{index} is returned.

2.6.2.5 Unit macros

These macros access the deal with unit indices and pointers.
2.6. ADVANCED PROGRAMMING FEATURES

LegalUnit(index)
    int index;

computes TRUE if index is the index of an existing unit, FALSE otherwise.

UnitIndex(up)
    Unit * up;

computes the index of the unit pointed to by up. If up does not point to a unit, computes garbage.

2.6.2.6 Miscellaneous library functions

The following functions are in the standard simulator library.

SiteValue(name, sp)
    char * name;
    Site * sp;

If sp is a pointer to a linked list of site structures, such as in the sites field of the Unit structure, and name is the name of one of the sites in the linked list, then the function returns the value of that site. If no such site is found, 0 is returned and an error message printed.

2.6.3 Saving and reloading user data structures

The user may wish to create data structures separate from the main network data structure. Hooks are provided in the simulator to enable user-written functions to save and reload these structures when the save, checkpoint, load and restore commands are issued.

2.6.4 Customizing unit, site and link data structures

Each unit, site and link structure contains a field, data, which is for general purpose use. This field is the size of an integer or float, depending on which simulator is being used, but in any case is assumed to be the same size as a pointer. Therefore it is possible to use this field as a pointer to an arbitrary user-defined data structure. A mechanism is provided for the field to be re-defined for user code, and hooks are provided in the simulator code to call user functions to deal with displaying, saving, and loading these user-created structures.
2.6.5 Customizing the simulator command interface

Any user written function which has a name commencing "Cmd." will be treated as a regular command by the simulator, and will be available at all interfaces. The simulator passes an argc-argv structure to command functions. In addition, any command commencing "Debug.Cmd." will be available at the debug and interrupt interfaces (see Sections 2.4.3.2 and 2.4.3.5), and will take precedence over level 0 commands of the same name.

2.6.6 Modeling Time

One aspect of neural modeling that the simulator was not designed to deal with is modeling time, for instance modeling propagation delays along fibers. By customizing the network data structure as outlined above, it is possible to adapt the simulator to model such parameters, albeit in a somewhat limited fashion. An example is given in the Advanced Programming Manual (Chapter 3).

2.7 An Extended Example of network construction

In this section we provide an examples of a relatively complex network in order to introduce some of the ways in which networks may be constructed. This network is one of the examples found in the example subdirectory (usually "example, but this is site-dependent). Read the README file before running it.

2.7.1 Designing a network

The process of creating a connection network can be broken down into three related stages. First a design for the network in terms of units, links and activation functions (unit, site and link) is specified. Second a program is designed to build this network. Third the program is coded in C. We present these three stages for the Four-Color problem. If one is building a network to model some process specified at a higher level of description (e.g. a model of a cognitive process), the higher level specification must be made before a network to implement it can be designed.

2.7.2 The Problem: four coloring a map

The problem is to color a map using four colors so that no neighboring regions have the same color. The colors are RED, BLUE, GREEN and WHITE. We shall represent each region with four units, one for each color. The dynamics of the network are simple. Each region wants to turn on exactly one color node. If some region is shut out by neighbors' colors it may turn on some color anyway, which may force off some other region's color node. The general idea is for each region's units to inhibit each other (corresponding to the
2.7. AN EXTENDED EXAMPLE OF NETWORK CONSTRUCTION

notion that a region can have at most one color), and for neighboring regions to inhibit each other from having the same color. We shall use asynchronous simulation to ensure that the search space is explored until a stable state corresponding to a correct coloring is found.

We have four units for each region. What exactly will the links be? Our units will have one site - “inhibit”. Each region’s units must inhibit each other – we shall accomplish this by making an inhibitory link from each unit to each of the other units in that region. Thus each region will require twelve links internally – three for each of the four units. Since it is a very strong constraint that a region have only one color, the weight on these links will be highly negative (high inhibition).

How shall we accomplish inter-region inhibition? No two neighboring regions can have the same color, so if two regions have a border, we make an inhibitory link from the color units in one region to the corresponding units in the other. For instance, if region X borders region Y, we add inhibitory links from region X’s blue unit to region Y’s blue unit, and vice versa. The same applies for the red, green and white units. Since we wish the network to search the space of possible colorings we must allow neighboring regions to have the same color for a short period of time, so the weight on these links will be moderate and negative (moderate inhibition).

How do we control the search through the state space? First we will say that if a unit is receiving no inhibition, it will turn on. If there is strong inhibition (e.g. from another color in the same region), the unit will remain off. If there is weak inhibition (i.e. from bordering region(s) of the same color, the unit will turn on or remain on with a probability which decreases with the amount of inhibition.

Finally, exactly what activation functions shall we use to implement the descriptions above? Since the weights are not changed, link functions can be the NULL function. The site function will simply take the weighted sum of the inputs. What weights shall we use? For the intra-region inhibition the maximum negative weight of -1000. For the inter-region same-color inhibition we shall use -100. The unit function will simply look at the inhibition arriving. If there is none, i.e. the region is not yet colored and no neighboring region is colored with the same color, then the unit will turn on with activation and output 1000. If there is inhibition from neighboring countries (in the 100’s) but not from units in the same region (in the 1000’s) then with some small probability dependent on the strength of the inhibition, the unit will turn on with activation and output 1000. If there is inhibition from another unit in the same region (i.e. it is already colored) then the unit will turn off with activation and output 0.

2.7.3 Designing the build program

There are several things to note about the network design. The map of the regions is not specified explicitly. All links are bi-directional, i.e. if unit X inhibits unit Y then unit Y inhibits unit X with both links having the same weight. The links can be divided into two kinds, intra-region (twelve for each region), and inter-region (four for each border). These suggest some design decisions.
First we shall specify the map using a data file, which will be read in by the build program. The map file format will be:

```
<number of regions>
<region number><region number>
<region number><region number>
...
<region number><region number>
```

The first item specifies the number of regions, the remaining pairs specify the pairs of regions which border each other (each region is assumed to have been assigned a number). Thus the data file:

```
3
1 2
3 2
```

would specify a three region map with one region sandwiched between the other two, as in El Salvador, Honduras, Nicaragua. The map will be read in by the top level function, build.

Since the links are effectively bi-directional, we shall have a function `mux` (mutual exclusion) which creates two links, one in each direction. Since each region has the same internal structure, we shall have a function `region` to create a region. Since each border consists of mutual exclusion links between corresponding color units for the two regions, we shall have function `border` to create these links.

How shall we assign the units? Our method will be to assign them in blocks of four, one block for each region. Within each block, the first unit will be for RED, the second BLUE, the third GREEN, and the fourth WHITE. For example unit number 6 (the seventh unit as unit numbers start at 0) will be the GREEN unit for the second region.

Thus the overall mechanism for building the network will be to read in the number of regions, allocate sufficient units accordingly, make the specified number of regions (with `region`), and then read in each border and make it (with `border`).

The site for each unit will have flag NO_LINK_FUNC_FLAG set to indicate that no link functions are operative. The site function will simply be the library function SFweightedsum. The unit function will operate as described above, with the addition that it will add the unit to a set, change if the potential changes, and remove it if it doesn't. While simulating we may then use the show command to cause just the units which have changed since the last step to be displayed. In addition we will have two named states, Change and Static, which will correspond to membership or not of the set change respectively.

### 2.7.4 Implementing in C

Now that we have the design, the build program can be coded.
2.7.4.1 Top level build function

At the outermost level, the program appears in Table 2.2. The first two lines allow us to use the Unix input/output facilities, and the simulator functions respectively. These lines will be present in every build program. The next two lines define constants STATIC and CHANGE to have values 0 and 1 respectively. The fifth line declares the unit function which is defined later. Finally the definition of the function build is given. This function is called from the simulator interface with the name of a data file as the only argument. The first if statement checks that indeed only one argument to build has been given. The second if statement checks that the argument is indeed a file that the program can read. If either of these checks fail, the function return's to the simulator with an error message.

Assuming all is ok so far, the number of regions (the first item in the data file) is read into variable count. Each region requires four units so count*4 units are allocated with a call to AllocateUnits. The set we wish to use for display, change is declared, and the state names “Static” and “Change” are bound to the values “STATIC” and “CHANGE”. If this seems confusing, remember that the units keep their states as a number, e.g. STATIC == 0. By binding a name to a state value, using DeclareState, the simulator will display the name rather than the number.

Finally we are ready to do the real work of building the network. The first for statement makes the appropriate number of regions by successive calls to the function region. The second for statement repeatedly reads a border specification from the map file and makes it via a call to the function border, until the end of the map file is found. Once all this is accomplished, the network is in place and the user is informed how many regions and borders were made.
CHAPTER 2. SIMULATOR USER MANUAL

```c
#include "sim.h" /* simulator definitions */
#define STATIC 0 /* constant state names */
#define CHANGE 1

int UFcolor(); /* unit function declarations */

/* build is the topmost network building function. It is called from
the simulator to build the network */

build(argc,argv)
int argc; /* number of arguments */
char *argv[]; /* array of argument strings */
{
    int count,i,reg1,reg2,readstatus;
    FILE *infile;
    
    if(argc != 2) /* expect build + 1 argument */
    {
        printf("Usage: build <desc file name>\n");
        return;
    }

    infile = fopen(argv[1],"r"); /* open data file */
    if(infile == NULL) /* if cannot open file */
    {
        printf("build: could not open %s\n", argv[1]);
        return;
    }

    fscanf(infile,"%d",&count); /* read number of regions from file */
    AllocateUnits(count*4); /* make units for regions X 4 colors */

    DeclareSet("change"); /* declare a set name */
    DeclareState("Static",STATIC); /* state name declaration */
    DeclareState("Change",CHANGE); /* state name declaration */

    for(i = 0;i < count;i++) /* make the regions */
    {
        region();
    }

    for(i=0;i++ ) /* make the borders */
    {
        /* exit loop with break statement */
        readstatus = fscanf(infile,"%d %d",&reg1,&reg2);/* read two regions */
        if(readstatus != 2) break; /* assume end of data - leave loop */
        border(reg1,reg2); /* make the border */
    }
    printf("%d regions with %d borders",count,i);
}
```

Table 2.2: Example code for a build function
2.7. AN EXTENDED EXAMPLE OF NETWORK CONSTRUCTION

2.7.4.2 Making a region

The code to make a region is given in Table 2.3. The first few lines are to define values for our colors. These will be used to index into the string array colornames, so they take values 0, 1, 2 and 3 in the same order as in colornames. Now we come to the code for the function region. It is declared static to ensure that when loading multiple object files this function is not visible to any other object file (consult a C manual if this is not clear). The first line in the function declares an integer regionnum, which is also static. The effect of this version of static is to make regionnum a permanent variable, so that it retains its value between calls to the function. At each call to the function it is incremented, so it represents the number of the region that is currently being made. The first region will have number zero.

The task of making the units for the region is done by the four calls to makecolor. Each call makes one unit corresponding to one color for the region, and because the calls are consecutive, the four units are next to each other in the unit array. We name them using the Name Unit simulator function. Finally the mutually exclusive links between the region’s units are made by calls to mux. The weight on the links is set to -1000, as specified in the design. We return the number of the region just made, and increment regionnum before exiting.

Next we describe the code for the functions makecolor and mux which is shown in Table 2.4. The function makecolor makes a unit (MakeUnit), adds a site called “inhibit” to it (AddSite), sets the no link function flag, and returns the index of the unit made. Notice that the type of the unit is taken from the string vector colornames according to the color being made, and that the unit function is called UFcolor – this will be described below. The initial and reset state for the unit is STATIC. Also notice that the site function is SFweightedsum, a library function.

The function mux simply makes two links, one in either direction, between the two units, each link going to site “inhibit” and having the same weight.

2.7.4.3 Making borders

Making a border is a simple matter of creating mutually inhibitory links between the region units representing the same color. The function mapunit is a utility to get the unit index from the region number and unit color.

The border is actually made by function border. This calls mux to create the four pairs of links (one pair for each color) using mapunit to get the unit indices. The code appears in Table 2.5.

There are faster ways of finding the index of a unit than the name lookup used by mapunit. For example, the expression 4*region+color gives the correct index. However, if we decide to add new units to the network, the expression might be invalidated, but the name lookup will still work. This is an important general principle: if you wish your network
/* Make a region. Four units are made - one for each color. The entire region is named as an array, with the color macros below specifying the appropriate indices. They inhibit each other strongly. */

#define RED 0
#define BLUE 1
#define GREEN 2
#define WHITE 3
static char *colornames[] = {"red","blue","green","white"};

static int region()
{
    static int regionnum = 0; /* which region - static means value remains between calls; it is initialized at startup to 0 */
    int i,j,first;
    char buf[16];

    /* make 4 consecutive units, each for a different color */
    first = makecolor(RED); /* save index for name declaration*/
    makecolor(BLUE);
    makecolor(GREEN);
    makecolor(WHITE);

    sprintf(buf,"region%1d",regionnum); /* buf contains name */
    NameUnit(buf,VECTOR,first,4); /* vector of length 4 */

    for(i = 0;i < 4;i++) /* make them inhibit each other */
        for(j = i+1;j < 4;j++)
            mux(first+i,first+j,-1000);
    return regionnum++; /* returns region number */
}

Table 2.3: Example code for making a region
2.7. AN EXTENDED EXAMPLE OF NETWORK CONSTRUCTION

```c
/* make a single unit representing a region color; return index of that unit */
static int makecolor(type)
    int type;        /* a string containing color name */
{
    int index;

    index = MakeUnit(colornames[type],UFcolor,0,0,0,STATIC,STATIC);
    AddSite(index,"inhibit",SFweightedsum);
    SetFlag(index,NO_LINK_FUNC_FLAG);    /* weights don't change */
    return index;
}

static mux(unit1,unit2,weight)      /* mutually exclusive links */
    int unit1,unit2,weight;
{
    MakeLink(unit1,unit2,"inhibit",weight,0,NUL);
    MakeLink(unit2,unit1,"inhibit",weight,0,NUL);
}
```

Table 2.4: Example code for makecolor and mux functions

to be modifiable, use the simulator functions to access data items, rather than finding them with your own code. In this case, with small maps, using the simulator functions produces no noticeable delay.

No link functions are required because weights don't change. The NO_LINK_FUNC_FLAG can also be set in the units flags. If it wasn't, a default, empty link function would be called for every link, for each step of simulation. This does no harm, but does take some time.

2.7.4.4 The unit function

The unit function UFcolor performs the function described in the design. Color units are either on (potential and output equal to 1000) or off (0). If there is no inhibition or dice returns true a unit will turn on. The macro dice returns true with a probability dependent on the amount of inhibition the unit is receiving, and is used to determine whether or not to switch the unit on. If the inhibition is over 1000 (another color unit in the same region is on) then dice returns false. If the inhibition is under 1000 (corresponding color unit(s) in neighboring region(s) are on) then the unit will turn on or remain on with probability approximately (10 - #conflicts)/20. The code in shown in Table 2.6. The function UFcolor defines the behavior of the color nodes, and thus, of the whole network.

Units are in one of two states: STATIC and CHANGE. The state does not affect their
/* mapunit takes a region number and color and returns that unit's index */
static mapunit(region,color)
    int region,color;
{
    char buf[15];
sprintf(buf,"region%ld",region); /*get name of region in buf */
    return NameToInd(buf,color); /*look up index */
}

/* make a border between region1 and region2 */
static border(region1,region2)
    int region1,region2;
{
    int i;

    mux(mapunit(region1,BLUE),mapunit(region2,BLUE),-100);
    mux(mapunit(region1,RED),mapunit(region2,RED),-100);
    mux(mapunit(region1,GREEN),mapunit(region2,GREEN),-100);
    mux(mapunit(region1,WHITE),mapunit(region2,WHITE),-100);
}

Table 2.5: Example code for making borders

behavior, but it does make it easier to see what is happening when watching unit lists scroll by. Several steps in a row with no units in a change state probably means the network has settled on a solution.

UFcolor also dynamically updates membership in the set "change" so that it contains units which have changed state this step. This set is used as a show set, so units which have just changed will always participate in a show.

2.7.5 A command script to demonstrate the network

An important way of controlling the simulation is the use of command scripts. The following script has all the commands necessary to start and run the four color network.
2.7. AN EXTENDED EXAMPLE OF NETWORK CONSTRUCTION

```c
/*
dice(inhibit) has value 1 with probability: \((1000+inhibit)/1999\);
Note that if inhibit is <= -1000, this probability is zero.
*/
#define dice(inhibit) ((random()%1999) < (1000 + inhibit))

UFCcolor(up)
  Unit *up;
{
  int inhibit;
  int oldpot;

  oldpot = up->potential;    /* remember old potential */
inhibit = SiteValue("inhibit",up->sites);
if(inhibit >= 0 || dice(inhibit))
{
  /* unit on */
  up->potential = 1000;
  up->output = 1000;
}
else
{
  /* unit off */
  up->potential = 0;
  up->output = 0;
}

/* change state and set membership if necessary */
if(oldpot != up->potential)
{
  AddSet("change",UnitIndex(up));
  up->state = CHANGE;
}
else
{
  RemSet("change",UnitIndex(up));
  up->state = STATIC;
}
}
```

Table 2.6: Example code for unit function
The first command calls the function *build* with map file *map*. This causes the network to be constructed. The three show commands together mean that we will see only those units which are on or have just gone off. Turning the pipe off means that output to the terminal will not go through the "more" filter. Turning pause on causes the simulator to wait for a prompt after each step. The simulation must be run asynchronously or the whole network will oscillate: all on, all off, all on, ... The *async* command puts the simulator in asynchronous mode with all units simulated at each step. The *printpause* command prints the message and waits for a prompt. The final command tells the simulator to run for 10 steps.

Command scripts can considerably ease the burden of simulating. It is easy to create script files with slight variations in the simulation parameters and control. Feeding in a lot of information to a network can be done as well, for instance for initialization.

### 2.8 Floating Point version

The default simulator that is built by *makesim* when no flags are specified is the *integer* version, in which all unit, site and link values are stored, manipulated and displayed as integers. The primary purpose of this is speed: for computers without hardware floating point support, this integer version will run simulations many times faster than if floating point operations are used. However this requires special attention by you when coding unit, site and link activation functions. Also the granularity provided may not be sufficient for your purposes. However if you have floating point support or wish to use floating point despite the performance consequences, support is provided to create a floating point version of the simulator.

The floating point version of the simulator uses floating point values for *potentials*, *outputs*, *site values*, *weights*, *link values*, and unit, site and link *data* fields. The floating point version is created by specifying the `-f` flag to *makesim* (see Section 2.3). If this flag is given, user code will be compiled with the flag `-DFSIM` (see the man page for the C compiler, *cc*, for details of the `-D` flag). Thus user code may use conditional compilation to automatically be recompiled for integer/floating point simulation. The type *FLINT* is defined in the user compilation environment to be a float or an integer depending on whether
the compilation is for the floating point or integer simulator. If you are compiling with the
Graphics Interface, using the -f flag will automatically include the floating point version of
the Graphics Interface as well.

If you will be running the floating point version of the simulator on a machine that has
hardware support for floating point, you should make sure that the C compiler knows this
when compiling the simulator sources and user code. The best way to do this in the Unix
environment is to set the environment variable `FLOAT_OPTION` according to the hardware
your system has; for example `-ffpu` for a floating point accelerator or `-f68881` for a floating
point co-processor. If the `FLOAT_OPTION` is not set, the floating point will be done in
software. You should contact your system administrator for advice if you are unsure of
what your system has and how to specify it to the C compiler.
Chapter 3

Advanced Programming Manual

3.1 Introduction

This manual assumes familiarity with the operation of the simulator as described in Chapter 2. The multitude of simulator functions that can be called from user code are described. Examples of customizing the data structures and command interface are given. The sample networks in the example subdirectory use some of the facilities described here. Almost all the functions are used by the simulator itself, so examining the simulator source code will reveal further uses.

3.2 The Network Data Structure

The network data structure is the heart of the simulator. It is the basis of the representation of units, sites and links. Simply put, it is an array of Unit structures, with a linked list of Site structures attached to each Unit structure, and a linked list of Link structures attached to each Site structure. These structure definitions described in Table 3.1.

The complex structure is to allow various fields to be either a short integer, as above, or a float for the floating point simulator (see Section 3.22). The data fields may be redefined by the user to extend the network data structure (see Section 3.18).

3.2.1 Creating space for units

Before any units can be made, you should try to specify the total number of units in the network that will be needed. This is done through the AllocateUnits function:

```c
AllocateUnits(number)
    int number;
```
typedef short weight_type;
typedef int data_type;
typedef int func_type;
typedef short pot_type;
typedef short Output;

typedef func_type (* func_ptr)();
typedef data_type link_data_type;
typedef data_type site_data_type;
typedef data_type unit_data_type;

typedef struct link
{
    func_ptr link_f;
    weight_type weight;
    Output * value;
    link_data_type data;
    int from_unit;
    struct link *next;
} Link;

typedef struct site
{
    char * name;
    Output value;
    short no_inputs;
    site_data_type data;
    func_ptr site_f;
    Link * inputs;
    struct site *next;
} Site;

typedef struct unit
{
    unsigned int flags;
    char * type;
    func_ptr unit_f;
    char * name;
    pot_type init_potential;
    pot_type potential;
    Output output;
    short init_state;
    short state;
    short no_site;
    unit_data_type data;
    unsigned int sets;
    Site * sites;
} Unit;

Table 3.1: Simulator network data structure
3.2. THE NETWORK DATA STRUCTURE

If *AllocateUnits* has not been called before the first call to *MakeUnit*, then 1000 units are automatically allocated. If you try to construct more units than have been allocated, the simulator allocates a new and larger unit array, copies the already made units into the new array, and deletes the old array. The extra number of units allocated whenever the simulator runs out of allocated units is the same as the most recent call to *AllocateUnits*. You may call *AllocateUnits* as many times as necessary. Each call increments the number of units allocated by the argument to *AllocateUnits*. Thus if you initially call *AllocateUnits*(2000) you will get 2000 units. If later you call *AllocateUnits*(100) you will now have 2100 units allocated. If later on you try to make units past 2100, the simulator will increase the number of allocated units in increments of 100.

3.2.2 Making units

Units may be made with a call to *MakeUnit*. This function builds a new unit, using space allocated by *AllocateUnits*, for example:

```c
int MakeUnit(type, func, ipot, potential, data, output, istate, state)
    char *type;
    func_ptr func;
    int istate, state, ipot, potential, output, data;
```

*type* is a pointer to a character string, and is simply used for display purposes. *func* is a pointer to the function used to simulate the unit's action. *potential* is the activation level for the unit. *data* is a four byte value for the unit data field described above. *output* is the initial output of the unit. *state* is a short integer representing the initial state value. *ipot* and *istate* are the values to set the unit potential and state when the network is reset. *MakeUnit* returns the index in the unit array of the unit created. The first call to *MakeUnit* builds the unit with index 0, and consecutive calls to *MakeUnit* will return consecutive indices. An example of a call to *MakeUnit* would be:

```c
unit_index = MakeUnit("retinal", UFsum, 500, 500, 0, 50, 1, 1);
```

The function pointer may be NULL, in which case a function which does nothing will be called by the simulator to simulate the unit action. If not NULL, the function must be either one you have written, or one of the library functions. For simple networks the library functions (see Section 3.7) should be sufficient.

3.2.3 Adding sites

Once a unit has been created, one or more sites may be attached to it with calls to *AddSite*:
Site * AddSite(index, name, func, data)

    int unit, data;
    char *name;
    func_ptr func;

`index` is the index of the unit to which the site is to be attached. `name` is a pointer to a character string which will be the name of the site. `func` is a pointer to the function to be called to simulate the action of the site. `data` is the four byte value to be placed in the site `data` field described above.

Links to the unit cannot be made until there is a site attached to the unit to which they may go. A call to AddSite might look like:

    AddSite(unit_index,"excite",SFweightedsum,0);

AddSite returns a pointer to the newly created site structure. As with units, the function may be NULL, one of your functions, or one of the library functions.

### 3.2.4 Making links

A link from a unit to a site on another unit is created with a call to MakeLink:

Link * MakeLink(from,to,site,weight,data,func)

    int from,to;
    int weight, data;
    char *site;
    func_ptr func;

`from` is the index of the unit where the link originates. `to` is the index of the unit to which the link is going. `site` is a pointer to a character string which is the name of the site on the destination unit at which the link is to arrive. `weight` is the weight to put on the link, and should be within range of a short integer. By convention weights are scaled down by a factor of 1000, thus a specified weight of 500 will be treated as a weight of 0.5. This is to allow weights in the range 0 to 1 without having to use floating point arithmetic. Weights may be negative. MakeLink returns a pointer to the link structure created. An example of a call to MakeLink might be:

    MakeLink(unit_index, unit_index, "excite", -500, 0, LFsimple);

This would make a link from the unit to itself, to be attached at the site "excite", with a weight of -500 (meaning -0.5), and function LFsimple. Such a link could be used to provide exponential decay. As with units, the function may be NULL, one of your functions, or one of the library functions.
3.2.5 Deleting sites and links

Several functions exist for removing links and sites from an existing network. These functions may be called from within user programs. They are:

```c
Site * DeleteSite(index,sitename)
    int index; /* delete one site */
    char * sitename;

DeleteSites(ulow,uhigh,uset,usetind,sitename)
    int ulow,uhigh,uset,usetind; /* delete many sites */
    char * sitename;

Link * DeleteLink(from,to,sitename)
    int from, to; /* delete one link */
    char * sitename;

DeleteLinks(ul,uh,us,usind,ulow,uhigh,uset,usetind,sitename)
    int ul,uh,us,usind,ulow,uhigh,uset,usetind;
    char * sitename; /* delete many links */
```

`DeleteSite(index,name)` deletes the site named `name` on unit with index `index`. It also deletes all links arriving at that site. A pointer to the deleted site is returned. This is a valid pointer which may be used until the next call to `AddSite`. In particular, if any user defined data structure is associated with the `data` field of the site, this pointer can be used to free up the space allocated for that structure. This function looks for a function called `User_Link_Delete`. If it is found, it is called for each link deleted, with the pointer to the unit, the pointer to the site, and the pointer to the link as parameters. This function should be written by the user to free up space used for a user defined data structure associated with the link `data` field, if any.

`DeleteSites(ulow,uhigh,uset,usetind,sitename)` deletes all sites in a group of units with name `sitename` and all links arriving at that site. The group of units is all units in the range `ulow` to `uhigh`, and if `uset` is TRUE then only those units in the range that are in the set with index `usetind`. In addition to looking for and calling `User_Link_Delete` (as in `DeleteSite` above), this function looks for and (if found) calls `User_Site_Delete` to free up any structure associated with each site `data` field. This function should be written by the user if necessary; it can expect the pointer to the unit and the pointer to the site as arguments, in that order.

`DeleteLink(from,to,sitename)` deletes a single link from unit with index `from` to site with name `sitename` on unit with index `to`. It returns a pointer to the deleted structure which is valid until the next call to `MakeLink` or `MakeDelayLink` or `ScavengeLinks` (see below). This pointer should be used to delete any user allocated structure associated with the link `data` field.
DeleteLinks(ul, uh, us, usind, ulow, uhigh, uset, usetind, sitename) deletes many links. The first four parameters specify the group of units where the links originate, in a similar manner to the specification of a unit group for DeleteSites. The second four parameters specify the destination unit group, and the last parameter the name of the site at which the links arrive, or “all” meaning all sites. This function looks for and (if found) calls User_DeleteLink for every link deleted.

3.2.6 Deleting the network

The function RestartNetwork can be called from a user program to completely destroy the current network, returning all network space to the system. You can then build a new network from scratch.

RestartNetwork()

RestartNetwork() only resets the simulator data structures to their original state. If you wish to restart both the simulator and the Graphics Interface, or just the Graphics Interface, use gi_command("restart") or gi_command("restart gi"), respectively. See Section 4.11.2 for more information on the gi_command interface

3.3 Naming

The Name Table access functions are described in section 3.17. These functions provide additional possibilities.

3.3.1 Unit names

NameUnit(name, type, index, length, depth)
char *name;
int type, index, length, depth;

As well as naming a single unit, this function can name a vector or 2-D array of units. The name may then be used during simulation from the command interface, and may also be used during network construction. name is a pointer to the character string name to be given. type is the type of name - SCALAR, VECTOR, or ARRAY. index is the index of the unit to be named, or the first unit in the vector or array. length is the number of units if it is a VECTOR, and the number of columns if it is an ARRAY, and is undefined for SCALAR. depth is the number of rows for an ARRAY, and is undefined for SCALAR and VECTOR.
3.3. NAMING

```c
char * IndToName(u)
    int u;

Returns a pointer to a volatile string containing name of the unit with index u, or **NO NAME** if the unit has not been given a name. If the name is that of a VECTOR or ARRAY, the name has the form name[offset] or name[row][column].

```c
int NameToInd(name, column, row)
    char * name;
    int column, row;

Returns the index of the unit with the given name. If the name is that of a VECTOR, then column gives offset of the unit within the vector. If the name is that of an ARRAY, then column and row give the column and row of the unit within the array. If the name is not that of a unit, or either of the indices are out of range, then the function returns -1.

3.3.2 Function names

```c
char * FuncToName(function_pointer)
    func_ptr function_pointer;

If function_pointer is a pointer to a user function, e.g. a unit function, or to a simulator command function, then FuncToName returns a pointer to the name of the function, otherwise NULL.

```c
func_ptr NameToFunc(name)
    char * name;

If name is the name of a user function, or a simulator command function, then NameToFunc returns a pointer to the function, otherwise NULL.

```c
char * IndToFuncName (index)
    int index;

Passed and index into the function table, returns the pointer to that function's name, or NULL if the index is out of range.
3.3.3 Set and State names

```c
void DeclareState(char *name, int num);
```

associates a name with a state number. num must be in the range 0 to 99.

```c
int NameToSet(char *name);
```

```c
int NameToState(char *name);
```

passed a set/state name, returns the set/state number, or -1 if the name is not that of a set/state.

```c
char *SetToName(int number);
```

```c
char *StateToName(int number);
```

passed a set/state number, returns a pointer to the set name. If the number does not correspond to a set/state, the function returns a pointer to a volatile string containing the character version of the number.

3.3.4 Miscellaneous name functions

```c
char * NameToType(char *name);
```

If name is the name of a unit type, a unit, a site, a function, a unit state, a set, or an unused name in the name table, NameToType returns a pointer to a volatile string detailing the type of name, e.g. “unit vector name”. If the name is not found, it returns a pointer to the volatile string “unknown name”.

3.4 Display functions

```c
void ListLinks();
```
writes a one-line description of each link to the display output file *Disp*, with a header. The line consists of the four values: source index, destination index, weight, and data.

```c
DisplayUnit(u)
int u;
```

```c
DisplayUnitP(u, up)
int u;
Unit * up;
```

writes a complete description of the unit with index *u* to the display output file *Disp*, including descriptions of all the links to that unit. *DisplayUnitP* avoids having to index into the unit array by having the unit pointer passed in as a parameter.

```c
ListUnits(all)
int all;
```

is used for listing units. If *all* is nonzero (TRUE), it writes a one-line description of all units to the display output file *Disp*. If all is zero (FALSE) it writes the description only for units with the list flag set. Each line contains the unit index, name (**NO NAME** if not named), type, potential, output and state.

```c
ShowUnits()
```

does a show. If a unit has its show flag set, or it has potential greater than the show potential, or it is a member of a show set, then the unit is displayed in detail.

```c
PipeBegin()
```

If piping is turned on (PipeFlag non-zero) then the display output file *Disp* is set to be a file pointer to the `popen`'ed process whose name is maintained in `PipeCommand`, otherwise *Disp* is set to `stdout`. This function should be called before any of the display functions listed above.

```c
PipeBegin()
```

is the counterpart to *PipeBegin*. If *Disp* is not `stdout` then it closes the pipe and sets *Disp* to `stdout`. This function should be called after the displaying has been done.

```c
LOGfprintf(fp, str, ...
    FILE * fp;
    char * str;
    ...
```
An augmented/restricted version of *fprintf*. The string *str* is written to file *fp*, with substitution of the arguments for `%s`, `%d`, `%x`, `%f`, `%g`, and `%c`. If *Logging* is TRUE, the string is written to the Log file as well as to *fp*. If *fp* is *stderr* then a message count for the *Graphics Interface* is incremented. If *Format* is TRUE, the function formats the string into lines no longer than 75 characters, and indents all the lines thus formed by 3 spaces.

### 3.5 Set functions and macros

#### 3.5.1 Adding units to a set

```c
AddToSet(name,ulow,uhigh)
    char *name;
    int ulow,uhigh;

AddSet(name,uindex)
    char * name;
    int uindex;
```

adds units with index *ulow* to index *uhigh* to the set *name*. Returns TRUE if successful, FALSE otherwise. *AddSet* simply calls *AddToSet* with *ulow* = *uhigh* = *uindex*. Does a name lookup to find the set index.

```c
AddSetI(setindex,uindex)*
    int setindex,uindex;
```

is a macro which adds unit *uindex* to set with index *setindex*. Faster because no name lookup. *uindex* must be a valid unit index. For tight loops.

```c
AddSetP(setindex,up)*
    int setindex;
    Unit * up;
```

is a macro which adds unit pointed to by *up* to set with index *setindex*. Fastest because no name look up and no need to index into unit array. For tight loops.
3.5.2 Removing units from a set

RemFromSet(name,ulow,uhigh)
    char *name;
    int ulow,uhigh;
RemSet(name,uindex)
    char * name;
    int uindex;

removes units with index ulow to index uhigh from the set name. Returns TRUE if successful, FALSE otherwise. RemSet simply calls RemFromSet with ulow = uhigh = uindex.

RemSetI(setindex,uindex)*
    int setindex,uindex;

is a macro which removes unit uindex from set with index setindex. Faster because no name look up. uindex must be a valid unit index. For tight loops.

RemSetP(setindex,up)*
    int setindex;
    Unit * up;

is a macro which removes unit pointed to by up from the set with index setindex. Fastest because no name look up and no need to index into unit array. For tight loops.

3.5.3 Creating and deleting sets

DeclareSet(name)
    char *name;
creates a set with the given name. Returns the index assigned to the set, or -1 if the set could not be created.

DeleteSet(name)
    char *name;
deletes the set with the given name. Returns TRUE if successful, FALSE if the name is not that of a set.
3.5.4 Set theoretic functions

\begin{verbatim}
DifferenceSet(name, name1, name2)
    char *name, *name1, *name2;

IntersectSet(name, name1, name2)
    char *name, *name1, *name2;

UnionSet(name, name1, name2)
    char *name, *name1, *name2;
\end{verbatim}

assigns the difference/intersection/union of set \textit{name1} and \textit{name2} to set \textit{name}. All three sets must already exist. Returns TRUE if successful, FALSE otherwise.

\begin{verbatim}
InverseSet(name, name1)
    char *name, *name1;
\end{verbatim}

assigns the inverse of set \textit{name1} to the set \textit{name}. Both sets must already exist. Returns TRUE if successful, FALSE otherwise.

3.5.5 Set membership tests

\begin{verbatim}
MemberSet(name, uindex)
    char *name;
    int uindex;
\end{verbatim}

returns TRUE if unit with index \textit{uindex} is in the set \textit{name}. Otherwise returns FALSE, indicating the index was not legal, the name was not that of a set, or the unit wasn’t in the set.

\begin{verbatim}
MemberSetI(setindex, uindex) *
    int setindex, uindex;
\end{verbatim}

is a macro which calculates TRUE if unit \textit{uindex} is in the set with index \textit{setindex}, FALSE otherwise. Faster because no name look up. \textit{uindex} must be a valid unit index. For tight loops.

\begin{verbatim}
MemberSetP(setindex, up) *
    int setindex;
    Unit * up;
\end{verbatim}
is a macro which calculates TRUE if unit pointed to by up is in the set with index setindex, FALSE otherwise. Even faster because no name look up and no need to index into unit array. For tight loops.

```
MemberSetS(setindex, unitsetbits)
    int setindex, unitsetbits;
```

is a macro which calculates TRUE if unit set bits field unitsetbits has the setindex bit set, FALSE otherwise. Fastest because no name look up, no indexing into unit array, and no adding in offset for unit set bits field. Not much use unless each unit is being tested for membership of several sets.

### 3.5.6 Miscellaneous set functions

```
IsSet(name)
    char *name;
```

returns TRUE if the name is that of a set, FALSE otherwise.

### 3.6 Modifying and Accessing network values

```
SetOutput(index, value)
SetPotential(index, value)
SetState(index, value)
SetData(index, value)
    int index, value;
```

Unit index is given output, potential, state, or data value.

```
int GetOutput(index)
int GetPotential(index)
int GetState(index)
int GetData(index)
    int index;
```

Output, potential, state or data of unit index is returned.

```
SetWeight(ul, uh, us, usind, ulow, uhigh, uset, usetind, sitename, randomval, val, pert)
    int ul, uh, us, usind, ulow, uhigh, uset, usetind, randomval;
    FLINT val, pert;
    char * sitename;
```
This function sets the weights on one or more links. The first four parameters specify the units from which the links originate. The second four parameters specify the destination units. sitename is the name of the site at which the links arrive, or optionally ALL, meaning any site on a destination unit. The last three values specify the weight. If randomval is FALSE, the weight is simply val. If randomval is TRUE, then the weights are randomly distributed in the range val-pert to val+pert.

The source and destination units are in the range ul(ow) to uh(igh). If uset is FALSE, all the units in these two ranges are considered. If uset is TRUE, then only those units in the range that are in the set whose number is usind (for source units) or usetind (for destination units) are considered. Thus,

```c
SetWeight(0, 100, FALSE, 0, 1000, TRUE, 6, "excite", FALSE, 500);
```

would set the weight on any link from a unit in the range 0 to 100 to a site “excite” on a unit in the range 0 to 1000 which is in set number six to 500.

```c
RandomiseWeights(mean, pert)
    FLINT mean;
    int pert;
```

This function sets all the network weights to values in the range mean-pert to mean+pert. The values are evenly distributed throughout this range.

### 3.7 Library functions

There is one library function that is not a unit, site or link function. The other library functions are described in the User Manual, but included here for completeness.

```c
SiteValue(name, sp)
    char * name;
    Site * sp;
```

If sp is a pointer to a linked list of site structures, such as in the sites field of the Unit structure, and name is the name of one of the sites in the linked list, then the function returns the value of that site. If no such site is found, 0 is returned and an error message printed.

### 3.7.1 Unit functions

UFsum is a unit function which sets output and potential to the sum of all site values.
3.7. LIBRARY FUNCTIONS

3.7.2 Site functions

**SFmax** sets the site value to the maximum input value.

**SFmin** sets the site value to the minimum input value.

**SFsum** sets the site value to the sum of the input values.

**SFweightedmax** sets the site value to the maximum weighted input value. A weight of 1000 is treated as unity; the input value is multiplied by its weight and the result divided by 1000.

**SFweightedmin(up,sp)** sets the site value to the minimum weighted input value. A weight of 1000 is treated as unity.

**SFweightedsum** sets the site value to the sum of the weighted input values. A weight of 1000 is treated as unity.

**SFand** returns 1 if all its inputs are positive, otherwise 0.

**SFxor(up,sp)** returns 1 if exactly one of its inputs is nonzero, otherwise 0.

**SFprod** returns product of inputs.

3.7.3 Link functions

**LFsimple** sets the data field of the link to be the input value (unweighted). This does not affect the behavior of the network, but does help with debugging.

3.7.4 Weight scaling

The library functions assume that weights are scaled up by a factor of 1000. Thus a weight value of 500 represents a real weight of 0.5. This is to allow representation of values in the range 0 to 1 without having to use floating point arithmetic. The floating point library functions also scale by 1000 for compatibility.

Since all the functions that use weights (mainly site and link functions) can be written by the user, any weight scaling factor may be used. The only restriction is that if library functions that deal with weights (such as **SFweightedsum**) are used, weights must be scaled by 1000.
3.8 Unit index and pointer macros

LegalUnit(index)*
    int count;
computes TRUE if index is the index of an existing unit, FALSE otherwise.

UnitIndex(up)*
    Unit * up;
computes the index of the unit pointed to by up. If up does not point to a unit, computes garbage.

3.9 Flag Macros

Various macros are defined to set, clear, and test the flags in a unit. Each unit has 32 flags associated with it. Currently flags 0 to 6 are used by the simulator, and flags 7 to 11 are reserved for future simulator use. Flags 12 to 19 should be used for library packages, and so user code should be restricted to flags 20 through 31, preferably working from 31 down. Some of the simulator reserved flags may be set by the user for one or more units.

The user-settable flags are as follows:

- SHOW_FLAG : if set then the unit is in the Show set (for the show command).
- LIST_FLAG : if set then the unit is in the List set (for the list command).
- NO_LINK_FUNC_FLAG : if set then no functions are called for the links into a unit. This will result in speed up.
- NO_SITE_FUNC_FLAG : if set then no site or link functions are called for the unit.
- NO_UNIT_FUNC_FLAG : if set then no unit, site or link functions are called for the unit. The output of the unit remains the same.

3.9.1 Setting a flag

SetFlag(uindex,flagno)*
SetFlagP(up,flagno)*

SetFlag sets flag flagno in the unit with index uindex. SetFlagP does the same thing, but up is a pointer to the unit, thus avoiding indexing into the unit array.
3.9. FLAG MACROS

3.9.2 Clearing a flag

UnsetFlag(uindex,flagno)*
UnsetFlagP(up,flag)*

UnsetFlag clears flag flagno in the unit with index uindex. UnsetFlagP does the same thing, but up is a pointer to the unit, thus avoiding indexing into the unit array.

3.9.3 Testing a flag

TestFlag(uindex,flagno)*
TestFlagP(up,flagno)*
TestFlagF(uf,flagno)*

TestFlag calculates TRUE if flag flagno in the unit with index uindex is set, FALSE otherwise. TestFlagP does the same thing, but up is a pointer to the unit, thus avoiding indexing into the unit array. TestFlagF uses the bit vector uf (an unsigned int) and thus avoids adding in the offset for the flag field in the unit. TestFlagF is only of any use if several flags are being tested for each of many units in a tight loop.

3.9.4 An example

If flags are used a lot, it is advantageous to be able to use the macros which take a Unit pointer inside loops. An example of how this is done is a code fragment from the simulator source.

    register Unit * up;
    register int which, ucount;

    for ( which = 0, up = UnitList, ucount = NoUnits;
         which < ucount;
         which++, up++)
        UnsetFlagP(up,STEP_SIM_FLAG);

This code fragment clears the STEP_SIM_FLAG for every unit. To avoid having to index into the unit array, UnitList, for every unit, the loop maintains a current index, which, and a current unit pointer, up, both of which are incremented each time round the loop. Now the the pointer version of the flag clearing macro, UnsetFlagP can be used, so that no indexing into the unit array is ever done. For an array of thousands of units this can be significant, especially if the code were in a unit function.
3.10 Simulating

Reset()

resets the network: sets the system Clock to zero; sets the potential and state of each unit to init_persistent and init_state respectively; sets the output of each unit to zero

Step(count)
    int count;

simulates count steps. Echoes and shows will be done if appropriate.

Sync()

Further simulation steps will be synchronous.

Async(seed)
    int seed;

Further simulation steps will be asynchronous, as in the async command. The random number generator will be seeded with seed unless it is zero, in which case it will be seeded with the UNIX system time.

3.11 File functions

A number of functions are available that deal with files.

3.11.1 Network saving functions

FILE * GetNetFile(fname)
    char * fname;

If fname is NULL, asks the user for a filename to use for saving, opens the file and returns a descriptor to it. If fname is not NULL, a file with this name is opened and the descriptor to it returned. Will not overwrite or append to a file without user confirmation.

CloseNetFile()

closes the save file opened with GetNetFile (if one was).
3.11. FILE FUNCTIONS

NetSave(savef)
FILE *savef;
writes a time stamp into the file savef using StampTime, followed by the structure of the network (that is enough information to be able to reconstruct all the units, sites and links, including unit names and types and all function pointers. Finally calls SaveState to save the state of the network (i.e. the weights, outputs, potentials, etc.).

NetLoad(nfp)
FILE *nfp;
is the function complimentary to NetSave. Reads in and checks the time stamp in the file nfp with CheckStamp, reconstructs the network from the file data, and then calls RestoreState to restore the state of the network.

3.11.2 Network checkpointing functions

FILE * GetChkFile(fname)
char * fname;
If fname is NULL, asks the user for a filename to use for checkpointing, opens the file and returns a descriptor to it. If fname is not NULL, a file with this name is opened and the descriptor to it returned. Will not overwrite or append to a file without user confirmation.

CloseChkFile()
closes the checkpoint file opened with GetChkFile (if one was).

NetCheckpoint(savef)
FILE *savef;
writes a time stamp into the file savef using StampTime, followed by the state of the network, that is all the weights, potentials, outputs, etc. Uses the following function to save the state.

SaveState(savef)
FILE *savef;
writes the state of the network to file for reading in later.

RestoreNetwork(nfp)
FILE *nfp;
is the function complimentary to NetCheckpoint. Reads in and checks the time stamp in the file \textit{nfp} with \textit{CheckStamp}, and then restores the state of the network using the following function.

```c
RestoreState(nfp)
    FILE *nfp;
```

restores the state of the network from file \textit{nfp}.

### 3.11.3 Logging functions

**FILE * GetCmdFile()**

opens a file for logging the keyboard input only, and returns a descriptor to it. The file name is of the form \textit{run????.cmd.#} where ???? is the process ID of the current process, and # is an integer.

**SaveCmdFile()**

closes the file opened with \textit{GetCmdFile}, and asks the user if it should be saved. If the answer is yes, prompts for a name for the file and does a UNIX environment call to \textit{mv} it to that name. Otherwise it does a UNIX environment call to \textit{rm} the file.

**FILE * GetLogFile(fname)**

```c
    char * fname;
```

if \textit{fname} is NULL, asks the user for a filename to use for logging all i/o, opens the file and returns a descriptor to it. If \textit{fname} is not NULL, a file with this name is opened and the descriptor to it returned. Will not overwrite or append to a file without user confirmation.

**AskLogOn ()**

asks the user if (s)he would like to commence a logging the i/o. If the answer is yes, calls \textit{LogOn} to open a log file.

**AskLogOff ()**

asks the user if (s)he would like to close the current log file. If the answer is yes, calls \textit{LogOff} to close the log file.

**LogOn()**

asks the user for a file name to use for the log file (supplying a default), opens the file and stores the descriptor to it in \textit{LogFile}.

**LogOff()**

closes the file opened in \textit{LogOn(LogFile)}. 
3.11.4 Other file functions

**StampFile(nfp, type)**

```c
FILE *nfp;
int type;
```

writes a time, process and image name stamp to the file *nfp*. If *type* is TRUE (i.e. a checkpoint file) the process ID number is written first, otherwise (i.e. a save file) the process ID number is not written. Then the function writes out the name of the program that is running (i.e. the simulator executable) and the time it was made in human-readable form. Next it writes out the current time in human-readable form. Finally it writes out the current system time in seconds. Returns -1 on failure. For example:

```
Processid = 882
Image = sim written Wed Apr 29 15:03:58 1987
Current Time = Thu Apr 30 23:32:58 1987
546835378
```

**CheckStamp(nfp, type)**

```c
FILE *nfp;
int type;
```

reads in and checks a stamp made by *StampFile*. If *type* is TRUE, it expects a process ID number, if FALSE it expects no process ID number. The function will return -1 if the check fails because the stamp format is incorrect or missing. It will issue warnings if the file is over a week old, if the stamped image name is different to the current program file name (i.e. the simulator has been recreated), or *type* is TRUE, meaning a checkpoint file and the stamped process ID number is different to the current process ID number.

**ReadCmdFile(fname)**

```c
char *fname;
```

open the file named *fname* to read commands from. This may be nested (i.e. a command in one file causing this function to be called to open another command file) to a depth of 16 files.

**CloseCmdFile()**

close the file which is currently being read from, and continue reading from the one that caused this function call, or stdin if it occurred by the user issuing the read command.
3.12 Calling the Command Interface

It is possible to call the simulator debug command interface from user code, thus allowing a unit function to interrupt a simulation step or other function so that the user can examine or even modify the network mid-step. In fact this is the way the construction debugging facility operates. A more useful feature is the ability to build an interface on top of the simulator interface, which is exactly what the Graphics Interface is.

3.12.1 Calling the debug interface

```c
debug_command_reader(str)
    char * str;
```

This function runs the debug command interface. The parameter is a string to be used in the prompt. `Debug` should probably be incremented before calling this function (the value of `Debug` is printed in the prompt) so that it is clear from the prompt how many layers of interfaces are running. For instance, the control.C interrupt routine executes the following code fragment:

```c
Debug++;
debug_command_reader("interrupt");
Debug--;
```

resulting in the prompt “interrupt[2]>” where `Debug` has value 2. Most of the regular simulator commands can be called from the debug interface, except those to actually run the network. In addition the user can add their own command functions, as described in section 3.19. The `debug_command_reader` returns when the `quit` command is issued. Normally the user should zero `Errors` before calling `debug_command_reader`.

3.12.2 Running another interface on top of the standard one

```c
char * extern_command_reader(cmd_line) /* called externally */
    char * cmd_line; /* command string */
```

This is the function called by the Graphics Interface to pass a command to the simulator. `cmd_line` is a character string containing the command to be executed. `extern_command_reader` executes the command and returns a message string if any output to `stderr` occurred via `LOGfprintf` during the command, or `NULL` if there was no such output. An interface can be built on top of the simulator, while retaining all the simulator commands, by simply passing on all simulator commands to the simulator via this function. This command may also be used to execute a single simulator command, for example:
extern_command_reader("d u 3");

would cause unit 3 to be displayed in standard fashion.

3.13 Parsing command lines

Several functions make it easier to write new simulator commands. These functions will perform some simple lexical analysis and will process command line specification of units, if it is done in the standard fashion. The simulator passes command lines to command functions already parsed into an argv-argc structure. Whitespace in the command line indicates argument termination. To use the functions described in this section, the file "lex.h" should be included.

3.13.1 Lexical functions

int Lex(cmd)
    char *cmd;

Lex is the main lexical analyzing routine used to parse simulator commands. cmd is not the whole command, rather it is one argument from the argv-argc structure representing the command. Lex parses the argument, which should be an integer, a floating point number, a quoted string, a character string containing no whitespace, the character "+" or "-", or a unit identifier of the form name[index][index], where both indices are optional. Parsing unit identifiers should be done with GetUnits described below.

Lex will return the following values (#define 'd in the standard include file):

<table>
<thead>
<tr>
<th>Return value</th>
<th>character string</th>
</tr>
</thead>
<tbody>
<tr>
<td>CBRACK</td>
<td>]</td>
</tr>
<tr>
<td>OBRACK</td>
<td>[</td>
</tr>
<tr>
<td>END_STRING</td>
<td>'\0'</td>
</tr>
<tr>
<td>ALL</td>
<td>all</td>
</tr>
<tr>
<td>AUTO</td>
<td>auto</td>
</tr>
<tr>
<td>LINK</td>
<td>c or connections or link or links</td>
</tr>
<tr>
<td>CLOCK</td>
<td>clock</td>
</tr>
<tr>
<td>DEFAULT</td>
<td>def or default</td>
</tr>
<tr>
<td>FUNC</td>
<td>func</td>
</tr>
<tr>
<td>FROM</td>
<td>from</td>
</tr>
<tr>
<td>IPOT</td>
<td>ipot</td>
</tr>
<tr>
<td>ISTATE</td>
<td>istate</td>
</tr>
<tr>
<td>NAME</td>
<td>name</td>
</tr>
</tbody>
</table>
ON on
OFF off
OUT out
POT pot
RANDOM random
SHOW sh or show
STATE s or state or states
SET set
SITE site
TYPE type
TO to
UNIT u or unit or units
WEIGHT weight
HELP ?
STRING " <characters>"
PLUS +
MINUS -
INT <digits>
FLOAT -cdigits.digits»
IDENT anything other than the above

The most recently parsed token is held in the character string Yyarg. The most recently parsed integer is held in Yyint. The most recently parsed floating point number is held in Yyfloat. Thus command functions can check the return type and get the value or string from one of these three variables, if it is needed.

Cmdindex is the index into the argument string cmd that Lex has reached. It is reset to zero when the end of the string is reached. In most cases this will have happened by the time Lex returns, but when parsing, say, “RETINA[3][4]”, it will return on encountering the first “[”, and the next call to Lex will continue where it left off. As far as Lex is concerned, token delimiters are “\0”, “[” and “]”. Unit names, such as “RETINA[3][4]” should be parsed with GetUnits.

Curarg is the index into the command argv vector indicating the current command argument being processed. It is incremented by Lex on encountering the end of the string (cmd).

UnLex(tok)
    int tok;

Unlex effectively puts the most recent token processed by Lex back in the input stream, to be re-processed on the next call to Lex. Only one token can be Unlex’d.
3.13. PARSING COMMAND LINES

EAT is a macro that chews up the rest of the argument currently being processed. It is usually only used if Lex returns an unexpected type of token, and command parsing has to be abandoned. EAT then resets Cmdindex to an appropriate initial value for processing the next command. See Section 3.19 for an example of the use of Lex, EAT and GetUnits.

3.13.2 Parsing unit specifications

GetUnits(argc,argv)
    int argc;
    char ** argv;

GetUnits processes a specification of a range of units, given in the normal form (ask the “help UnitId” command for details). It sets the variables Ulow, Uhigh, Uset, and Usetind to indicate which units where specified. Ulow and Uhigh indicate the low and high unit indices in the range; if they are equal, a single unit was specified. Uset is a boolean indicating whether a set of units was specified, and if it is TRUE, then Usetind holds the number associated with the set (i.e. the index into the sets bit vector in the Unit structure).

GetUnits returns FALSE (= 0) if it fails, that is if a valid unit specification was not found. Otherwise it returns one of the following values:

Return value | meaning
-------------|----------
FALSE | invalid unit specification
ALL | all units
TRUE | other valid unit specification

FOR_UNITS(index)*

FOR_UNITS is a macro to use with unit range specifications. The parameter is an integer, which is used as a unit index inside a for loop to cycle through the unit range, using the values set by GetUnits in Ulow, Uhigh, Uset, and Usetind. For instance, the code:

FOR_UNITS(z)
    SetOutput(z,10);

would cause all the units in the specification parsed by GetUnits to have their outputs set to 10. An example of the use of FOR_UNITS in conjunction with GetUnits is given in the next section. FOR_UNITS_P is simply a version of FOR_UNITS that uses a unit pointer rather than a unit index, so that the code in the for loop need not index into the unit array. FOR_UNITS_P does the indexing itself in a fast manner, so this is the preferred macro for speed.
3.13.3 An example of a simulator command function

As an example, Table 3.2 contains the code for the disp command.

As described in section 3.19, this function obeys the standard format for command functions. First it tests to see if help information has been requested, and if so jumps to the helpinfo label. Then it sets Curarg to 1 (argv[0] is the command name) and calls Lex to parse the expected string "units", and tests that it was indeed found. The call to Lex will have incremented Curarg, so now GetUnits is called to find the units which should be displayed. If further parsing were required, Curarg would have the correct value, but we expect no more arguments for this command so proceed to display the units. PipeBegin is called to set up the pipe process for displaying (if one has been requested with the pipe command), and then the FOR_UNITS is used to call DisplayUnitP for all the units in the range.

3.14 Guarded code functions

When a control.C interrupt is issued from the keyboard, the simulator enters the interrupt interface (see Section 3.12.1), so that the network may be examined and possibly modified. To guard against entry to this interface when the network data structure is in the middle of being modified (and may therefore have inconsistent or invalid pointers), functions are provided to delay entry until a safe state is achieved.

Guard()

Release()

Guard simply increments Guarded and exits. Release decrements Guarded and checks if an interrupt has occurred since the previous call to Guard. If so, and if Guarded is zero, the interrupt processing routine is called to enter the interrupt interface. The fact that Guarded is incremented and decremented by each matched pair of calls to Guard and Release means that these calls may be nested, and interrupts will only be processed when the outermost level is reached.

3.15 Miscellaneous functions

NullFunc()

This is the function that does nothing. It is used for units, sites and links whose function specification is NULL in the call to MakeUnit, AddSite or MakeLink.

UserWait(str)
char * str;
3.15. MISCELLANEOUS FUNCTIONS

Table 3.2: Code for the *disp* command

```c
Cmd_disp(argc, argv)
int argc;
char ** argv;
{
    register int u;
    register Unit * up;

    if ((argc == 2) && (Lex(argv[1]) == HELP))
        goto helpinfo;
    if (argc > 2)
        {
            Curarg=1; /* first command argument */
            if(Lex(argv[Curarg]) == UNIT)
                {
                    if(!GetUnits(argc,argv)) return 0;
                    PipeBegin();
                    FOR_UNITS_P(u,up)
                        DisplayUnitP(u,up);
                    PipeEnd();
                    return 0;
                }
            else
                { /* EAT */
                    goto synerror;
                }
        }
    else
        goto synerror;

helpinfo:
    Format = TRUE;
    LOGfprintf(Disp,"The d isp command is used to display the \ values associated with one or more units, for instance the potential, \ output, state, functions, site names and values, link weights and values.\n");
    Format = FALSE;

synerror:
    LOGfprintf(Disp,"\nUsage: d isp u <UnitID>\n");
    return 0;
}
```
prints the provided prompt, \texttt{str}, then reads a single character in cbreak mode. Returns the character, mapped to lower case. Used to pause during simulation or display.

### 3.16 Simulator Variables

Many variables used by the simulator are accessible to user code. Modifying these should be done with care. The complete list follows.

- \texttt{int AutoFix} A boolean value that indicates whether automatic correction of construction errors is enabled.
- \texttt{int Clock} The system clock, or count of simulation steps.
- \texttt{FILE * CmdFile} The file to which keyboard input is written, if \texttt{LogCmd} is TRUE. The file is closed on exit from the simulator, and saved at user discretion.
- \texttt{int Cmdindex} Index into the current character of the current command argument. Used by \texttt{Lex}, see Section 3.13.1.
- \texttt{int Curarg} Index into \texttt{argv} structure for command functions, indicating next argument to process. See Section 3.19.
- \texttt{int Debug} Indicates the debug and interface level. Zero means normal interface, debugging switched off. One means normal level, debugging switched on. Incremented during construction commands such as \texttt{MakeUnit}.
- \texttt{FILE *DispF} The file to which display output is written, \textit{i.e.} during a \texttt{display}, \texttt{list}, or \texttt{help} command. If piping is enabled (see Section 3.4), the pointer is to the popen'ed process. Otherwise it points to \texttt{stdout}.
- \texttt{int Echo} A boolean value that indicates whether echoing is enabled.
- \texttt{int EchoStep} Number of steps between simulator echo messages.
- \texttt{unsigned int Errors} A bit vector containing the error types when debugging.
- \texttt{int ExecFraet} During fair asynchronous simulation (\texttt{fsync} command), the percentage of units simulated each step
- \texttt{int ExecLimit} During fair asynchronous simulation, the number of steps before all units have been simulated at least once.
- \texttt{int Format} A boolean value indicating whether \texttt{LOGfprintf}(see Section 3.4) should format the output.
- \texttt{int Guarded} Incremented by the \texttt{Guard}, decremented by the \texttt{Release} function (see Section 3.14), when zero code is not guarded against interrupt processing.
3.16. SIMULATOR VARIABLES

int LastSet Maximum number of sets allowed (currently 32).

int LastUnit Index of last unit that there is space for. Set in AllocateUnits

int LogCmd A boolean indicating whether keyboard input is being saved in the command
log file.

FILE * LogFile The file to which the log is written, that is the record of all keyboard
input and simulator output.

int Logging A boolean indicating whether logging is enabled.

int NoLinks Actual number of links made so far.

int NoSets Actual number of sets used currently.

int NoStates Maximum allowed number of states with names.

int NoUnits Actual number of units made so far.

Output * Outputs Vector of unit output values. Links get their values with a pointer
into this array. Updated in Step.

int Pause A boolean indicating if pausing is enabled.

char PipeCommand [] Name of pipe process to use for display output, if piping is en-
abled.

int PipeFlag A boolean indicating whether piping is enabled.

char **SetNames Array of pointers to set names, indexed by set number.

int Show A boolean indicating whether showing is enabled.

unsigned int ShowSets A bit vector indicating which sets are in the Show set.

int ShowPot During a show, display all units with potential higher than this value.

int ShowStep Perform a show every this many steps.

int StateCount Actual number of states with names.

char ** StateNames Array of pointers to state names, indexed by state value.

int SyncFlag Simulation update protocol. Can by SYNC (synchronous, sync command),
FAIRASYNC (fair asynchronous, fsync command) or ASYNC (asynchronous, async
command).

int Uhigh, Ulow Set by GetUnit(s) (see Section 3.13.1), indicates range of units found.
Unit * UnitList  Pointer to unit array, the main data structure.

int Uset, Usetind  Set by GetUnit(s) (see Section 3.13.1). Uset is a boolean indicating if
a set name was found, in which case Usetind is the set index.

char Yyarg[] Contains the token most recently parsed by Lex.

float Yyfioat Contains the floating point value most recently parsed by Lex.

int Yyint Contains the integer value most recently parsed by Lex.

3.17 The Name Table

All the names used in the simulator (for units, unit types, sites, states, functions and sets)
are stored in the global name table. User code may also insert names into the name table,
look them up, and delete them. The name table access functions are:

    char * EnterName ( name, type, data1, data2, data3 );
    char * AlterName ( name, type, data1, data2, data3 );
    NameDesc * FindName ( name, descriptor_pointer );
    char * name;
    int type, data1, data2, data3;
    NameDesc * descriptor_pointer;

EnterName will enter the name in the table. type is used by the simulator to determine
what type of name it is. Type numbers 0 through 8 are used by the simulator, and numbers
9 through 99 are reserved to the simulator. Libraries should use numbers 100 through 999,
and user code numbers 1000 and up. The three data fields are simply entered in the table.
If the name is successfully entered, the function returns the pointer to the stored name
character string. Care should be taken not corrupt this character string. EnterName will
fail and return NULL if the name is already in the symbol table, unless all fields match
those already in the table.

AlterName is just like EnterName, except that it never fails; if the name is already in
the table, it is simply overwritten. This function should be used with extreme care.

FindName looks up the name in the name table, and fills in the descriptor with the
table entry contents. It returns a pointer to the descriptor if the name was found, and
NULL if not. The descriptor is of type NameDesc (see below for details), so a typical call
to FindName might look like:
3.18. CUSTOMIZING UNIT, SITE AND LINK DATA STRUCTURES

NameDesc descriptor;

if (FindName ("some-name", &descriptor) == NULL)
    printf("can't find name: %s\n", "some-name");
else
    ...  

The types defined by the simulator are as follows:

<table>
<thead>
<tr>
<th>Type</th>
<th>Value used for</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCALAR</td>
<td>single unit name</td>
</tr>
<tr>
<td>VECTOR</td>
<td>unit vector name</td>
</tr>
<tr>
<td>ARRAY</td>
<td>unit array name</td>
</tr>
<tr>
<td>SET_SYM</td>
<td>set name</td>
</tr>
<tr>
<td>STATE_SYM</td>
<td>state name</td>
</tr>
<tr>
<td>STRING_SYM</td>
<td>unused name (may be re-used)</td>
</tr>
<tr>
<td>TYPE_SYM</td>
<td>unit type name</td>
</tr>
<tr>
<td>FUNC_SYM</td>
<td>function name</td>
</tr>
<tr>
<td>SITE_SYM</td>
<td>site name</td>
</tr>
</tbody>
</table>

Name table entries have the following structure:

```c
typedef struct n_i_desc
{
    char  *name; /* Pointer to name */
    short type; /* Type of unit {0=SCALAR, 1=VECTOR, 2=ARRAY} */
    short size; /* Size of vector if VECTOR, number of columns if ARRAY */
    short length; /* Number of rows if type 2 */
    int index; /* Index of first unit name applies to */
    struct n_i_desc *next;
} NameDesc;
```

Each name must have a unique use, i.e. one cannot use the same name for a site that is used for a unit, type, function, state, set, etc.

3.18 Customizing unit, site and link data structures

Each unit, site and link structure contains a field, data, which is for general purpose use. This field is the size of an integer or float, depending on which simulator is being used, but in any case is assumed to be the same size as a pointer. Therefore it is possible to use this field as a pointer to an arbitrary user-defined data structure. The simulator uses
types `unit_data_type`, `site_data_type` and `link_data_type` for the unit, site and link data fields respectively. By using the -D flag in `makesim` (see Section 3.23 and the man page for `makesim`) the user can re-define the type, but must define it to be of size four bytes, so that user code is compatible with the simulator code.

3.18.1 Redefining the data type

Suppose, for instance, one wanted to delay the incoming values on a link by an arbitrary time steps. Then the data field for each link could be used as a pointer to a structure which stored the previous input values and weights. For example:

```c
typedef struct inp
{
    short weight;
    short value;
} input;

typedef struct l_d_type
{
    short count;
    input * inputs;
} link_data;

typedef link_data * link_data_type;
typedef int site_data_type;
typedef int unit_data_type;
```

Here `unit_data_type` and `site_data_type` are defined to be `int`, just as the default. But `link_data_type` is now a pointer to a structure of type `link_data`, which contains a `count` field specifying the length of the propagation delay on the link. It also has a pointer, `inputs`, to a vector of type `input`, each element of which contains fields for `weight` and `value`.

3.18.2 Making a link

Since the simulator itself has no knowledge of this redefinition, it will not allocate space for the `link_data` structure, nor for the vector of `input`'s. Thus the user code must allocate the space, conventionally at the same time that `MakeLink` is called to make the link. For example, user code would do something like:

```c
typedef link_data * link_data_type;
typedef int site_data_type;
typedef int unit_data_type;
```
3.18. CUSTOMIZING UNIT, SITE AND LINK DATA STRUCTURES

... 
lp = MakeLink(from, to, "excite", 1000, 0, LFdelay);
MakeLinkData(lp, delay);
...

where delay is the number of simulation steps delay and MakeLinkData is:

MakeLinkData(lp, count)

    int count;
    Link * lp;

{
    int i;

    lp->data = (link_data_type) malloc (sizeof(struct l_d_type));
    lp->data->count = count;
    lp->data->inputs = (input *) malloc (sizeof(input) * count);
    for (i = 0; i < count; i++)
        lp->data->inputs[i].weight = lp->data->inputs[i].value = 0;
}

The incoming parameters are the pointer to the link, and the count of the number of steps delay. First the link.data structure is malloc ed, and the pointer to it stored in the link.data field. Then the count is stored in the malloc ed structure, and the vector of length count and type input is malloc ed, the pointer to it being stored in the link.data structure. Finally all values and weights in the vector are initialized to zero.

3.18.3 The link function

Now the link function, LFdelay, would simply shift the values along one place, and store the current input value in the final vector location.

LFdelay(up, sp, lp)

    Unit * up;
    Site * sp;
    Link * lp;

{
    int i;

    for (i = 0; i < lp->data->count - 1; i++)
        lp->data->inputs[i] = lp->data->inputs[i+1];
    lp->data->inputs[i].weight = lp->weight;
    lp->data->inputs[i].value = *(lp->value);
}
The for loop shifts the vector entries up one place (towards the first entry - or top if it is thought of as a stack). Then the incoming values are stored in the final entry, or bottom of the stack.

### 3.18.4 The site function

Suppose we just wanted a standard weighted sum to be computed at the site, but using these delayed inputs. The function could be:

```c
SFdelayweightedsum(up,sp)
Unit * up;
Site * sp;
{
    Link * lp;
    int sum = 0;
    for (lp = sp->inputs; lp != NULL; lp = lp->next)
        sum += lp->data->inputs[0].weight * lp->data->inputs[0].value;
    sp->value = sum/1000;
}
```

The for loop simply sums the weighted inputs from the top of the stack of delayed inputs, and scales the result appropriately.

### 3.18.5 Linking with simulator code

Since the unit, site and link structures are specified (at the C level) in a file that is included in user code, one might wonder how this code will ever compile. The solution, as hinted at above, is to provide a flag to the `makesim` command (see Section 3.23) that specifies a file of user defined data structures to include instead of the standard ones. This is the -D flag; check the `makesim` man page. The file must define the types `unit_data_type`, `site_data_type`, and `link_data_type`. These types must be of the same size as an integer or float. Although the simulator code will continue to treat the data fields as an integer, since the simulator never actually uses these fields, it is permissible for user code to treat the field as a pointer, or any other integer-sized structure.

### 3.18.6 Displaying, saving, loading, etc.

The problems arise in displaying, saving and loading. By default the simulator would display, save, and load the data field as an integer or float. The pointer value would not be restored on a load, and the extra structure would not be malloced. The solution to this problem is for the user to write specially named functions to display, save and load these
structures. In addition the general help information provided by the simulator when the Help command (see Section 3.20) is used might need to be augmented. The functions that may need to be written are as follows:

- **User.Unit_Display(fp,up)** called when the Unit structure is displayed.
- **User.Site_Display(fp,up,sp)** called when the Site structure is displayed.
- **User.Link_Display(fp,up,sp,lp)** called when the Link structure is displayed.
- **User.Link.List(fp,up,sp,lp)** called when links are listed.
- **User.Unit_Checkpoint(fp,up)** called when the Unit structure is checkpointed.
- **User.Site_Checkpoint(fp,up,sp)** called when the Site structure is checkpointed.
- **User.Link_Checkpoint(fp,up,sp,lp)** called when the Link structure is checkpointed.
- **User.Unit.Restore(fp,up)** called when the Unit structure is restored.
- **User.Site.Restore(fp,up,sp)** called when the Site structure is restored.
- **User.Link.Restore(fp,up,sp,lp)** called when the Link structure is restored.
- **User.Unit.Save(fp,up)** called when the Unit structure is saved.
- **User.Site.Save(fp,up,sp)** called when the Site structure is saved.
- **User.Link.Save(fp,up,sp,lp)** called when the Link structure is saved.
- **User.Unit.Load(fp,up)** called when the Unit structure is loaded.
- **User.Site.Load(fp,up,sp)** called when the Site structure is loaded.
- **User.Link.Load(fp,up,sp,lp)** called when the Link structure is loaded.
- **User.Help_Info()** called after the general help information has been printed when the Help command is used.

where the arguments are a file pointer for the display information to be written to and pointers to the Unit, Site and Link.

If these functions exist in user code, then the simulator will call them to handle the data field for units, sites, and/or links, and to print extra general help information. If they do not exist, the default simulator action of treating the field as an integer or float will be taken. Examples of some of them follow, using the propagation delay definitions given in the preceding sections.
3.18.7 Displaying units

User_Link_Display(fp, up, sp, lp)
   FILE * fp;
   Unit * up;
   Site * sp;
   Link * lp;
{
   int i;
   for (i = 0; i < lp->data->count; i++)
      fprintf(fp, "\t\t%d delay - value: %d weight: %d\n", i+1,
               lp->data->inputs[i].value,lp->data->inputs[i].weight);
}

This function is called when the display command is issued from the simulator command interface. Instead of printing the link data field, the simulator calls this function to display the information. The display information is written to the file pointer, with lp being a pointer to the link currently being displayed, sp a pointer to the site at which the link arrives, and up a pointer to the unit to which the site is attached. In this case the latter two pointers are not used, but they are passed as parameters to the functions dealing with links for completeness. As can be seen, the function simply prints out the stack of delayed input values and weights. Then when a unit is displayed, the link would be printed in the following fashion:

link:**NO NAME** (3) func:LFdelay weight:1000 value:0
   1 delay - value: 0 weight: 0
   2 delay - value: 0 weight: 0
   3 delay - value: 0 weight: 0

3.18.8 Listing links

The list command lists each link, and as part of the display prints the value of the link data field. In our example we might wish to have the delay associated with the link displayed instead.
3.18. CUSTOMIZING UNIT, SITE AND LINK DATA STRUCTURES

User_Link_List(fp, up, sp, lp)

{ 
    fprintf(fp,"delay: %d\n",lp->data->count);
}

As in the link display function in the previous section, the file pointer and pointers to the unit, site and link are passed in as arguments. The function simply prints the delay value in the count field.

3.18.9 Checkpointing and Restoring

When a checkpoint command is issued from the simulator command interface, the state of the network is saved to file, that is the values in the data structure. On a restore command, these values are restored from the file. To ensure that the extra structures we have defined and created are also checkpointed and restored, the following functions are written:
User_Link_Checkpoint(fp,up,sp,lp)
    FILE * fp;
    Unit * up;
    Site * sp;
    Link * lp;
{
    int i,j,k;

    fprintf(fp," %hd",lp->data->count);
    for (i = 0; i < lp->data->count; i++)
        fprintf(fp," %hd %hd",lp->data->inputs[i].weight,
            lp->data->inputs[i].value);
    fprintf(fp,"
")
}

User_Link_Restore(fp,up,sp,lp)
    FILE * fp;
    Unit * up;
    Site * sp;
    Link * lp;
{
    int i, count;

    fscanf(fp,"%d",&count);
    for (i = 0; i < count; i++)
        fscanf(fp,"%hd %hd",&lp->data->inputs[i].weight),
            &lp->data->inputs[i].value));
}

User_Link_Checkpoint prints out the count field, corresponding to both the delay on the link and the length of the vector of incoming weights and values. Then it prints out the vector of weights and values. User_Link_Restore simply does the complimentary thing; it reads in the count, and then reads in that number of incoming weight/value pairs, and saves them in the vector. The only important thing about these functions, and it is crucially important, is that the Restore function reads in exactly the same amount of data that the Checkpoint function wrote out. Otherwise, the restore process will fail.

3.18.10 Saving and Loading

When a save command is issued from the simulator command interface, the structure and state of the network is saved to file. On a load command, the network is built in the simulator, and the state reset. The saving and loading of the state are handled by an internal call to the checkpoint and restore process, so that the only extra thing that needs
to be done is write functions to save the structure of our extended links, and recreate that structure when loading.

```c
User_Link_Save(fp,up,sp,lp)
    FILE * fp;
    Unit * up;
    Site * sp;
    Link * lp;
{
    fprintf(fp," %d ",lp->data->count);
}

User_Link_Load(fp,up,sp,lp)
    FILE * fp;
    Unit * up;
    Site * sp;
    Link * lp;
{
    int count;

    fscanf(fp,"%d",&count);
    MakeLinkData(lp,count);
}
```

The only important factor in the extended link structure is the length of the vector of delayed input weight/value pairs. Thus `User_Link_Save` simply writes out this length. When loading, the extra structures we defined have to be explicitly created; the link is being made afresh. `User_Link_Load` reads in the size of the vector, and calls the data structure creation function `MakeLinkData` (see Section 3.18.2) to allocate and initialize the space. The values will be restored by `User_Link_Restore` when the restore process is internally called by the simulator.

Once again, the most crucial thing to get right with these functions is that the `Load` function reads in exactly the same amount of data as the `Save` function writes out.

### 3.18.11 Unit and Site functions

In our example, the unit and site data fields are simply the simulator default integers. If they had been redefined to be some structure, we would need to write corresponding functions to deal with them, named as in section 3.18.6.

### 3.18.12 Completing the example

Using the functions described above, we may complete the example with a function to build a small network. This example is included in the `example/userdef` subdirectory. The build function simply creates ten units, each one linked to the other nine, with the delay on each
link set to the absolute difference between the unit indices. This is not meant to represent anything, just to provide a simple example.

```
build()
{
    int i,j,k, u;
    Link * lp;

    AllocateUnits(10);
    for (i = 0; i < 10; i++)
    {
        u = MakeUnit("neuron",UFsum,0,0,0,0,0,0);
        AddSite(u,"excite",SFdelayweightedsum,0);
    }
    for (i = 0; i < 10; i++)
        for (j = 0; j < 10; j++)
            if (i != j)
                {lp = MakeLink(i,j,"excite",1000,0,LFdelay);
                    MakeLinkData(lp,(10+i-j)%10);
                }
}
```

The only departure from conventional network construction code is the call to `MakeLinkData` as each link is constructed.

### 3.19 Customizing the simulator command interface

Any user written function which has a name commencing “Cmd.” will be treated as a regular command by the simulator, and will be available at all interfaces. The simulator passes an `argc-argv` structure to command functions. There is a standard format for command functions, one aspect of which is obligatory. The function must allow for the case when `argc` is 2 and the `argv[1]` is the string “?” This occurs when the the user types `command ?` or `help command`. In this case the function should print help information, including the command syntax. Conventionally single character command functions, e.g. `Cmd_e`, are abbreviations for other commands, and so are not listed when the user types `?` to the prompt.

Suppose one wanted a command, `linkdata`, to set one of the delayed weight/value pairs in a link (as in the example above). In canonical form, this might look like the code in Table 3.3.

Much more error checking should be done, but for brevity it is omitted here. The important point is the initial check for the `help` condition described above (note the use of `Lex`). If it exists, there is a jump to the help information label `helpinfo`. Here the simulator global
#include "lex.h"

Cmd_linkdata(argc, argv)

    int argc;
    char ** argv;
{
    int from, to, delay, weight, value;
    char sitename[100];
    Unit * up;
    Site * sp;
    Link * lp;

    if (argc == 2 && Lex(argv[1]) == HELP)
        goto helpinfo;
    if (argc != 7) goto synerror;
    sscanf(argv[1], "%d", &from);
    sscanf(argv[2], "%d", &to);
    sscanf(argv[3], "%s", sitename);
    sscanf(argv[4], "%d", &delay);
    sscanf(argv[5], "%d", &weight);
    sscanf(argv[6], "%d", &value);
    up = &UnitList[to]; /* get unit pointer */
    for (sp = up->sites; /* find site */
            sp != NULL && strcmp(sp->name, sitename);
            sp = sp->next);
    if (sp == NULL) goto synerror;
    for (lp = sp->inputs; /* find link */
            lp != NULL && lp->from_unit != from;
            lp = lp->next);
    if (lp == NULL) goto synerror;
    if (delay > lp->data->count) goto synerror; /* check delay in range */
    lp->data->inputs[delay-l].weight = weight; /* set new values */
    lp->data->inputs[delay-l].value = value;
    return 0;

helpinfo:
    Format = TRUE; /* print detailed help */
    LOGfprintf(DispF, "The linkdata command is used to set the weight \\
    and value in a time delayed link. You must give, in order, the unit \\
    from which the link originates, the unit to which it goes, the site at \\
    which it arrives, which delay you want to adjust the values for, the \\
    new weight, and the new value\n");
    Format = FALSE;

synerror:
    LOGfprintf(DispF,
        "\nUsage: linkdata [\nFrom-UnitID] \nTo-UnitID] \nTo-site > \n<delay> \n<weight> \n<value>\n\n\n");
    return 0;
}

Table 3.3: Example command interface code
variable *Format* is set to TRUE so that *LOGfprintf* will format the output (see Section 3.4). The help information is written to *DispF* since *help* is one of the commands whose output is piped. The string inside the call to *LOGfprintf* actually contains no newlines (except at the very end) – the backslashes you see at the ends of lines merely allow continuation of strings across multiple lines in the C programming language. After the help information has been displayed, formatting is switched off.

The other label, *synerror*, is conventionally used if a syntax error is discovered while processing the command *argc-argv* structure. The command syntax is printed, again to *DispF* since conventionally the help information should include the syntax specification. If piping is turned off *DispF* will be *stdout*, so everything works even if *synerror* is jumped to from elsewhere in the command function code.

Any function whose name begins with "Debug_Cmd_" will be available at the debug interface, but not the normal interface. It will take precedence over any normal command of the same name. For example, to create a different *linkdata* command at the debug interface, the function would be called "Debug_Cmd_linkdata". This function would be called when the *linkdata* command was issued at the debug and interrupt interfaces, but "Cmd_linkdata" would be called at the normal interface.

All command functions should return 0, whether there is an error or not, with the exception of any command used to exit the debug interface, such as the simulator function *Debug_Cmd_quit*. Any such function should return TRUE. *Errors* is a bit vector used within the debug facility. One cannot exit the debug facility to the normal interface until *Errors* is zero.

### 3.20 Adding to the Help information

If a simulator has been customized it may be that extra information describing the customization should be printed with the general help information (i.e. the message that is printed when the command *help* is issued). The customizer can write a function, *User_Help_Info*, which will be called by the simulator after the standard help information has been displayed. Continuing the example in the preceding two sections, such a function could be:
3.21 AVOIDING NAME CLASHES

User_Help_Info()
{
    Format = TRUE;
    LOGfprintf(Disp, "This simulator is augmented to model fiber \ 
propagation delays. If you display a unit, you will notice each link \ 
has a series of delays. Every link has at least one delay. At a \ 
given time step, the current incoming value and the current weight are \ 
saved at the bottom of the list. At each time step the values and \ 
weights percolate up the list one place. When a value/weight pair \ 
reaches the top of the list, it is used by the site function.\n\n")
    Format = FALSE;
}

As with the command function, formatting is switched on for the call to LOGfprintf, and off after it.

3.21 Avoiding name clashes

To avoid name clashes, do not use any names for functions, variables, data types, etc., that begin with si_. These are reserved to the simulator code.

3.22 Floating Point version

The floating point version of the simulator uses floats for the weight, data, output, and potential fields of the unit, site and link structures. In addition the function pointers for the unit, site and link functions are declared float * instead of int *. User code will be compiled with the -DFSIM flag so that conditional compilation is possible if one wants to freely move the code between integer and floating point simulators. In addition the type FLINT is defined to be an int in the integer simulator and a float in the floating point simulator.

3.23 Linking user and simulator code

User code must be compiled and linked with the simulator object files to form an executable simulator. makesim is a UNIX Bourne shell script to perform this somewhat complex process. The steps that must be taken are:

1. Compile the user source files.

2. Extract the names of user functions from the user object files and build a program to load the function names and pointers into the simulator function table. Thus the
simulator will have a mapping from function name to function pointer, allowing the user to call the functions from the simulator command interface. This process also finds the function names in all libraries. The program `grabnames` interrogates object and library files and its output is fed to the program `makebind` which builds the appropriate C source file.

3. Compile the C source file created in the previous step.

4. Link the user object files, the object file created in the previous step, and the appropriate simulator object files.

`makesim` has a number of flags, which are described in the man page. Note that the user must have write permission in the simulator source directories to use the `-z` (compile simulator development version) flag. There is a mechanism in `makesim` for checking that user files have been compiled for at most one type of simulator (i.e. either integer or float). Briefly, the standard include file for user code (either `sim.h` or `fsim.h`) contains a specially named statically declared integer, with different names for the different types. `makesim` checks the user object files to ensure that at most one of these names exists.

### 3.24 The simulator as a subroutine

It may be desirable to call the simulator as a subroutine from another program. This has been facilitated by splitting the simulator into initialization and run functions, and moving the top level `main` function to a separate file. The new simulator can be installed by `making` these targets in `src/uniproc`: `main int float`.

#### 3.24.1 Controlling the simulator from another program

Once the simulator has been installed, to use it as a subroutine of another program the following steps must be taken:

1. Make a relocatable simulator. Issue the normal `makesim` command but give it the `-R` flag as well. For example, to make an empty relocatable simulator with graphics, simply `makesim -R` will be enough. This will produce an object file, with default name `sim.o`.

2. Place calls in the program code to initialize and run the simulator. The function `simulator_setup(argc, argv)` initializes the simulator. There should be at least one string in the `argc-argv` structure, the first string in the controlling program's `argc-argv` structure (i.e. the name of the controlling program). The function `simulator_run` transfers control to the simulator code. This function returns when the `return` command is issued to the simulator interface, or the `quit` button on the Graphics Interface is clicked. The function `simulator_quit` closes down the graphics window and should be called before exiting the controlling program.
3. Link the relocatable simulator object file (*sim.o*) with the object files for the program which is to call the simulator.

### 3.24.2 Return command

**Syntax:** return  
**Example:** return

This command returns to whatever invoked the simulator, normally UNIX. If the simulator was called by some other program, control is returned to that program.

### 3.24.3 Integration with MIT Scheme and Kyoto Common Lisp

The simulator has been integrated with MIT Scheme and Kyoto Common Lisp. The directories `contrib/scheme` and `contrib/kcl` contain instructions and the necessary modifications to Scheme and KCL. To use KCL, modifications to make the `malloc` package work are given. To use Scheme, some interface primitives are provided. Given the choice, Scheme is recommended over KCL for memory allocation efficiency reasons.

As an example, to integrate the simulator with the MIT Scheme package, step 2 above would require writing primitives which can be called from scheme to call `simulator_setup`, `simulator_run` and `simulator_quit`. These primitives and the simulator relocatable object are then added to the `makefile` for the Scheme package, and the package recompiled. Then, in Scheme, calling the primitives would allow Scheme to run the simulator. Additional primitives might also have to be written if it is desired that Scheme be able to access unit activation levels.
3.25 Implementation Details

This section of the manual is a high-level overview of the uniprocessor implementation of the simulator. The graphics interface implementation is described in Section 4.14.

3.25.1 Main Data Structure and Execution Cycle

The main data structure, representing units, sites and links, is illustrated in Figure 3.1. Each unit is represented by a data structure of type Unit with various fields, as shown in the diagram. Space for the units is allocated in a vector of these structures, with global variable UnitList pointing to the first unit in the vector. Normally units are referred to by their index into this vector. Sites for a unit are represented with a linked list of Site structures. Links arriving at a site are represented with a linked list of Link structures. In Figure 3.1, all pointers are shown with arrows, other fields are simply for data storage.

A unit has the following fields:

- output - used to store the current output of the unit.
- potential - used to store the current potential of the unit.
- state - used to store the current state of the unit.
- data - undefined use by the user.
- func - a pointer to the unit function.
- type - a pointer to a character string.
- name - a pointer to a character string.
- flags - a vector of bits of information about the unit.
- sets - a vector of bits giving set membership for the unit.
- init_state - initial state of the unit, when network is reset.
- init_potential - initial potential of the unit, when network is reset.
- no_sites - number of sites attached to the unit.
- sites - pointer to the linked list of sites.
3.25. IMPLEMENTATION DETAILS

Figure 3.1: Main Data Structure
A site has the following fields:

- **value** - used to store the current value at the site.
- **data** - undefined use by the user.
- **s_func** - a pointer to the unit function.
- **name** - a pointer to a character string.
- **no.inputs** - number of links arriving at the site.
- **inputs** - pointer to the linked list of incoming links.
- **next** - pointer to the next site in the list (NULL terminated).

A link has the following fields:

- **value** - a pointer to the incoming activation value.
- **data** - undefined use by the user.
- **weight** - the weight associated with the link.
- **l_func** - a pointer to the unit function.
- **from-unit** - the index of the unit where the incoming activation is from.
- **next** - pointer to the next site in the list (NULL terminated).

In addition to these structures, there is an array of output values, pointed to by *Outputs*, with the same number of elements as the unit array. The *value* field in each link structure points to one of the elements of the output array, as shown in Figure 3.1. The reason for this separate output array is in part explained in the next paragraph.

The execution cycle, performed by function *Step*, is as follows (for synchronous mode). For each unit in turn, through the array of units, calculate the site function at each site in turn down the linked list, followed by the unit function, and then execute all link functions at links arriving at the unit. When all units have updated, copy the value in the output field of each unit into the corresponding location in the output array.

In asynchronous operation, the units are updated in pseudo-random order, with the output values copied from the each unit to the output array as the unit is updated. One of the bits in the units *flag* field is used to control this process. Units are selected at random from the entire population. If the STEP_SIM_FLAG is not set, the unit is updated and the flag set. Otherwise another unit is selected, and the process continues until all units have been updated. At the end of the step, the flag is unset on all units.
3.25. IMPLEMENTATION DETAILS

In fair asynchronous operation, at each step a similar process occurs, except that only a certain fraction of the units are updated. In addition to the \texttt{STEP\_SIM\_FLAG}, which is set and unset on each step, another flag is set each time a unit is updated -- the \texttt{LIMIT\_SIM\_FLAG}. When the limiting number of steps is reached, all units whose \texttt{LIMIT\_SIM\_FLAG} is not set are updated, and then the \texttt{LIMIT\_SIM\_FLAG} is unset for all units.

In normal operation the site function will look at all the links in its list of links, combining the value, weight and data fields to yield a total value for the site, which is placed in the \texttt{value} field for the site. The unit function will look at all the sites in its list of sites, combining the value fields and the units state, potential and data fields to yield an output for the unit. The link function, often omitted, would modify the link weight or other parameters.

The \texttt{type} and \texttt{name} fields point to character name strings holding the type and name for the unit or site. All name strings are indexed in the Name Table, described in the next section. The actual character string is maintained in a single location, with the name table and unit fields pointing to the same string. Manipulation of these fields should be handled with care since many units' fields as well as the name table may point to a single character string.

3.25.2 NameTable

The simulator NameTable is a standard hashed bucket table, each bucket being a null-terminated linked list of name table entries. A name table entry has a pointer to the character name string (\texttt{name}), a pointer to the next entry in the list (\texttt{next}), a tag specifier (\texttt{type}), and three data fields (\texttt{index}, \texttt{size}, \texttt{length}). The use of these fields is dependent upon what what type of name it is (i.e. the \texttt{tag} value). Users have access to the name table, both for reading and writing. The standard name types defined are:

- \textbf{SCALAR} - a name associated with a single unit. The \texttt{index} field holds the index of the unit.
- \textbf{VECTOR} - a name associated with a one dimensional vector of units, which are simply a contiguous group of units in the Unit Array. The \texttt{index} field holds the index of the first unit in the group, and the \texttt{size} field the number of units in the group.
- \textbf{ARRAY} - a name associated with a two dimensional array of units, which are again a contiguous group of units in the Unit Array. Conceptualize this as follows: the 2D array is linearized by concatenating the second row onto the end of the first, the third onto the end of that, \textit{etc}. Then this big vector is mapped onto a section of the Unit Array as with one dimensional unit vectors. In this case the \texttt{index} field holds the index of the first unit in the section, the \texttt{size} field the number of columns and the \texttt{length} field the number of rows. Evolutionary development is to blame for the confusing field names.
3.25.3 Command Interface

The command interface has been designed to be easily extensible. The basic concept is that there is a function associated with each command, the name of the function being the command name with the string “Cmd.” prepended. All these function names are known in the name table. When the user types a command line, it is read and parsed into tokens separated by whitespace. These are placed in an argv-arc structure just as for entry to a standard UNIX program. The first token is assumed to be a command name. The command interpreter prepends “Cmd.” to the command name and looks up the resulting function name in the Name Table. If it is not found, an error message is posted. If it is found, the index field of the name table entry gives an index into the Item Table (see Section 3.25.5). The entry in the Item Table contains a pointer to the function. The command interpreter then simply calls this function with the argvargc pair as arguments.

Since all global functions in user code are made known to the Name Table (see Section 3.25.9), all the user need do to create a new command is write a new function with the appropriate name. The format for these functions is described in Section 3.19. The standard command functions are contained in the files commoncmds.c, conunicmds.c and simunicmds.c, all such functions having names which commence with the string “Cmd.”.

3.25.4 Debugging package

Debugging during network construction is controlled by the global variable Debug. Its value describes the debug level, level zero meaning debugging turned off, level one meaning
debugging turned on, and levels two and above meaning debugging on and processing an
error. If debugging is on, then calls to MakeUnit, AddSite and MakeLink are routed through
routines in the file debug.c which check the validity of the parameters. A bit vector Errors
is used in these routines to accumulate errors. If errors are found, the debug interface is
entered. The command interpreter for this interface is essentially a copy of the normal
command interpreter, except that instead of looking for functions with names of the form
"Cmd.xxx" it first looks for a function with the name "Debug_Cmd.xxx". If such a function
is found, it is called. Otherwise the regular "Cmd.xxx" function is searched for and called.
In this way various commands can perform different functions at the two interfaces. Until
all errors are fixed, debugging turned off, or the ignore command issued, the user cannot
exit from the debug interface (except by typing control-D, which always exits to UNIX).

Since various network construction calls can be issued from the debug command in­
terface, such as MakeUnit, error processing is nested. Every time the debug interface is
entered, Debug is incremented; when the debug interface is exited, it is decremented. Func­
tions in debug.c take care of pushing and popping the information about each error on the
procedure call stack (it’s somewhat convoluted).

Debug commands are located in the same files as the regular commands, the function
name always beginning with the string “Debug_Cmd.”. As with regular commands, users
can write new debug commands by prepending this string to the command name.

3.25.5 Dynamic Code Loading and the Item Table

Dynamic code loading is achieved by examining the object file to be loaded, linking it with
various files and examining the result to find out how much space is needed for the code,
mallocting the required space, resolving references in the code, and reading the code into
the malloced space. Most of these actions are achieved via judicious use of the UNIX ld
command.

Figure 3.2 shows the process. Commencing at the top left of the diagram, with the user
object file and the simulator executable file (whose name is in si_Program), the simulator
executes a UNIX ld command to link the user code using the simulator code file symbol
table, at address zero. The resulting object file is opened to find out how much space is
needed for the code; space for the user code is then malloced. Once again the user code is
linked using the simulator code file symbol table, but this time at the address just malloced.
The resulting code file is read into the space malloced in core, as shown at the bottom left
of the Figure 3.2.

Now we have the user code in core, with all references from user code to simulator
functions resolved. But we still need to enter all user functions and global variables in
the Item Table, so that they will be accessible from the simulator interface. This process
is shown on the right of Figure G. Using the nm UNIX command the simulator extracts
the names of the user functions and variables from the user object file and puts them in
another file. Then it reads this file and writes a C program into another file. This program
Figure 3.2: Dynamic Code Loading
3.25. IMPLEMENTATION DETAILS

consists mainly of calls to \textit{AddItemToTable}, using the names of the functions and variables. The program contains one function, which makes all these calls. The simulator compiles the program, and then links it at address zero using the symbol table from user object file which was read into core. The resulting code file is opened to find out how much space is needed for the program, and the space is malloced. Once again the program is linked using the symbol table from user object file which was read into core, this time at the address of the just malloced space. The resulting binding object file will have the addresses of all the user functions and variables resolved; it is read into the space malloced, the program is executed (using the entry point to the single function) thus entering all the items in the Item Table, and the space is freed. The final item entered in the Item Table by the binding program is the code unit just loaded, so that the address of the space holding the code is retained.

Now we have the user code in core, and the user's functions and variables in the Item Table. The final thing to do is go through the entire network to set unit, site and link function pointers to point to the function just read in, if appropriate. This is done by \texttt{si\_ChangeFuncPointers}.

The Item Table is shown in Figure 3.3. Each entry has three fields:

- \textit{item} - a pointer to the item (function, variable or codeunit).
- \textit{name} - a pointer to the character string containing the name of the item (the same string used by the name table).
- \textit{next} - an index into the next item in the chain of items.

Items in the Item Table are formed into chains, one chain associated with each code unit. The first item in the chain is for the code unit, the remainder are for the functions and variables in the code unit. The list is terminated by an invalid value in the \textit{next} field (-1). Every item is also placed in the Name Table. The \textit{index} field of the name table entry contains the index into the Item Table. The \textit{size} field of the name table entry contains the index into the Item Table of the code unit containing the item.

There are items in the item table that vary from these rules. These are the functions loaded on entry to the simulator - the standard command functions and user functions linked with the \texttt{makesim} shell script. These functions are not linked in a list - the \textit{next} field in the item table entry is always -1. In addition the name table entry for the item has its \textit{size} field set to -1. In this way statically loaded functions can be distinguished from dynamically loaded functions.

When a code unit is dynamically loaded, the simulator first checks to see if a code unit of the same name is already loaded. If it is, then the existing chain of items for the code unit is removed from the Item Table, but the chain is saved. If dynamic loading fails for whatever reason, including failure to enter the new items in the item table, then the original chain is replaced. Otherwise, upon successful completion of loading and binding, the original
Figure 3.3: Item Table with Chain

The Item Table is automatically expanded in size as more items are entered. This is accomplished by mallocing a larger table, copying the old one to the new one, and freeing the old table.

Another complication is the possibility that a code unit being loaded contains an item whose name is the same as that of an item in another code unit. In this case, the original item remains, since it may be referenced in the original code unit, but the entry in the Item Table for that name is updated to the new item, and if it is a unit, site or link function the network pointers are updated. Thus the old item will not be overwritten, but it will be inaccessible from the user interface. If this process of gradual attrition of items in code units causes all the items in a loaded code unit to become inaccessible from the interface (and therefore from anywhere), then the space containing the code unit is freed.

Chain is deleted and the space containing the original code freed. Chain manipulation is performed by the functions `si_RemoveCodeUnit`, `RestoreCodeUnit` and `si_FreeCodeUnit`.

[Diagram of Item Table with Chain]
3.25. IMPLEMENTATION DETAILS

3.25.6 Simulating Propagation Delay

The original simulator has been modified to allow simulation of propagation delay along links. The basic concept, illustrated in Figure 3.4, is to associate with each unit a buffer of output values. The elements of the output array are now pointers to the buffers, rather than containing outputs explicitly. The link pointers then point to a location in this buffer. At each simulation step, the values in the buffer are all shifted along one place and the new output value copied into the first position. Thus the link value pointer always accesses the output of the unit delayed by however many elements along the buffer it points to.

Buffer management is as follows. A call to MakeDelayUnit specifies the length of the buffer to be associated with a unit. If the unit is made with MakeUnit the buffer is of length 1 (i.e. just one output position in the buffer, plus the count entry specifying the number of elements in the buffer). When links are made with MakeDelayLink, if the buffer is not long enough a new buffer is malloced, the links into the old buffer re-mapped to the new buffer, and the old buffer freed. The re-mapping causes a search through every link in the network built so far – this can be exponentially time consuming. Thus it is best to make units with the correct size buffer to start with, or a little larger. Performance degrades if the buffers are too long – since the values are pushed along each buffer at each simulation step. The function ScavengeBuffers reduces each buffer to the smallest possible size.

In the code, conditional compilation for propagation delay is achieved with the symbol TSIM.

3.25.7 Memory Management

The memory items which are in any sense managed are the units, sites and links, name table entries, item table entries, and propagation delay buffers. Of these by far the most important are the links, since in almost every network these will predominate. Space for links and sites is allocated in blocks. The link blocks are chained forwards and backwards, with the head of the chain pointed two by First.Link. The chaining is achieved by using the last link element in each block to point to the previous and next blocks in the chain, rather than using it as a link in the network structure. The next field points to the first element in the next block in the chain, the value field to the last element of the previous block. Isn't C wonderful? Site blocks are also chained, but only in the forward direction. The reason for the backward chaining of the link blocks is that we want to be able to delete links and reclaim the space, as describe below. The unit array is allocated by a call to AllocateUnits. If more units are required, a larger array is malloced, the old array copied to the new array, and the old array freed. In addition the array of outputs (or buffer pointers) is extended in a similar fashion.

To enable deletion and reuse of links and sites, we maintain NULL-terminated free lists of each of these elements. When the list is empty, we allocate a new block of links or sites and chain the elements together (using the next field) to form the free list. When links or sites are deleted, they are added to the free list. Now if many links are deleted, we may wish
CHAPTER 3. ADVANCED PROGRAMMING MANUAL

Figure 3.4: Simulating Propagation Delay
to free up some of the space represented by the links on the free list. This is accomplished with the function ScavengeLinks. The basic concept is as follows. First all links on the free list are marked by setting their value field to NULL. Then the links attached to sites, which are connected in linked lists, are linked in the reverse direction, so now forming doubly linked lists. The value field of each link is used to point to the previous link. The first link (the one pointed to by the inputs field of the site) has its value field set to point the site, and this link is further marked by negating the value in its from_unit field. For the propagation delay version, this is slightly modified: in addition to doubly linking the lists using the value field, the high bit of the from_unit field is set for the first link in each list (that pointed to by the site), and the next highest 8 bits (23 – 30) are used to record the delay on the link. This means that the maximum delay is 2**8 or 256 steps, and that there can be no more than 2**23 or approximately 8 million units in any simulation using the propagation delay facility.

To scavenge the links, we use two pointers freelp and usedlp. freelp always points to an unused link. It starts at the beginning of the first link block (pointed to by First_Link) and moves through each block in turn. usedlp starts at the last link in the last link block and works backwards towards the first link in the first link block. It always points to a link which is in use (value pointer not NULL). usedlp is copied into freelp, the pointers in the doubly linked list of links which held usedlp updated, and freelp and usedlp moved to the next free and used link respectively. When they pass each other the used links have all been moved to the blocks at the beginning of the block chain. The empty blocks are then freed, and the value and from_unit fields of the used links reset to their original values. Finally the remaining free links are chained and form the free list. Or something like that.

Name Table entries are reclaimed if unused (entry field type is STRING_SYM) by the function ScavengeNames. The Item Table can get bigger, but never smaller, as described in the section on Dynamic Loading. The propagation delay buffers may be scavenged, as described in the preceding section. Buffer scavenging works like this. First we go through all links and in each buffer mark the maximum delay needed in the first buffer location. Then for each buffer that is longer than needed, we reallocate the buffer. Finally we reset the first buffer location to the correct value, i.e. the value in the unit output field.

### 3.25.8 User defined network extensions

As described in the user manual, it is possible for the user to redefine the data field for units, sites and links to point to arbitrarily large structures. If such redefinition is performed, the user must write functions to handle the structures on network saving and checkpointing. These functions must be loaded with the makesim shell script. Then the program makebind, called by the shell script, puts pointers to the user functions in simulator global variables (User_Link_Data, etc.). All this should be done through the item table; the implementation is a result of evolutionary development.
3.25.9 The shell script *makesim*

This shell script performs many functions. First it compiles user code files. Then it uses the UNIX command `nm` to extract the names of global symbols in the user code and libraries (in a similar fashion to the dynamic code loading facility). The programs `grabnames` and `makebind` in the src/tools directory are used to extract the names of the user and library functions from the file produced by `nm`, and to write a program to add those functions to the Item Table. This binding program is then compiled, and linked with the user object files, the libraries and the simulator object files to form the final executable.

During construction of the simulator object file, the object files containing the simulator basic commands are examined in a similar way, using the program `cmdmkbbind`, and a program to add these command functions to the Item Table created, compiled and linked with the other component object files to form the final simulator object file.

When the executable is run, the program calls the routine which enters the standard command functions into the Item Table, and then calls the routine which enters the user and library functions into the Item Table. In this way, when the simulator prompt appears, the command functions, user functions and library functions are loaded in the Item Table.

3.25.10 Control flow with other packages

It is possible to run the simulator as a subroutine of another program. There are two routines provided to accomplish this. The function `simulator_setup` is called to initialize the simulator and graphics. The function `simulator_run` is called to transfer control to the simulator interface, or graphics window. Control is returned to the calling program by issuing a “return” command or clicking the quit button on the graphics. When there is no graphics, the return command simply sets a flag that causes control flow to exit from the `simulator_run` function. If the graphics package is in use, then `simulator_run` causes a `tool.select` to be issued for the graphics window, transferring control to that window. The “return” command or “quit” button simply cause a `tool.done` to be issued for the graphics window, returning control to the calling window.

3.25.11 Saving, Loading, Checkpointing and Restoring

These functions are performed by writing the network structure and state to file, or reading it in. The only complexity occurs with handling names. First the name table is dumped in its entirety. As dumping occurs, a count of the number of function, site and type names is kept, and the `length` field of the name table entry for such a name is given the current value of the count. Then the network structure is dumped, using the values in the `length` field of the name table entry instead of the name for sites, functions and unit types. Finally the `length` fields that were changed are reset to zero. Upon loading, the name table is read in and a vector of pointers to site, function and type names created. As the network is
reconstructed, the values dumped for site, function and type names are used as indices into this vector to find the actual character strings for the names.

Saving and loading back-propagation networks is not possible at present, since the error propagation links which use the link.f field in the links cannot be restored. It is not clear how to do this cleanly, but the information is all there since each link maintains the index of the unit whence the link originates.
Chapter 4

Graphics Interface User Manual

4.1 Background

The Graphics Interface (from now on called GI) was developed as an extension to the Rochester Connectionist Simulator for use on Sun graphic workstations. It provides a graphic output display for networks created by the Rochester Connectionist Simulator making it possible to observe the behavior of the network as it runs. Once your network has been built by the simulator, GI gives you a display panel upon which can be arranged graphic symbols (icons) representing particular aspects (potential, output, state, links, etc.) of units of your network that you want to observe. As simulation steps are run, the appearance of these icons will change as the values of their selected aspect change. Thus you can visually observe the dynamic overall behavior of your network (or part of it) as it runs, which can be very useful in getting an intuitive feeling for what the network is doing. You can also draw and write text and line drawings on the display panel for documentation of your network. Remember that GI is strictly a “read-only” interface; that is, anything you do with the graphics has no affect on your network itself and the base simulator is (almost) completely unaware of GI's existence.

Release 4.2 includes an X11 interface (referred to as XGI), which is described in section 4.15. XGI provides significantly greater functionality than GI. It was developed from GI, and programs that use GI should be portable to XGI with little effort. The description of XGI assumes familiarity with the use of GI, so new users should read the following sections which describe GI before reading section 4.15.

4.2 Getting Access to GI Functions

The GI is linked into the simulator itself at “makesim” time automatically. If you do not want the GI graphics package you should specify the -ng flag with the makesim command. GI comes in two “flavors”: an integer version and a floating point version. The integer
version is the default, but using the -f flag with makesim pulls in the floating point version that is consistent with the simulator. See Section 2.8 for more details of the floating point version of the simulator.

Because SunView graphic tools pull a lot of library routines into their objects, the final load objects tend to be quite large, usually between 500 and 1000 KB (depending on whether you compile with the -g option). You will probably want to delete these objects once you are finished with them if disk space is a critical resource as it is at the University of Rochester.

RESTRICTION: If your own simulator code makes use of external variables you must be aware that name clashes are possible between your code and GI functions since they are linked together as one object. All GI external variables and functions have names beginning with the prefix “gl.”, so you would be wise to avoid naming any of your external objects using this prefix.

4.3 Using the GI Tool

Once you have made your simulator with “makesim” you can run your simulation session with the GI interface. Simply type in the name of your simulator object as usual. You will notice a message when the simulator comes up “building Graphic Interface tool – please wait”. It will take a few seconds and then you should see the GI tool come up on the right-hand side of your screen (see Figure 4.1). You can use any of the SunView “.W” flags following the simulator command if you wish to customize aspects of the tool. For example, the command:

```latex
mysim -Wp 0 100 -Ws 1000 800
```

would bring the GI tool up on the extreme left-hand side of the screen, 100 pixels from the top and sized at 1000 pixels wide by 800 pixels high. See the SunView documentation for a complete list and explanation of all the -W options available. Note: if you are using the optional saved simulator file parameter, it must appear before any of the -W flags.

Once the GI tool has appeared on the display, you will notice it consists of six separate (but interrelated) panels:

- The **info** panel is the top panel of the display and is used for showing detailed textual information about specific units in your network.

- The **mode** panel is the wide short panel immediately below the info panel and is used both for changing “modes” (described below) and for turning logging on or off (also described below).

- The **display** panel is the mostly large blank panel on the left side of the window just below the mode panel. It is the canvas on which the graphical representation of your network will be displayed.
4.3. USING THE GI TOOL

Figure 4.1: Initial GI tool window
• The control panel is the complicated looking panel with all the buttons and prompts and is just to the right of the display panel. This panel is the primary way you will interface to the GI functions in order to control the graphical expression of your network on the display panel.

• The message panel is the thin panel directly below the display panel. It is used solely for displaying errors, warnings or informational messages.

• The command panel is just below the message panel and is the bottom most panel of the GI tool. It is your primary interface with the base simulator and essentially takes the place of its command interface.

Like most Suntools, in order to work within a particular panel, you need to move the cursor into it. Successful operation of GI requires understanding how these panels operate and how they relate to one another. A detailed description of the operation of each panel is contained in the next sections.

4.4 The Command Panel

The command panel is the the bottom leftmost panel of the GI tool and usually the first one you will use. It consists of a prompt “->” which looks very much like the standard simulator prompt for good reason: it essentially is the simulator prompt. That is, any simulator command that you type followed by a return is sent unchanged to the simulator command interpreter and executed. For example, if the first thing you normally do is build your network with “call buildmynet”, that is exactly what you would type into this prompt. The simulator will process your command, and once it returns, the command will move up to the next line and you can type in another command. The only difference between using this command line and the one on the standard simulator is that any text output generated by the simulator (for example on a “list” command), will show up in the original window the simulator was started in, not in the GI tool window. Thus if you plan to use simulator commands for generating displayed output, you should set up your Sun windows so that you can see both the GI tool and the original simulator window at the same time. However, if the simulator command generates an error message from the simulator, the error will appear in the GI message panel as well.

4.5 The Message Panel

The message panel is the simplest because the only interaction you will have with it is to read text that has been put there by GI for your information or befuddlement. Mostly (hopefully) they will be confirmational messages like “Show command successful”. Sometimes there will be warning messages to inform you of something that happened that you may not be aware
of (such as that a unit has been “displayed” off the viewable screen) but may be OK anyway. At other times there will be error messages informing you with utter clarity as to what went wrong and maybe even a clue on how to fix it. Hopefully, very seldom will you see obscure looking messages like “undesignated <what> in get_unit procedure” which are indications of a program failure. If you run into any of these consistently, please let us know so we can try to figure out the problem. You should also see any error messages resulting from commands sent to the simulator in this window.

4.6 The Mode Panel

The mode panel is right below the info panel and controls two separate functions. The left side of the panel has the word MODE: followed by the choices “Main”, “Link”, “Text”, “Draw” and “Custom” with one of them reverse-imaged. Clicking over “Mode:” or one of the choices will change modes accordingly. Different modes allow you to do different operations on the display panel. You will notice that switching from one mode to another results in a different set of prompts being displayed on the control panel. The mouse actions performed in the display panel will also change based on the current mode. A hint as to what the three mouse buttons do will appear in the message panel each time a mode is selected. Briefly, the different modes are used for the following functions.

- **Main** mode is used primarily for setting up and displaying the units in your network on the display panel. In main mode the control panel will have prompts that pertain to what, how and where you want the units in your network displayed.

- **Link** mode is for interactively examining and verifying the topology of your network. That is, in Link mode the icons being displayed are always showing connection strengths between units rather than some other aspect such as their output, potential, state, etc.

- **Text** mode is for placing printable ASCII characters on the display primarily for documentation and publication purposes. The control panel prompt asks for a font which allows a variety of type faces and sizes to be used on the display.

- **Draw** mode allows you to draw boxes and other straight line objects on the display for documentation and aesthetic purposes.

- **Custom** mode enables you to set the mouse buttons to specific commands that can be executed while the mouse is on the display panel. These commands can have symbolic arguments that are filled in at execution time based on where the cursor is located in the display panel.

More detail on how to work in these modes is contained in the following section: *The Control Panel.*
The right side of the mode panel contains two fields for controlling the logging of commands. **LOG:** is a switch which can be set either “on” or “off” by clicking over it with the mouse and indicates whether you want GI and simulator commands you issue to be logged to a file. If logging is “on” then actions that affect the display screen will be turned into commands and written to the file named in the prompt just to the right of the switch. The default log file used is named “gi.log” but of course you can change this to any file name you want. When logging is turned off, any commands that had been written to the log file are immediately flushed into the file and can thus be read in immediately if desired. If you then specify another log file and turn logging back on, then the original log file will be closed and the new log file will be opened for write (deleting its old contents) as soon as the next command completes. However if you don’t change log file names, and switch logging back on and off, then the log file will accumulate commands that are issued whenever logging is on.

The reason you might want to log commands is that you can then “replay” your simulation session back later (via the simulator “read” command) thus saving yourself the trouble of setting up the network display. You can also edit the log file to “weed out” or change the way the session will be reenacted. Of course to do this you will have to understand the syntax of the simulator commands – which we assume you are already familiar with – and the GI commands which are documented in section 4.10. More detail regarding the log file is contained in section 4.11.3.

### 4.7 The Control Panel

The control panel is located on the right side of the GI tool and is normally useful only after you have built your network (presumably by interfacing to the simulator through the command panel). Through interacting with it you set up the display with the graphic representation of your network, run the network and do other miscellaneous functions such as writing the display image to a file or placing text or line drawings on the display. The control panel is divided into an upper and lower part. The upper part (above the Clock and Origin messages) is used primarily for setting up the display panel the way you want it to look and will have different prompts displayed depending on what mode you are in. (See previous section on the mode panel.) The lower part is used mostly after the network has been laid out for actually running and watching the simulation run. It looks and operates the same in every mode and we’ll describe it next.

#### 4.7.1 Lower control panel – running the simulation

The buttons on the lower part of the control panel are most useful during and after a simulation run. The **GO** button is for actually running some number of steps of the simulation once the network has been laid out on the display screen. It has two associated prompts, one (“number steps:”) specifies the number of steps to run, and the other (“update steps:”)
indicates how often to update the display while the simulation is running. For example setting “number steps:” to 20 and “update steps:” to 4 will cause 20 simulation steps to be run with the display being updated every 4 steps.

The DUMP button is used to save the actual display panel image in a raster file. The default name is “gi.image” but you can change that to any name you wish. If you have utilities that can display or print raster files, these can be used to display or print that image of the display panel.

The RESHOW button simply rewrites everything onto the screen (from scratch, so to speak), so if in moving things around, the display has somehow gotten messed up, RESHOW should put things right again. It has a prompt: “:” which defaults to the current Origin. If you would like the display to show a different portion of the display space, change the value to the desired origin before pressing RESHOW and the display will be translated appropriately using the new origin (which will then become the current origin). The relationship between display space and the origin, is be explained in detail in section 4.8.

The QUIT button does exactly that: ends without recourse the simulation session and returns to the shell. Make sure you have saved everything you need to before pressing this button since it gives no second chances.

### 4.7.2 Main Mode — laying out the network

When you are in main mode (see Figure 4.2), the upper part of the control panel will have three buttons representing the three basic commands that are used for initial layout of your network on the display panel: SHOW, CHANGE and ERASE. They all act by taking the appropriate parameters from the the prompts (WHO, HOW MANY, WHERE, etc.) and then performing the requested action on the display screen.

The SHOW command is used to specify how and where to display units that are not already currently displayed. The ERASE command is used to erase units from the display (not from your network) that currently are displayed. Thus ERASE is the functional inverse of SHOW. The CHANGE command is used to change some representation of units that are already displayed. CHANGE is sort of a combination (or shortcut) of the ERASE and the SHOW commands. That is, it acts as though the specified units were first ERASEd and then SHOWn again with (possibly) different attributes.

The prompts (WHO, HOW MANY, WHAT, etc.) need to be filled in appropriately before pressing any of the command buttons. Note that defaults are set up in all the prompts. In fact, once you have built your network you can immediately left click over the SHOW button which will execute the default SHOW command thereby laying out your entire network in a default fashion. However, for your specific purposes this default layout may be somewhat inappropriate. In order to specify the exact way you want your network to appear on the display panel, you need to understand the semantics of the prompts and how they apply to each command:
Figure 4.2: GI tool window: Main mode
WHO (along with HOW MANY) determine which units in your network will be SHOWn, CHANGEd or ERASEd. The WHO part specifies the beginning unit (if HOW MANY specifies more than one) in an ordering of units by unit index. There are four ways to specify a unit in the WHO prompt – By unit index (note the default is “0” indicating the first unit in any network), by unit name, by unit type or by set name. These are all things you can specify via functions in the basic simulator within the C code that you wrote to build your network. For convenience in starting over, the ERASE command will allow you to specify “all” for WHO which will erase everything that has been displayed.

HOW MANY specifies the number of units the command is to be applied to, beginning with the unit specified in the WHO prompt. This can be either a decimal number or the word “all” which means all units matching the WHO prompt.

Note: Depending on what the command is, the beginning unit specified by WHO is handled slightly differently. For SHOW, the WHO unit will be the first unit (starting from unit 0) that matches a unit which is not currently displayed. So if, for example, your network has 100 units of type “input”, then setting WHO to “input” and HOW MANY to “10” and pressing SHOW will display the first ten of them. Pressing the SHOW button again will display the next 10, and so on until SHOW can’t find 10 units of type “input” that aren’t already displayed (which will then result in an error message). The ERASE command sort of works in reverse. Once all 100 input units are displayed (and setting WHO to “input” and HOW MANY to “10”) ERASE will erase the first 10 displayed, pressing ERASE again will erase the next 10, and so on until ERASE can’t find 10 “input” units that are still displayed. CHANGE always works on the 1st unit it can find that is displayed. Thus setting WHO to “input” and HOW MANY to “10” and pressing CHANGE multiple times will always affect only the first 10 “input” units displayed.

WHAT specifies two things: the “aspect” of the unit you wish displayed (Potential, Output, State, Data, Link/in, and Link/out) and the expected range of values that you expect that aspect to take on during the simulation. The desired aspect is selected by clicking left over it (which becomes reverse imaged) and you specify the range by typing into the “from:” and “to:” prompts. Note that the default is to display the unit potential over a range of -1000 to 1000. The meaning of “Potential”, “Output” and “State” should be self-evident. “Data” refers to the data field contained in each unit. Link/in and Link/out specify that the aspect of the unit you want displayed is the link weight of that unit to (Link/in) another (target) unit, or the weight from (Link/out) another (target) unit. Note that when you select Link/in or Link/out as the aspect another prompt appears labeled “target:” This is where you specify the target unit in the same manner that you specified WHO. Note, however that the target unit is just one particular unit. Thus if you specify a set name or type, the target will be just the first one found. The safest way to specify the target to make sure you get just what you want is by unit index or unit name. If the target happens to already be
on the display screen, clicking the right mouse button over it will automatically copy that unit's index to the "target:" prompt for you. If you specify just a unit in the target field, GI will use the first matching link it can find for that target. However if there is more than one link between the target and a unit you may want to specify a particular link. You can therefore add a "site" designation to the target as well. The site designation is simply the site name where the link is attached and you specify it by appending the site name to the unit separated by a slash ("/"). Thus if you were interested in the link between target unit 43 at site "special" you would put "43/special" into the target prompt. (If you have multiple links between two units all at the same site you out of luck; there is no way in GI to distinguish among them).

• HOW specifies what kind of graphic object(s) you want the unit aspect displayed as. There are six choices; the first five are polygons and the last a dark square, is a grey-scale icon. You select the one you want by clicking left over it which moves a horizontal bar beneath the selected choice. If you don't like any of the choices presented, there is a seventh choice (on the next line) designated by an icon with a "?". If you select the "?" icon, you are prompted to fill in the "name:" of your own icons. (See Section 4.11.1 for details.) The default choices given are only the prototype of what the unit will actually look like; the actual appearance of the unit during the simulation will depend on its current value for the aspect selected for it. The icons shown are what the unit will look like when and if it reaches the maximum value for its selected aspect. The way the value of the unit aspect is shown for the five polygons is by the two dimensions of size and shading. Each shape has 16 different icons associated with it and based on the particular value of the unit aspect at the time, one of those shapes is selected to represent the unit and is then displayed. What actually happens is that the range is divided into 16 evenly sized subranges and whatever subrange the value happens to falls into, that corresponding icon is displayed. (If the value falls outside the range, it is treated as though it were the maximum or minimum of the range). The 16 icons are ordered such that the maximum value in the range corresponds to a large, light colored polygon, and as the values decrease the polygon becomes smaller and smaller until at the middle of the range (normally 0) the point is reached at which the polygon is at its smallest. As the value gets smaller (normally more negative) the icon becomes dark and begins to get larger until at the low end of the range it appears as a large dark polygon. The exception to the above rule applies to the grey-scale icon. At its largest value it is a dark rectangle, and decreases by becoming progressively lighter until at its minimum value it is almost a completely white space it however, does not change size. By the way, if you prefer the reverse (have the dark polygons or lighter grey-scales represent larger values) simply switch the numbers in the "to:" and "from:" prompts.

• WHERE is used to control where on the display panel the aspect of the unit you have selected will appear. As such it is only looked at by the SHOW command. There are several things that need to be specified about the positioning of the icons. They all
require that you understand a little about the geometry of the display panel and how it is referenced.

The objects on the display panel are positioned by x and y coordinates with 0,0 being the upper left-hand corner of the panel with x becoming increasingly positive as you move to the right and y becoming more positive as you move down. This is in accordance with Sun conventions for window geometry. On the display panel you may notice an odd looking “X”-like object which from now on will be referred to as the “marker”. (If you don’t, simply click left over the display panel and the marker will appear there). If you look at the prompts “start x:” and “start y:” you will notice that they have coordinates already in them. These happen to be the coordinates of the marker on the display panel. Notice that if you change these coordinates, the marker will correspondingly change position. Similarly when you move the marker around (by clicking left on the display panel where you want it to move to) the coordinates will automatically change to reflect its new position. This synchronicity between these prompts and the marker comes in handy when laying out units of your network since the “start x:” and “start y:” prompts indicate where the first unit will be positioned on a SHOW command. If only one unit is going to be displayed, then the rest of the WHERE prompts are irrelevant. However if a number of units are going to be displayed, the other WHERE prompts are used to specify how the whole group are to be laid out. The underlying strategy is to lay out multiple units in a rectangular fashion in rows and columns. in “reading order”, that is, left-to-right and top-to-bottom. The “start x:” and “start y:” designate the top left of the rectangle, “space x:” and “space y:” specify how many pixels to leave between each column and row respectively, and “units per row:” tells how many icons to put in each row. If “units per row:” is set to max, SHOW will put as many all the icons in one row even if that means some of the icons will be off the display panel. Be assured however that even though the units are not presently shown, they are they are still there and in the next section you’ll learn different ways to make them visible. If you want the units displayed in a right to left or top to bottom order, simply use negative values for the “space x:” and “space y:” prompts, respectively. Note that you can make the units “overlap” by specifying small (less than the icon size) negative spacings. However, GI does not guarantee that the results will be pretty.

Once appropriately filled in, clicking left over the SHOW, CHANGE or ERASE buttons will perform the specified command. It useful to fool around with these commands in order to get a feel for how to arrange and rearrange the units. An important thing to remember is that the SHOW command uses all the prompts, the CHANGE command uses all the prompts except WHERE, and the ERASE command only uses WHO and HOW MANY. Remember that no amount of messing around with the display will have any effect on the network itself; in a sense GI is “read only” as far as the actual network is concerned.

4.7.3 Link Mode – checking the connections
Figure 4.3: GI tool window: Link mode
Selecting Link mode on the mode panel will put you in link mode (see Figure 4.3). Link mode was designed for the single-minded purpose of making it easy for you to tell how your network is connected. No matter what aspect your unit icons were displaying in the other modes, in link mode all icons are always tracking the weight of links from or to some one other unit called the target unit. Link mode is as if you CHANGED all the units displayed to either Link/in or Link/out to or from a particular target unit. The advantage to link mode is that the original definitions of your units are not lost. That is, no matter what you do in link mode, switching back to main or any other mode will restore the units to what they were before you entered link mode. Thus think of link mode as a temporary escape from the "usual" definitions of the unit icons to one where only links are displayed. Here's how to use it.

Notice that the control panel has three prompts in it: TARGET, HOW and DIRECTION. The idea is to pick a unit as the "target" in much the same way as you do in main mode, either by clicking the right mouse button over the target unit or by typing the target unit name (or index, type or setname) into the TARGET prompt. You can add a site name to the TARGET prompt by appending it to the unit separated by a "/". Once you pick a target (by either clicking or pressing return in the TARGET prompt) all the other icons on the display will change to show the weight of their link (if any) to that unit. The other two prompts are for controlling how the links are shown:

- **DIRECTION**: which way the link is supposed to go. Link/in indicates that each unit on the display will show the weight of the link from the unit to the target; Link/out specifies that each unit will show the weight of a link to the unit from the target.

- **HOW**: what kind of icon and using what range of values. For choice of icon, the familiar polygonal ones are available as well as the default one labeled "same". If one of the polygonal ones are chosen, then all the displayed units will change to that one icon shape (only during link mode, of course). Selecting "same" means to use the same icon shape for each unit that was used in main mode. Range just indicates what the expected link values will be so as to proportion the icon changes appropriately among subranges.

4.7.4 Text Mode – displaying text

There may be times that for documentation or publication purposes you may want to put text on the display panel along with the network. Selecting "Text" on the mode panel puts you in text mode which allows you to do this (see Figure 4.4). Once in text mode you simply click left anywhere on the display panel where you want your text to start. A black rectangle will appear meaning you can start typing characters and they will then appear on the display. You can use the delete or backspace key to fix errors and the Return key will put you on the next "line". However each line of text you enter will actually be treated as a separate text "object" that can be manipulated (i.e. moved or deleted) independently.
Figure 4.4: GI tool window: Text mode
Also any significant mouse action, such as leaving the window or pressing another button, will create a separate object for the text entered so far and you will have to click left again to set the position for another text string. The longest possible single text object is 80 characters. If you type in a string longer than 80 characters, GI will automatically break it up into two or more text strings. You will normally be unaware that this happened unless you try to move or delete the string in which case the fact that it is not really a single object will become apparent.

The control panel in text mode has just one prompt — specifying the text font. Thus you can put text on the display in a variety of fonts (in fact each text object can be in a different font). You have to remember to specify the font before you set the text position as leaving the window to change font will necessitate setting the text position again. Also one you start typing in a text object, there is no interactive way (yet) to change its font. The default font is just your workstation default font and will be used if you don’t specify anything or the font you specify cannot be found. Just about any font the system supports can be used. The default directory used is “/usr/lib/fonts/fixedwidthfonts/” so if you want a font in that directory you only need specify the file name of the font. For fonts in other directories you have to fully qualify the font file name. By the way, the FONT prompt has three lines; should you run out of space on the current line, typed in characters will automatically be continued on the next. A warning regarding using variable pitched fonts: The SunOS 2.0 font structure did not allow efficient manipulation of variable pitched fonts (this was changed in 3.0). Thus selecting a variable pitched font will produce funny-looking spacing while you are typing the text in. However a RESHOW will put it back together with the proper spacing.

Text objects can be moved about the display manually by clicking the middle mouse button down over the text object. This will cause the cursor to change into a “grab” icon. Then moving the mouse (while keeping the middle button down) will cause the text object to track the “grab” cursor until you release the middle mouse button. Any screen damage caused by dragging text objects over other objects can be cleaned up with a RESHOW.

Text objects can be removed by clicking the right mouse button over them. A warning will then appear on the message panel asking you to confirm the operation by pressing the right mouse button again which then will (permanently) delete the text. Thus you need to click “right” twice over a text object to delete it.

If logging is enabled, all commands that create, delete or move text objects are also written to the log file allowing that text to recreated and positioned automatically when the log file is read in. In fact, if you wish to change fonts of text objects, one way to do it is to edit the log file items that created the text items with the new font before reading it. (See Section 4.10).

4.7.5 Draw Mode – line drawings

It may also be useful for you to be able to draw objects on the screen. “Draw” mode (see Figure 4.5) allows you to do this in one of three ways. You can draw “Line” objects consist-
Figure 4.5: GI tool window: Draw mode
ing of connected straight line segments box objects consisting of rectangles, or "bounding boxes". Select which one you want by clicking over the "Lines", "Boxes" or "Bounding boxes" switches of the TYPE prompt on the control panel. If you are drawing "Lines", click and release the left mouse button on the display panel where you want to start drawing. This will position a dot (vertex) at the cursor position. Moving the mouse will cause a line segment to emanate from that vertex to the current cursor position. Clicking and releasing the left button again will cause a new vertex to be placed at the cursor and a line segment drawn between the original vertex and the new one. Moving the mouse will then cause a new line segment to emanate from the new vertex to wherever the cursor is. By continuing to move the cursor and pressing the left mouse button you can create a line drawing of almost any complexity. To stop drawing just click left twice in the same place.

Boxes are drawn similarly except that only two diagonal corners of the box need to be specified. Clicking left and releasing will create one corner; moving the mouse will then cause a box to emanate from that corner to the current position of the cursor. Clicking down and releasing the left button will then set the other corner and the box object will have been created. Bounding boxes are drawn just like the "regular" boxes described in the previous paragraph. The only difference between them is what happens when you move them (see below).

Boxes and line drawings are treated similarly to text items. You can move them around with the middle mouse button (but you must "grab" them at a vertex) or delete them with the right button (again you must be near a vertex). Moving a normal box will only move that box. Moving a "bounding box" will move it as well as all the graphic objects contained within it. You will know when you are moving a bounding box as opposed to a regular box by the fact that as soon as you "grab" a bounding box, all the objects (except for other bounding boxes) will immediately "disappear" until you "drop" that box. Normally a bounding box only "contains" objects that are completely inside of it. The exception is that it will also contain all other bounding boxes that simply overlap it (as well as the objects they contain). Thus by "chaining" several bounding boxes together, you can move move an arbitrarily complex section of the display in one piece.

If logging is enabled, commands that generate, move or delete these drawn objects are written to the log file. GI limits each line object to no more than 10 vertices and will automatically break up objects you try to draw that are larger than this into two or more separate objects. Like text objects, if you exceed this limit, you will ordinarily be unaware of it until you try to move or delete the object(s).

4.7.6 Custom Mode – customizing mouse buttons

Custom mode is a little more complicated, but if you do a lot of simulation work, learning how to use it may save you a lot of time. When first enter into custom mode, the mouse buttons have the same actions as they do in Main mode. However in Custom mode you can override these actions with commands specified by you. The control panel has 6 prompts with little icons in front of them that (are supposed to) look like mice (see Figure 4.6). The
Figure 4.6: GI tool window: Custom mode
top three are for down button actions (left, middle and right) and the bottom three are for up (release) button actions. If you don’t define anything for a button (leave it "null or blank) it acts just like it does in Main mode. However if you do specify a command, then when that button action occurs while the cursor is over the display panel, that command will be executed just as if it were typed into the command panel. Thus what custom mode allows you to do is map GI or simulator commands to mouse buttons. For example, if you do a lot of resetting of your network, you can map the simulator command “reset” to the left mouse button. Then every time you press the left mouse button over the display panel, the simulator will do a network reset. The mouse buttons only act that way while you are in custom mode; we you go back to any of the other modes, the mouse buttons revert to their old meanings.

In order to make your customized commands more useful, you are allowed to create commands with symbolic arguments that are filled in at the time the mouse button is pushed and whose values depend on where the mouse is on the display panel. For now there are three substitution arguments you can use: $u, $x and $y. (Future releases may have more, so avoid using “$” names for anything). They have the following meanings and values:

- $u: returns the unit index of the unit icon (if any) underneath the mouse cursor. If there is no unit icon underneath the cursor the command will not be executed.
- $x: returns the x (horizontal) pixel coordinate of the mouse cursor in display space.
- $y: returns the y (vertical) pixel coordinate of the mouse cursor in display space.

You can use as many substitution arguments as you wish, but they must each be a separate argument in the command. That is, they must be surrounded by white space. So for example, you could map the left button to the command “pot $u 1000”. Then every time (while in custom mode) you clicked the left mouse button over a unit icon, the potential of that unit would be changed to 1000.

You can specify multiple commands to be executed sequentially with the press of a button by separating the commands by a semicolon (“;”) surrounded by blanks on both sides. You can also specify a partial command on one button with the command continued on another by making the last argument on the first part of the command two dashes (“--”). This feature could be useful when you need to pick up substitution arguments from different parts of the screen (since the mouse can only be in one place at a time). For example, if you wanted to make a link between two units shown on the screen you could map the first part of the MakeLink command (which needs the unit index of one unit) to left button down and the latter part of the MakeLink command (which needs the unit index of the other unit) to another button action, say left button up. Then by placing the mouse cursor over a unit, pressing the left button down then moving the cursor to another unit (or even the same one) and letting the button up, a MakeLink command will be executed that makes a link between those two units.
The fact that there doesn't seem to be space for a command longer than than a couple of dozen characters for each mouse button is an illusion: there are actually two lines. When you get to the end of the line, your keystrokes will automatically be continued on the second line. When you get to the end of the second line, you can still keep typing (for up to about 120 characters) with characters at the front of the second line disappearing as you type characters that appear at the end of the line. Those characters that disappeared are still part of the command, they just don't show up in the prompt. Thus you will only be able to see the prefix and suffix of really long commands. However the entire command will still appear on the command panel when they are executed as well as in the log file if logging is enabled.

While commands that you map to mouse buttons are written to the log file when executed, the mappings themselves do not unless you specifically ask them to be. Once you have set up the buttons as you desire, you can write those mappings to the log file by clicking over the LOG DEFINITIONS button beneath the last button prompt. You may, for example, have several different mappings that you use, and want to write them to separate files so that you can set them up by executing a simulator “read” command. Obviously to make effective use of “Custom” mode you will have to become familiar with the actual simulator and GI commands. We assume you are already know the simulator commands; the GI commands are discussed in detail in section 4.10.

4.8 The Display Panel

The display panel is, of course, the raison d'être for GI. The whole purpose of all the other panels is really just to put this piece of pixel real estate to best use in displaying the salient features of your running network. Although the display panel doesn’t have any buttons or prompts per se, there are a number of things you can do using the mouse depending on what mode you are in.

First of all, you may have noticed that the display panel seems kind of puny for displaying more than a few dozen nodes. Never fear, stretching the GI window in the normal SunView fashion will expand the display panel in both directions to the limit of the screen. (And of course you could have made it larger initially by using the -Ws flag). In making it larger, you will notice that any “hidden units” (not a joke) that were outside the display, will appear automatically if you make the display panel large enough to encompass them. They were really there all along, they just were not within the current scope of the display panel. Another way to bring units that are currently outside the panel into view without making the panel larger is to “move” (translate) the panel itself. This is done by pressing the middle mouse button down on a piece of screen that does not have a object under it (otherwise you'll move the object and not the display). You should notice the cursor changing to the “grab” icon again, except in reverse image; this indicates that you have indeed grabbed the screen and not something else. Now (while still holding the middle button down) move the mouse in the direction that you want the display window to move (the display doesn’t
You will notice that as you move it, the **Origin** prompt on the control panel will change reflecting where the new origin will be. The display itself will not track the mouse (it would be too slow) but when you release the middle mouse button, the whole display (i.e. the object in it) will suddenly jump to reflect the new origin. GI remembers the amount and direction of the last "jump" and will translate the origin the same amount repeatedly if, while holding the middle button down, you press the click the right mouse button. You can similarly go in the opposite direction by pressing the left button. Moving this way is called "jumping" and is a convenient way to "scroll" through two dimensional display space.

The Origin coordinates always indicate where the current display panel "window" is located in something called "display space". The way to think about the relation between "display space", the display panel and the "Origin" is as follows: Display space is a Cartesian plane stretching (almost) infinitely in both directions. The display panel is always showing a finite portion of this plane, specifically the rectangle whose upper left hand corner is located at the origin coordinates in display space. Initially the Origin is "0 0" meaning the display is looking at the positive quadrant of display space. However through the actions just described, any point in display space can be viewed and the Origin coordinates indicate just where in display space the display panel is currently "looking". Usually you will not have to think much about the actual coordinates since most of the work of laying out and examining your network can be done using the marker and appropriate mouse actions.

What the mouse buttons do on the display panel depends on what mode you have GI in. Although most of the mouse functions have been covered when discussing the different control panel prompts, we'll try to summarize them here by mode. Note that custom mode is absent since you define those button actions yourself.

**LEFT BUTTON** (mark)

- **Main mode:** If the cursor is over a unit icon, causes the detailed information about that unit to be displayed on the info panel (at the column marked NEXT). If there is no unit there, it moves the marker to the cursor position and redisplays it if it had been made invisible. Everywhere else, it should have no effect.

- **Link mode:** If the cursor is over a unit icon, causes the detailed information about that unit to be displayed on the info panel (as in main mode). Otherwise, does nothing.

- **Text mode:** Marks the start position of a new text string. Text string is terminated by moving cursor outside the window or pressing any other mouse button.

- **Draw mode:** Marks individual vertices for line drawings or marks opposite corners of box drawings.

**MIDDLE BUTTON** (move)

In all modes (except in Custom mode if you have redefined it) the middle button is used to move objects around on the display screen or to move the display window itself around in
display space. You simply click down over the object (or the display window if no object is underneath) and move the mouse with the button depressed and then releasing the button when the object is positioned as required. “Jump” moves with the window can be made by holding the middle button down and then depressing the right or left buttons.

**RIGHT BUTTON (target/erase)**

- **Main mode:** If the cursor is over a unit icon, it causes that unit to be reverse imaged (marking it as the current target) and copies its unit index into the “target” field of the control panel WHAT prompt. Clicking over a marked unit, “unmarks” it (although its unit index remains in the target prompt). If the cursor is over the marker, it nondisplays the marker. Otherwise the right button does nothing.

- **Link mode:** If the cursor is over a unit icon, it causes that unit to be reverse imaged and marks it as the current target unit. This causes all other units on the display to immediately change to reflect the value of a link between them and the now marked target unit. Otherwise the button does nothing.

- **Text mode:** If over a text object, attempts to delete that text object but first issues a warning message requiring you to click the right button down again to confirm the delete.

- **Draw mode:** Similar to text mode; if over a vertex of a drawn object, attempts to delete that object after asking for a confirmational right button click.

### 4.9 The Info Panel

While the purpose of GI is to give a birds-eye view of the behavior of a large number of units at a time, it is sometimes necessary to focus in on one or a small number of units. That is the purpose of the info panel. It has a number of different columns each capable of displaying detailed textual information for a particular unit. Note that as the GI tool window is stretched horizontally, the number of columns available for this type of information increases up to a maximum of eight. Initially none of the columns displays any information for any unit. The way to “activate” one of the columns is to click left over a particular unit on the display panel in main or link mode (or issue an “info” command – see Section 4.10). The particular info column that will display that unit’s information is the one that has the “NEXT” icon reverse-imaged just below it. You can specify which column is to be “next” by clicking left over its NEXT icon. Otherwise the NEXT column will automatically circulate to the right. A column on the info panel that is displaying information for a particular unit will continue to “track” that unit as the simulation proceeds, updating itself just as the display panel does. You can clear the information for a particular column by clicking over its NEXT icon when its NEXT icon is reverse-imaged (i.e. click twice over its NEXT icon).
There are several things to note about the columnar information that is displayed. First is that most of the time you will notice that one of the values in each column will be bold-faced. This indicates the aspect of the unit that is currently being captured by the display panel if the unit is being displayed. Secondly, if the aspect for that unit is not Link/in or Link/out, then (normally) no values for these items will be shown. However if the unit is displaying link weight or you are in link mode then the “Link:” field will contain a link weight and an additional field will show up underneath the “Link:” item, labeled “Target” which will contain the name of the unit on the “other side” of the link as well as the site name (if specified). In addition, the label “Target” will have either a “>” or a “<” following it indicating if the link being shown is “to” or “from” the target.

4.10 GI command interface

As has been hinted at above, most everything you can do (with the exception of link mode) with the buttons and mouse actions, can also be done with commands. This section documents what those commands are and defines their syntax. These commands can be entered on the command panel, read in from a file, or even executed from user code. It is these commands that are built and then written to the log file when logging is enabled and allows you to recreate your display screen or simulation session. In fact, although the SHOW, CHANGE and ERASE commands are executed by pressing buttons on the control panel, they actually first create a command string which is executed by the same routine that reads commands from the command panel. (You may have noticed that a “gi” command appears on the command panel when you execute a SHOW, CHANGE or ERASE through the panel buttons). Thus, although you don’t need to know the syntax of the commands to use GI, that knowledge is necessary for creating or modifying a command file or for programming the mouse buttons in Custom mode.

All the commands follow the same format: They begin with “gi” (indicating that the command is for GI rather than the simulator) followed by an argument consisting of a single letter that specifies the particular command, followed by an argument list containing some number of positional arguments specific to that command. The arguments must be in the exact order indicated, and at least one blank must separate each argument. An argument having blanks as part of it must be surrounded by double quotes. Double quotes that are meant to be part of an argument which itself is double-quoted, must be doubled. If you are building these commands in a file (to be read in by GI using the “read” command) the commands must be separated by a line feed with exactly one command per line (commands cannot cross line boundaries).

4.10.1 Placing units on the graphics display

The first three commands discussed are the SHOW, CHANGE and ERASE commands which are grouped together because they partially share the same argument list definition:
SHOW:

\[ \text{gi s <who> <num> <what> <lrange> <hrange> <target> <image>} \]
\[ \text{<xstart> <ystart> [<xspace> <yspace> [<numrow>]]} \]

CHANGE:

\[ \text{gi c <who> <num> <what> <lrange> <hrange> <target> <image>} \]

ERASE:

\[ \text{gi e <who> <num>} \]

where the parameters in <> have the following syntax:

<who> indicates the starting unit for the command and has the same syntax as the WHO prompt on the control panel: either a unit number, name, type or set.

<num> is a decimal number specifying how many units are affected by the command (beginning with the units specified by <who>).

<what> is a character string indicating the aspect of the unit and must be either “P” (potential), “O” (output), “S” (state), “D” (data), “Li” (Link/in) or “Lo” (Link/out).

<lrange> is a decimal number (or floating point number, if you are using the floating point version of the simulator) specifying the lower bound of the range of values the selected aspect of the unit will assume.

<hrange> is a decimal number (or floating point number if you are using the floating point version of the simulator) specifying the upper bound of the range of values the selected aspect of the unit will assume.

<target> specifies the target unit if the aspect is Link/in or Link/out and has the same syntax as <who>. If the aspect is other than Link/in or Link/out, this parameter should be 0. If you wish to specify a site as well, append it to the target unit separated by “/”. Thus “Mom/apple..pie” specifies a link to unit “Mom” at site “apple..pie”.

<image> is either a decimal number from 1 to 6 indicating which default icon family to use (1=square, 2=circle, 3=triangle, 4=pentagon, 5=diamond and 6=grey-scale), or the file name of your own icon family if you are using custom icons. See Section 4.11.1 for more information about customized icons.

<xstart> and <ystart> are decimal numbers specifying the position in display space for the first icon if more than one will be displayed by this SHOW command.

<xspace> and <yspace> are decimal numbers specifying how many pixels you want to separate the columns and rows of icons if more than one unit is to be SHOWn. Does not need to be specified if only one unit is being SHOWn. If the spacing is made slightly
(less than the width or height of an icon) negative, the icons will end up overlapping each other. Making the spacing more negative (larger than the width or height of the icon), will lay out the icons in right-to-left and/or bottom-to-top order.

<numrow> is a decimal number indicating how many icons you want per row if SHOWing more than one icon. The default “max” indicates that all the icons should be placed on the same row and like <xspace> and <yspace>, does not need to be specified if not more than one unit is being SHOWn.

For example, the command:

```
  gi s input all P 1 99 0 3 -100 200 40 50 5
```

would attempt to SHOW the potentials of all remaining unshown “input” units using the triangle icons, with the first icon being displayed at coordinate (-100, 200) with 40 pixels separating each icon horizontally and 50 pixels separating each vertically and 5 icons per row. The expected range the potentials will take is from 1 to 99. Note that even though the target parameter is ignored it still needs to be specified (“0”) as a placeholder.

### 4.10.2 Drawing lines or boxes and adding text

The next set of commands are for creating and deleting text or drawn objects and moving those objects around on the display:

**DRAW:**

```
  gi d <#vertices> <x1> <y1> <x2> <y2> [<x3> <y3> ... <x10> <y10>]
```

draws an object consisting of connected line segments starting at display coordinate (x1, y1) and ending at (xn, yn) where n is between 2 and 10. The <#vertices> argument is a decimal number indicating the total number of vertices (and should thus be n) and will thus always be one more than the number of line segments drawn. Thus

```
  gi d 5 50 100 50 200 150 200 150 200 150 100 50 100
```

would draw a box with opposite corners at (50, 100) and (150, 200). Note that 5 vertices were needed to draw the box as a closed figure. The above command would draw a normal box. To draw a “bounding box” simply replace the <#vertices> argument (“5”) with the letter “b”. Note that a bounding box must have exactly five vertices specified, otherwise an error message will be issued. You should also make sure these vertices define an actual rectangle.

**TEXT:**

```
  gi t <string> <x1> <y1> [<font name>]
```
creates a text string consisting of <string> at display coordinates (x1, y1) using the named font. If <string> contains embedded blanks, then it should be surrounded by double quotes. Double quotes that are part of a quoted string should be doubled. If the font is in the directory /usr/lib/fonts/fixedwidthfonts/ only the file name of the font need be supplied, otherwise it must be fully qualified. If not specified or "*default", the workstation default font will be used. Thus

```
gi t "lucky units" 100 200 screen.i.14
```

will display the string *lucky units* beginning at location 100 200 using a 14 point italic font.

### 4.10.3 Moving and deleting objects

The following commands are used to move or delete icons, text, or drawn objects.

**MOVE:**

```
gi m <x1> <y1> <x2> <y2>
```

moves either an icon, text or drawn object located at display coordinates (x1, y1) to the location (x2, y2). If no object is at (x1, y1) then the command simply does nothing (an error message is sent to the message panel, though). If more than one object is at (x1, y1) then only one of them (indeterminately) will be moved. Note that for drawn objects, the (x1, y1) must be "near" (within 3 pixels) of one of the object's vertices.

**DELETE:**

```
gi x <x> <y>
```

deletes a drawn or text object at that display coordinate location (x, y) if there is one. If no text or drawn object is at that location, then the command does nothing but put up an error message on the message panel.

### 4.10.4 Simulating and updating the graphics display

**GO:**

```
gi g [<#steps> [<#update_steps>]]
```

causes the simulator to run the network for <#steps> steps, updating the display screen every <#update_steps>. Thus has the same function and analogous syntax to the "GO" button on the control panel. Thus
gi g 20 5

will run the network for 20 simulation steps, updating the display after every 5 steps. If <#update_steps> is not supplied, it defaults to 1, as does the <#steps> argument. Setting the <#update_steps> argument greater than 1 will allow the simulation to proceed faster, but the tradeoff of course is that you won’t see the unit icons change after every step.

4.10.5 Redisplaying

RESHOW:

gi r [<x> <y>]

redisplays the screen “from scratch” so to speak with possibly a different display space origin (given by <x> and <y>). This is equivalent to interactively using the “RESHOW” button on the control panel. It causes GI to erase the screen and completely rebuild it from scratch using the new display space origin coordinates (if given). Any “damage” on the screen should then get cleared up.

4.10.6 Displaying unit details

INFO:

gi i <x> <y> [<col>]

displays the detailed information for a unit with an icon at display space location (x, y) on the info panel in the column (1-8) indicated by the <col> argument. If <col> is not specified, the current “next” column as indicated on the info panel is used. Issuing this command is equivalent to clicking the left mouse button over the unit icon while in main or link mode. If currently no icon is at location (x, y) nothing happens except that a warning is issued to the message panel.

4.10.7 Mapping mouse buttons

SET BUTTON:

gi b <#button> "command_string"
maps the indicated mouse button to the specified command string for execution in custom mode. (A separate "set button" command is built for each of the six mouse buttons and written to the log file when the "LOG DEFINITIONS" button is selected in custom mode). The mouse buttons are numbered from 1 through 6 with the left, middle and right buttons (down) numbered 1 through 3 respectively, with numbers 4 through 6 assigned to release of the left, middle and right buttons. Example:

\[
\text{gi b 4 "pot \$u 1000 ; gi i \$x \$y 3"}
\]

sets the left mouse button release action to two commands to be issued sequentially: The first command sets the potential of the unit under the cursor to 1000 and the second then displays the info for that unit in the third column on the info panel.

4.10.8 Setting the log file

SET LOGGING:

\[
\text{gi l on | off [<log file name>]}\]

turns logging either on or off and optionally specifies a name for the log file. Example:

\[
\text{gi l on my.log}
\]

turns logging on (if it were not already on) and specifies that the log file name should be "my.log".

4.10.9 Restarting GI

RESTART:

\[
\text{restart [gi]}
\]

If you are finished with a particular network, and want to get rid of it and start working with another network (or another instantiation of the current one) you could quit GI and rerun the simulator object. However there's a simpler way that doesn't require you to exit the tool. Just as the simulator has a "restart" command to destroy the current network, GI also has a restart command — in fact it's the very same command. Typing "restart" by itself will reinitialize the GI tool and also the simulator and thus the network itself will be destroyed. Just like starting over. If you include the optional parameter "gi", then only GI will be reinitialized — the network itself will still exist. Note that this is one GI command that doesn't begin with "gi".
4.11 Advanced Features

You should now know enough to make effective interactive use of GI. There are some other sophisticated features available, some of them hidden, that may be useful to you. However for now, it's probably better for you to go out and practice what you have learned so far and come back to this section when you're pretty familiar with the basic operation of GI or when your curiosity is just killing you.

4.11.1 Creating and using your own icons

You may not like some or any of the default icons we have given you to display units of your network. No problem; there is a way to specify your own icons if you're willing to go through a little trouble. What you have to do is create a "family" of icon files (one icon for each subrange you want to be able to distinguish) using icontool (or anything that puts the pixel definitions in the same format). Then, on the main mode WHAT prompt, select the icon with the "?" in it — this indicates that you wish to supply your own icons. Then put the file name of the icon family into the "Name:" field. When the SHOW or CHANGE command is executed, it will look for the specified icon files, read them in and use them for displaying those unit icons. There are some restrictions and rules you have to know to successfully use customized icons. First of all the icons within a particular family have to be the same size. For SunOS 2.0 to use them, icons must have a width of some multiple of 16 pixels, but can be of any height. (SunOS 3.0 may be more flexible). So if you are using icontool you can use both the "icon" (64 x 64 pixels) and "cursor" (16 x 16 pixels) type icons. (The default polygonal icons GI supplies are of the "cursor" type). So one thing customizing icons gives you is control over the size of the icons for displaying units in your network. There is also a naming convention for the files you are going to put the various icons into. The file names for a particular icon "family" should be of the format yourname.# where "#" is numbered from 0 through however many icons are in that family.

For example, say you wanted a set of smiley face icons to represent the output of some units and needed to be able to distinguish 4 different levels of output values. You could then use icontool to create 5 different icons named, say, funny.0, funny.1, funny.2, funny.3 and funny.4. The reason you need 5 icons to represent 4 ranges is that funny.1 through funny.4 will be used to divide up the output range (whatever it is) with funny.0 used only to indicate a value of exactly 0. Thus if you set the output range to be -30 to 70, funny.1 would be used for values (-infinity) to (-6), funny.2 for (-5) to (19), funny.3 for (20) to (44), and funny.4 for (45) to (+infinity). However for the "special" value of 0, funny.0 would be used. If you didn't want this special treatment of the 0 value, you could just make funny.0 look exactly like funny.2. When running GI, select the "?" icon and put the name "funny" into the "Name:" prompt that will then appear. Now doing a SHOW or CHANGE will result in those "funny" icons being used instead of one of the default polygonal ones. You can have (almost) any number of different icon families at a time (the size of memory is the only limitation). You can specify the same family many different times in a GI session.
without any additional memory overhead: GI remembers each icon family it loads in and will search through that list first before reading them from the file system. There is also no (practical) limit on the number of icons that can be in a family (i.e. the number of subranges). If you want you could specify a separate icon for every discrete value between -1000 and 1000 by creating an icon family (maybe tribe would be a better description) with 2001 members.

4.11.2 The programming interface for GI functions

The normal way one builds a network using the simulator is to create a C function that is callable from the simulator command interface that will build the network, possibly using parameters passed along with the call. From your callable C function you can then call many of the simulator functions directly. Frequent users of the simulator often exploit this to good advantage, for example, by dynamically creating and setting input potentials for a particular network simulation. What makes this technique so useful is

1. the ability to "call" your own C functions from the command interface, and
2. the ability to call the simulator functions from C code.

The point of this section is that you may want to do the same thing with the GI interface. For example you may want to have the job of displaying your network done dynamically by your own C code rather than interactively (which is time-consuming and error-prone) or by reading in a command file (which is inflexible). To allow you to do this, GI provides a programming interface to many of its functions, but in a different fashion from the simulator. Rather than provide and document the calling sequences to actual GI functions, we chose to limit all (documented) programming interfaces to a single routine: gi_command. It takes a single argument – a character pointer – and assumes that it points at a command string to be executed. If the command string begins with the argument gi then it tries to execute the command itself, otherwise it sends the command to the simulator for execution. Thus gi_command treats the command string passed to it as if it had been typed by you on the command panel. It returns to its caller an integer indicating whether the command was successful: 0 indicates (probable) success and -1 indicates (probable) error. In addition, if the command generated any error or confirmational messages, these will be displayed on the message panel. Thus if in your C code you specify:

```
return_code = gi_command("gi d 2 150 200 300 450");
```

executing that code (while in GI) will result in a drawn object being created that consists of a line segment from display space coordinates (150, 200) to (300, 450).

The advantages of this "single interface" approach is that it gives a consistent and easily remembered way to access most GI functions through code. Since gi_command is the routine
that internally processes all interactive commands, we can guarantee absolute consistency of function and error detection between the interactive and programming interfaces. It also provides “for free” program interfaces to all future GI functions implemented as interactive commands. In addition you only need to remember one syntax for both interactive and program commands. The disadvantage is that it probably takes a little more work to create the command string. However, the syntax of GI commands has purposely been made simple, terse and rigid; given C's powerful string commands (especially `sprint`) building the appropriate GI command in code should be fairly easy.

4.11.3 Using the log file

As noted previously, all commands typed into the command panel and executed, as well as most commands performed by mouse actions or panel buttons, are made into command strings and written to the log file (if logging is active). Logging is enabled and disabled by selecting the LOG: switch on the mode panel or via the log command. (See Section 4.6). As was mentioned there, the reason you might want to do this is to save yourself time and trouble the next time you are running that same network. Especially if you have spent a lot of time getting your network displayed exactly as you want it, you'll appreciate only having to do that once. However you have to be a little knowledgeable regarding the log file if you are to get maximum benefit from it. This section will try to give you some hints on possible log file usage and how, combined with the read command, it can be useful to you.

First of all, you should know exactly what commands GI has the capability of logging in the first place. To begin, basically all commands that affect the appearance of the display panel are logged. Thus all SHOW, ERASE and CHANGE commands are logged. Similarly commands are generated and logged whenever you create, move or delete a text or drawn object. “GO” and “RESHOW” commands are logged as well, since their execution has the potential to change the display panel. Also an “info” command is logged whenever you click left over a unit icon to display its values on the info panel. Finally, any command at all that you type directly into the command panel and execute, whether it be a simulator command, a GI command or a call to your own function, will be logged. This includes commands executed by mouse actions in custom mode, if those mouse actions are mapped to commands. If such commands have substitution parameters, they will be logged after those parameters have been resolved.

You should also know what commands are not logged. None of the mouse actions for displaying Links in link mode are logged. This was because it was felt that such commands are really only useful interactively - thus no “show links” command exists. (This may change at a later release). Also setting up the custom buttons will not automatically write out a “set button” command unless you explicitly request it by selecting the “SET DEFINITIONS” button. There is also the special case of the “read” command. Executing a read command may cause a number of other commands to be executed – namely those in the read file. Although the initial “read” command is logged, the commands executed as
a result of that read will not be logged. The reason is that, if you think about it, reading in of the subsequently created log file would execute all those commands twice; once when the read command is executed, and then again for each individual command that was in the read file. Thus only the bottom level "read" command is logged. Also not logged are commands that simply sets operational characteristics of the GI tool, for example, enabling logging or switching modes.

The obvious way to use a log file created during a previous session, is to use the simulator "read" command to read that list of commands in again, thereby "replaying" the GI session. Since the log file was a record of anything significant that happened, you should be able to recreate the session (almost) exactly. Since, strictly speaking, the "read" command is a simulator command, you may wonder what the simulator is going to do with all the gi commands since it doesn't recognize them. The answer is that in reality the simulator doesn't ever see any of the gi commands because when GI is active, the "read" command is actually processed by GI itself. What GI does is read each command from the read file and if it starts with a gi (or if it is another "read") it executes it itself, otherwise it passes it to the simulator for processing. Not only does this solve the problem of recording both simulator and GI commands in the same file, it also will allow you to use your command files stored on the Sun file system when the parallel simulator is running on the Butterfly Multiprocessor.

You can add comments to the log file by making sure the first non-blank character of the line is a "#". The read command (if executed from GI) will ignore the rest of the line. Similarly blank lines can be inserted for readability.

When the GI tool first comes up, by default logging is always enabled and the default log file is named gi.log. The first thing to remember is that anything that was previously in gi.log before GI was started, will get cleared out and rewritten after the first command that is logged. So anything that gets saved in gi.log in the current session will be erased the next time you run gi.log unless you do something. There are several ways to deal with this:

1. Once you have started the GI session, immediately change the name of the log file to something else; this will prevent you from accidentally clobbering it next time, or

2. After finishing your GI session, remember to move or copy the contents of gi.log to another file. Obviously solution (1) is safer but requires some forethought.

The next thing about using the log file is to make sure that what gets put into the log file is exactly what you want. For example, if you want the log file to contain only network set up and display commands, then you certainly don't want "go", "reshow", or simulator commands in the log file when you read it back in. On the other hand, you may want the log file to contain only those kinds of commands, that is, commands that run the simulation rather than set up the display. Or you may just want a log file to contain specific commands, such as ones that set up the custom mode mouse buttons. Again there are several ways to
4.12. MULTIPLE UNIT VIEWS

accomplish this. The first is to keep in mind the kind of a log file you wish to create and simply turn logging on or off at the appropriate times. As long as you don't rename the log file, turning it on and off during a single session will not destroy its contents; it will just act like a command accumulator. On the other hand, it you don't want to think about that kind of stuff during the session, you can simply record everything and later on just edit out all the commands you don't want. A third option is to use a file processor like awk or grep to filter the commands you want from the log file.

4.12 Multiple unit views

GI believes fairly strongly in the "one man—one vote" principle, paraphrased as "one unit, one view". That is, as you may have noticed, GI doesn't easily let you create more than one icon for a particular network unit. GI tries to prevents this by checking, whenever a SHOW command is issued, if the unit index of the unit you are requesting be SHOWn is already displayed. If it is, GI will normally ignore your request to display another icon for that unit. However there are many situations where having more than one icon per unit could be useful. If you had more than one view you could display several aspects of a single unit simultaneously, for example, its potential, output and state. Or you could show the same unit in different parts of display. The major problem with multiple unit views is finding a good way to specify which one of several that exist when building a CHANGE or ERASE command. We weren't able to come up with clean way ourselves as a matter of fact, but we did want to allow multiple unit views. Thus the method we have come up with is a bit awkward, but until we come up with something better, here's how it works:

If you already have displayed an icon for a particular unit, and you want to create an additional icon for it, simply prefix the backslash character (\") to however you designate that unit in the WHO prompt of the SHOW command. When processing the SHOW command, if the first character of the unit designation is a backslash, GI will skip doing the checks it normally does that make sure the unit is not already displayed. It also internally marks the new icon as being an "auxiliary" icon, as distinguished from the "primary" view of that unit that was first created. Thus you can have as many "auxiliary" icons as you want for a particular unit, but only one "primary" icon. The difference between "primary" and "auxiliary" icons will only become noticeable when you do a CHANGE or ERASE command. That is because the backslash prefix is valid on these commands as well; and necessary if you want to change or erase an auxiliary unit. To change or erase a primary icon, you do nothing different since the absence of the backslash in the name will cause GI to look only at the primary icons for that unit. On the other hand, if you wish to change or erase an auxiliary icon, you need to use the backslash prefix in the name so that GI only looks for icons marked as auxiliary. Unfortunately a problem arises if you have more than one auxiliary icon for a unit, and wish to change or erase just one of them. Since GI isn't able to distinguish between more than one auxiliary unit, it will just change or erase the first one it finds on the chain – which may or may not be the one you had in mind. There
is currently no good solution to this, except to caution you to set up any multiple auxiliary icons with care, since fixing them later may prove a little frustrating.

4.13 Performance Hints

Although for small, simple networks performance will probably not be a problem for you, large networks, or those that require many thousands of simulation steps, may tax your patience. Although GI has been designed to be fairly efficient, just the fact that it is writing out graphics commands and queries the status of all the units after every step, means it that it uses a fair amount of machine cycles. So if you are running a simulation session with GI where performance is important, there are a few things you can do that will help GI to run faster.

One, of course is not to require GI to update the display after every simulation step. Especially if your network changes rather slowly anyway, you may well be able to get by with only updating the display every fifth or tenth simulation step. You control how often GI updates its display panel via the “update steps:” parameter on the GO command. This will reduce GI overhead to zero between those simulation steps that don’t require an update.

Another way to reduce GI overhead is by restricting the display panel to just those units you are interested in watching. Say, for example, you have a network of 2000 units but for the particular simulation, you are only interested in the 200 that make up the “learning” layer. If all 2000 units are on the visible display panel, then every simulation step that requires an update will force GI to do 2000 units of work. On the other hand, by moving the display window or compressing the tool window to focus in on just the 200 of interest, you can cut GI’s overhead by 90%

You can also improve performance through optimizing the number of subranges or, equivalently, the size of the range for the aspect being tracked by an icon family. Since a significant amount of GI overhead results just from having to write a new icon to the display, GI only writes a new icon when absolutely necessary. That is, even if a unit’s tracked value changes at a simulation step, if the changed value is not outside the subrange of the currently displayed icon, GI takes care to not superfluously redisplay that same icon. So by coordinating the displayed range with the number of subranges you are actually interested in, you can cut down on the number of times GI has to write out a new icon. For example, let’s say you are really only interested in distinguishing when a units output is below or above 500; you don’t really care if it’s 250 or 400. If you use the default ranges (-1000, 1000) and the default icons (of which there are 20), then every time the value of such a unit changes by 100, a new icon will have to be displayed. A better strategy would be to change the range to (-5000, 5000) in which case a icon display would be generated
4.14. IMPLEMENTATION DETAILS

only when the unit changes from below 500 to above. Or equivalently, you could design
and load in an icon family that only had 3 members – one for 0, one for 0-500 and one for
500-1000 – and then set the range to (0, 1000).

4.14 Implementation Details

This section is directed to the programmer who may need to understand the working of the
GI interface code. Although the code itself is well documented at the module level, this text
is meant to provide a high level discussion of the underlying principles and organization of
the software package.

4.14.1 Structural Organization

The GI code is distributed among more than a dozen source files. The primary organization
revolves around the panel. There are six panels and thus six of the source files (obviously
named) contain the code necessary to implement the panel itself as well as most of the
routines used to access the panel and its data as a unit. In addition there are several other
source files whose organization revolves around function:

main.c contains the mainline (a misnomer), that is the entry point for setting up the GI
tool to begin with. This file actually sets up the tool windows and then passes control
to the SunView select mechanism.

update.c contains the routines used to update the display after something has happened
(such as a simulation step) and it is necessary to update the display panel with the
new information for the displayed unit icons.

show.c contains the routines that perform the SHOW, CHANGE and ERASE functions
that manipulate the display panel in terms of unit icons being displayed, changed or
erased.

misc.c contains various miscellaneous routines used by several of the other modules. They
have all been put in this one place for convenience since they don’t really “belong” to
any one particular function.

gisim.c contains all the routines that access routines or data structures of the simulator
itself. All interfaces to the simulator by GI routines should go through a routine in this
file. We wish to segregate these simulator interface routines so that we can localize
changes to the simulator interface (for example, the parallel version of the simulator
on the Butterfly) to routines in this one file, rather than having to make changes (and
have separate versions of the code) everywhere.
gistart.c contains the interface to the entry point. This routine and this routine alone is processed by the nametable function and the presence of the function name gi_start in the simulator’s function table tells the simulator that gi has been compiled into the object code.

4.14.2 External Variables and Functions

GI uses many external (global) variables and functions. Most of these are defined in globals.c and externalized in externs.h. However some globals that are closely tied to certain routines are defined in those routines and externalized in separate header files for that file. For example, most of the SunView Panel item variables that are externalized are defined in the corresponding *.panel.c file and externalized in *.panel.h.

Note that all external GI variables and functions begin with the suffix gi_. This is to enable the user, whose functions will be linked with the GI code, to avoid external name clashes. Any new GI external variables and functions should continue to follow this convention.

Most of the global defines and structure definitions are contained in the file macros.h which is an include in every source file. Anything added or changed in this file will be globally available to all the GI modules, but by the same token, will necessitate a complete recompile of all the GI sources.

4.14.3 Data Structures

The most complex data structures in GI are used to maintain the objects on the display panel. There are basically three kinds of graphic objects that can appear on the display:

grobj graphic objects for an icon that is tracking some aspect of a network unit. There is one of these for every icon on the display panel, maintained as a separate structure on one of two doubly linked lists.

taxobj graphic object for a text item that appears on the display. There is a separate taxobj structure for each line of text on the display and they are all linked together on their own doubly linked list.

drobj graphic objects for a drawn object consisting of 2–10 vertices joined by line segments. There is a separate drobj structure for every connected drawn object and they too are linked together on their own doubly linked list.

In order to optimize performance, two list of grobjs are always maintained. One contains all the grobjs that are currently displayed. The list header for this chain is called gi_marker because the marker icon itself resides inside the header structure. The other chain contains all the other grobjs that exist but are currently outside of the display panel window. The
4.14. IMPLEMENTATION DETAILS

header for this structure is named gi_off_display and its structure is not used for anything currently. Any time the display panel changes size or the panel window is moved or a reshow with a new origin is done, these two chains have to be updated so that each contains the right grobjs for the new window. The routines that accomplish this are located in update.c source file.

Each of the grobj structure point to structures that maintain the information on the icon family that control the appearance of the icon relative to the aspect of the unit it is tracking. These icon structures (called gricons) contain pointers to array of pixrect pointer that make up the icon family as well as variables indicating how many members in the family and what the dimensions of the icons are. It also contains a pointer to the icon family name if this structure contains a user-defined icon family. These gricons themselves are chained together on a singly linked list in order to be able to search for previously defined user icons.

All strings, vertices (of drawn objects) and fonts are maintained in separate areas using a similar strategy. For example string space is maintained in any number of separate buffers all of which are allocated dynamically as STRING_SPACE_SIZE bytes. Initially no string space is allocated. As soon as the first string needs to be stored a chunk is allocated and pointers maintained to the free area. When the chunk is used up, another buffer is allocated and so on. Since strings once stored always need to be retained these buffers are never deallocated until the GI exits. A similar strategy is followed for maintaining vertices and font definitions.

The only other data structure of consequence is the one (info_unit) that maintains the columns of information on the info panel. This is maintained as an array of 8 (MAX_INFO_COLS) structures which keep track of what unit is being tracked by the column as well as the current values for all aspects of the unit that appear in the column.

4.14.4 Processing Strategy

Most of the complexity of the code revolves around trying to be as smart as possible in NOT writing graphics object to the screen unless it is absolutely necessary. That is the reason two chains of grobjs are maintained – the objects on the off.display chain are not even looked at during the simulation. However even the grobjs on the displayed chain have a number of flags designed to increase the efficiency of the display actions. There is also one other important consideration in the update strategy. That is that the actions of getting the new values for the units after a simulation step is completely separate from the eventual display of the unit on the display panel. It is crucial to performance that these actions be strictly segregated for the version that will run on the butterfly. This is because all the values for all the units on the display are gotten at the SAME time (in one large buffer) in order to take advantage of the parallelism in the butterfly version of the simulator. Thus instead of looking at each unit on the chain, getting its value and then displaying it, getting the next unit’s value, displaying it, etc., we choose to get ALL the unit values first and then display them all. Although this requires going through the display chain twice (once to get
CHAPTER 4. GRAPHICS INTERFACE USER MANUAL

The values, then to display them), this is a small price to pay for the expected efficiencies to be gotten with the Butterfly.

The flags for controlling what needs to be gotten and/or reshowed and what doesn't is contained in \texttt{gi\_reshow\_flag} which has a number of bit flags which need to be set or not depending on what kind of display behavior is called for. The \texttt{gi\_reshow\_flag} are set in various places of the code and checked and reset in the routine \texttt{gi\_reshow}. The bit flags in \texttt{gi\_reshow\_flag} have the following meaning and should be strictly adhered to:

\textbf{RESHOW\_NEEDED} indicates that there is something for \texttt{gi\_reshow} to do (i.e. there is at least one unit icon that may have to be updated). This flag is set on by any routine which through its processing knows that an object on one of the display chains may need to be reshowed.

\textbf{RESHOW\_ALL} indicates that all units on the display must be redisplayed regardless of the settings of their switches. This is set, for example, when the display window moves, necessitating the redisplay of everything on the display panel at different coordinates.

\textbf{SOME\_VALS\_NEEDED} indicates that some grobjs on the display chain need to have their values updated via the simulator. These grobjs are indicated by the fact that their \texttt{VALUE\_OK} flags are off.

\textbf{ALL\_VALS\_NEEDED} indicates that ALL grobjs on the display need to have their values updated regardless of the settings of their \texttt{VALUE\_OK} switch. This flag would be set on when the simulator has run a simulation step, or indeed if any command is sent to the simulator.

\textbf{SOME\_LINKS\_NEEDED} analogous to \textbf{SOME\_VALS\_NEEDED} except it only applies in link mode to the Link value that the unit icon is displaying. It indicates that at least one grobj (with \texttt{LINK\_OK} off) is on the display chain which needs to have its link value gotten from the simulator.

\textbf{ALL\_LINKS\_NEEDED} analogous to \textbf{ALL\_VALS\_NEEDED} except it only applies in link mode to the unit icon Link values. Indicates that ALL link values must be gotten from the simulator regardless of what the individual grobj flags indicate.

\textbf{CLEAR\_NEEDED} indicates that \texttt{gi\_reshow} should first clear the display panel before rewriting it. Caused when the display is moved to another part of display space or when the RESHOW button is pressed. Should thus take care of any “damage” on the screen.

What values have to be gotten and when a display of an icon is necessary is determined by the bit flags in each gricon. The critical flags and what they mean are as follows:

\textbf{DISPLAYED} on – indicates that the unit is currently displayed, and thus does not need to be redisplayed unless the display moves or its value changes
4.14. IMPLEMENTATION DETAILS

**VALUE_OK** on – indicates that the value in the `val` field is current and thus there is no need to go to the simulator. (Unless `ALL_VALUES_NEEDED` flag is on).

**VALUE_CHG** on – indicates that the value currently in the `val` field is correct but has been changed from the last time the unit has been displayed. Thus suggests that a new `icon.index` and thus a different icon may be necessary to reflect the value of the unit on the display screen.

**LINK_OK** analogous to `VALUE_OK` except only applies in link mode to the `link.val` field.

**LINK_CHG** analogous to `VALUE_CHG` except only applies in link mode to the `link.val` and `link.index` fields.

4.14.5 Link Mode processing

Each grobj maintains two independent values: one for the current aspect of the unit it is tracking, and one maintained explicitly for link mode representing a link weight between it and the current `target` unit/site. Since there is only one target for all units in link mode, the target unit and site are maintained in global variables `gi_cur_link_target`, `gi_cur_link_site` and `gi_cut_link_ptr`. However if a unit is specifically tracking a particular aspect outside of link mode, then the target information is maintained locally within the grobj itself.

Currently there is no support for “commands” for displaying links in link mode, mostly because it is felt that link mode is primarily an interactive task and doesn’t have much value in command mode. However this could change in the future.

4.14.6 Floating Point version

Version 4.0 of GI converted floating point values immediately to integer values and maintained them internally as such. Thus the fractional part of any unit aspect was lost and it was impossible to work within small ranges of values like \([-1.0,1.0]\). For version 4.1, GI was modified to provide “true” floating point support to match the capabilities of the simulator. This was done by using the defined type `FLINT` for values that could be either integer or float (depending on the simulator) and “#ifdef FSIM” statements for conditional compilation wherever the code had to be able to handle either type.
4.15 XGI: The X11 Interface

Synopsis: The X11 graphical interface (henceforth XGI) evolved out of the original GI interface described in the previous sections of this chapter. This section assumes familiarity with those preceding sections. Only the differences between the original interface and XGI are described. The two most important differences are that XGI allows multiple display panels whereas GI allows only one; and XGI provides customizable popup menus.

XGI was designed — if that is not too grand a word for it — to be used with large, highly structured networks, in which groups of units form modules within larger sub-systems which are themselves part of the overall network architecture. Experience with GI led us to the conclusion that a single cartesian display space is insufficient for complex networks. For intuitive display of structured networks, we need tools with which to construct structured displays. XGI incorporates two changes to facilitate structured display: multiple display windows; and customizable popup menus. Section 4.15.5 illustrates how these two facilities can be combined to form a rudimentary object-oriented graphical display.

The X11 interface consists of a control window, an arbitrary number of display windows, and a layout window. Various temporary popup windows can also appear. The interface runs under uwm and, perhaps, other window managers. When the simulator is run, the control window, a single display window, and the layout window appear on the screen. At all times, one of the display windows is the current display.

There are two types of user interaction with XGI. First, via the mouse and keyboard the user can layout the icons and control simulation. This type of interaction is described in the immediately following sections. Second, the user can layout icons and control simulation from within program code. This type of interaction is further sub-divided into: use of the GI command interface (see Section 4.10); and use of the functional programming interface. Control of XGI from within program code is described in section 4.15.6.

4.15.1 Control Window

On startup, an initialization message is displayed and then the control window popped up. Figure 4.7 shows the control window, which contains four areas. The dividing lines between the areas can be moved by means of the small black rectangular grips at the right of the window. The topmost panel controls the mode of the current display. The next panel down shows the current clock step, the simulation parameters, and several control buttons. The third panel shows messages from the simulator and the interface, the default message indicating what the mouse buttons control. The fourth panel shows a scrollable history of commands executed, and a prompt ‘->’ for user input.

The buttons in the mode panel control the mode of the current-display window. Different display windows may be in different modes. The button corresponding to the mode of the
current-display is stippled. Selecting a new current-display (see Section 4.15.2) may change which button is stippled.

The simulation control panel shows the current simulation clock value, a GO button, and several other control buttons. When clicked, the GO button causes the simulator to execute the number of steps specified at the steps type-in, with updates of the display window(s) at the increments specified in the update type-in. The NEW DISPLAY button pops up a new display window. The RESHOW DISPLAYS button causes all display windows to be reshown. The LOGFILE button pops up a dialog box which controls whether commands are logged in a file, and if so, what the file is named. The NO ECHO button causes commands to not be echoed in the scrolled history box, which can significantly speed network construction and execution if the gi.command() programming interface is used (see Section 4.10). The button label changes to ECHO ON when clicked, so that command echoing can be reinstated. The HELP button pops up an information window. The QUIT button causes the logfile to be closed and the simulator exited.

The messages which appear in the message panel are of two types: informative, and errors. The latter are distinguished by being preceded with “ERR:”. Often the message:

Check simulator window for information

will appear in the message panel. The simulator window is the window in which the simulator was originally invoked, which may be obscured by the control or display windows.

The command history is in the scrollable panel immediately above the input prompt. All commands typed in or sent to the gi.command() interface or invoked with mouse clicks are noted in this panel, unless the NO ECHO button has been clicked.
4.15.2 Display Windows

The display windows are modeled on the GI display panel described in section 4.8. There is always a current-display window. A display window may be selected as the current-display in several ways, the most reliable of which is to click the LAYOUT button in that display window. Figure 4.8 shows an example of a display window, with some smiley face icons.

A display window shows particular aspects of network activation and structure, in the form of graphical icons. Typically it is used to show unit activations or link weights. Each display window has a mode, set by the buttons on the CONTROL window. The mode of a display window can be changed to show unit activations or link weights, to add text or drawings to the display, or to enable custom menus. A display window contains two areas, separated by a horizontal line. The upper area contains control buttons while the lower area is used for displaying network state. Each display window has a name (in the top right corner), which can be changed by clicking on the RENAME button. The portion of the display space shown in the window can be changed with the JUMP button (read section 4.8 for a description of the concept of display space). The display may be redisplayed with the RESHOW button. The LAYOUT button pops up the layout window (see section 4.15.3), which controls where units are located and how units and links are displayed in MAIN, which text font is used in TEXT mode, and what type of figures are drawn in DRAW mode. The FREEZE button causes the display to not be updated. When clicked it changes its label to UNFREEZE so that upon a second click, updating of the display will be reinstated. The QUIT button destroys the display window.

In MAIN and LINK modes, left clicking on a unit which is displayed will cause a small
window to pop up, showing details of that unit. These information windows may also be frozen. Figure 4.9 shows an information window.

Arbitrary numbers of display windows may be created. They are independent, so that links between units in different windows are not shown. For maximum simulation speed without graphics updating, the FREEZE button should be used. An alternative is to iconify the display window, but this is not so effective because the X protocol calls are still made.

Mouse clicks events are dispatched at the end of each simulation step. Displays can be frozen and unfrozen many times during a long simulation. Dangerous user activity is possible, for example network construction during simulation. Caveat emptor!

4.15.2.1 Custom Icons

Custom icons are supported. XGI expects text files in the form of rows and columns of #’s and -’s, representing the on pixels and off pixels respectively. See the files “/example/xgi/icon/smiley.*” for samples. These files can be created by using the “bitmap” and “bmtoa” client distributed with the X Window System source code (X11/clients/bitmap). There is a shell script to help create icon families: “/bin/makeicon. Icon family file names should be as described in section 4.11.1 with the exception that an underscore ‘_’ is used in place of a dot ‘.’ – so an icon family called mir would be encoded in files mir.0, mir.1, mir.2, etc. Custom icons are loaded on demand when a display command requires them. XGI will look for the icon files in the current directory and in its subdirectory ./icon if it exists. Icons families in other directories can be specified by giving the full pathname. An icon family can be reloaded after editing, see section 4.15.6.

4.15.3 Layout Window

The layout window appears when the LAYOUT button in a display window is clicked (this window then becomes the current display window). Its size may need to be readjusted. The layout window information depends on the mode of the current display window:
CHAPTER 4. GRAPHICS INTERFACE USER MANUAL

WHAT

WHO: 0
HOW MANY: all

WHAT

Potential Output State Data

from: -1000 to: 1000

HOW

square circle triangle pentagon diamond grey

WHERE

start x: 5 start y: 5
space x: 5 space y: 5
units per row: max

SHOW CHANGE ERASE CLOSE

Figure 4.10: XGI Layout Window: Main mode

- MAIN MODE: The buttons control unit layout in the display window.
- LINK MODE: The buttons control link layout in the display window.
- TEXT MODE: The buttons control text font used in the display window.
- DRAW MODE: The buttons control drawing in the display window.
- CUSTOM MODE: A description of the custom mode menus is displayed.

Figure 4.11: XGI Layout Window: Text mode
The contents of the layout window change when the mode in the CONTROL window is changed. It does not automatically resize, so may need to be manually resized for all the buttons and text to be visible. Figure 4.10 shows the layout window in Main mode. It contains the same buttons and type-ins that are described for GI's control panel in section 4.7.2. In fact, the layout window is the same as GI's control panel, except in text mode and custom mode. Figure 4.11 shows the layout window in text mode: there are a number of buttons showing the fonts that may be selected. In custom mode, the layout window simply describes the action of the popup menus.

4.15.4 Custom Menus

In custom mode, there is a menu of actions associated with each mouse button in the display window. Figure 4.12 shows one of the menus. Menu A is associated with the left button, B with the middle button, and C with the right button. Initially the menu actions are all “unused”, meaning no action. Buttoning in the display window will cause the associated menu to popup. Then left-buttoning on a menu item will execute the action, middle-buttoning will popup a dialog box for editing the action, and right-buttoning will reset the action to “unused”. The dialog box for editing actions allows association of a simulator command with a menu item. The format for specifying the association is:

```
item-name : command
```

where the spaces before and after the : are necessary. Item-name can be of any length but only the first 10 characters are used. Command can be any command that could be typed to the input prompt. Figure 4.13 shows the dialog box, with the format for associating the action ‘call ShowInputs $u’ with the menu item ‘inputs’. In addition, as described in section 4.7.6, the variables $x, $y and $u can be used in a command. At action execution time,
Specify Item and Command, separated by a colon, e.g. list : I u all

inputs : call ShowInputs,u

CHANGE CANCEL

Figure 4.13: XGI Custom Menu Editing

these variables will be substituted with the values for the $x$ and $y$ locations of the mouse click that popped up the custom menu, and the index of the unit that is displayed at that location, if any. Command continuation, as described in section 4.7.6, is also retained across menu popups, allowing execution of a command that requires more than one unit index or location (for example, creating a link between two units). Clicking on the LOG MENU ITEMS button in the layout window will write the current custom menu item/command associations to the logfile, if there is one. Note that the custom menus are general across all display windows: one cannot have different custom menus in different displays.

4.15.5 Examples

Two examples are supplied in the directory `/example/xgi`. Examine the README file in directory for instructions. The first example uses custom icons, dynamic linking, and shows a simple method for linking your own X widgets with the simulator. Sample code used in this example is shown in Table 4.1.

The second example is a back-propagation network that learns to solve the 3 bit binary encoder problem. It illustrates the use of multiple display windows and implements a simple widget that allows you to control the three back-propagation parameters with scrollbar sliders, as the simulation is running. Section 4.15.7 shows how simple it is for users to integrate their own X widgets with the simulator.

Figure 4.14 shows a screen dump from a simulation in which XGI is used to implement a rudimentary object-oriented graphical interface. The large display panel named INPUT-FIELD shows a retinal array of locations. These are implemented as single units that perform no function in the network but store information for the custom menu actions to use. Associated with each location are sets of input units and feature units. The custom-menu items are set to call user-written functions which translate the location of a click in the input field and perform appropriate action. For example, the inputs item calls a function which pops up another display window, showing the input units at that location. Three of these display windows appear at the bottom left of Figure 4.14, corresponding to the three active locations in the input field. Similarly, the features item calls a function to pop up a display window showing the feature units at that location. Two feature-unit display windows are shown at the bottom right of Figure 4.14, and a third is shown iconified. The reset-loc item causes all units associated with the location to be reset to initial values. In this manner, a structured, object-oriented display is constructed with different display
Figure 4.14: Structured Display and Object Oriented Graphics

windows showing different modules or views of the network, and with the popup menu items performing functions dependent on what graphic object is buttoned.

4.15.6 Interacting with XGI from Program Code

As stated at the beginning of this description of XGI, the second major method of interaction with XGI is from user program code. This method encompasses two types of interaction: use of the gi_command() function to pass string commands to XGI, as described in section 4.10; and making calls on XGI functions directly. The latter is by far the most efficient method computationally. In the following subsections we describe how various effects may be achieved via these two methods. We omit description of the string commands that are described in section 4.10, including only the new string commands devised for XGI.

4.15.6.1 Creating Display Windows

\[
gi\_command("gi\_yc\ name\ xorig\ yorig\ width\ height");
\]

\[
gi\_make\_display(name, xorig, yorig, width, height)
    char * name;
    int xorig, yorig;
    int width, height;
\]

Make a display window named \textit{name}, of size \textit{width} \times \textit{height}, and pop it up with the top left hand corner at \((xorig, yorig)\). In the \textit{gi\_command()} version, default values are used for unspecified parameters.
4.15.6.2 Destroying Display Windows

\[ \text{gi_command(}"\text{gi yq name}"\text{);} \]

\[ \text{int gi_close_display_name(name)} \]

\[ \text{char * name;} \]

Destroys display window named \textit{name}. Function version returns 1 if successful, -1 if not.

\[ \text{gi_close_all_display_panels()} \]

Destroys all existing display windows, permanently.

\[ \text{gi_popdown_popups()} \]

Destroys all display and info windows and the layout window. This is the first step in exiting XGI.

4.15.6.3 Selecting the Current Display Window

\[ \text{gi_command(}"\text{gi ys name switch}"\text{);} \]

\[ \text{int gi_select_display_panel(name, switch)} \]

\[ \text{char * name;} \]

\[ \text{int switch;} \]

Sets display named \textit{name} to be the current display. If \textit{switch} is zero, then the CONTROL and LAYOUT windows are not updated. (If the display is selected merely to make some changes, after which another display is selected, then it is not necessary to change the layout window.) Function version returns 1 if successful, -1 otherwise.

4.15.6.4 Setting the Mode of the Current Display

Modes are indicated by integer values, with this mapping:

0:MAIN 1:LINK 2:TEXT 3:DRAW 4:CUSTOM
4.15. XGI: THE X11 INTERFACE

```c
int gi_select_display_mode(name, mode)
    char * name;
    int mode;

    Set the display named name to be in mode mode. Function version returns 1 if successful, -1 otherwise.
```

```c
int gi_set_display_mode_to_custom(flag)
    int flag;

    This function toggles the actions associated with mouse clicks in the current display window. If flag is 0, mouse clicks are dispatched to the normal functions (MAIN, LINK, TEXT and DRAW modes). If it is 1, mouse clicks in the window cause the custom menu popups to appear, as in CUSTOM mode.
```

### 4.15.6.5 Renaming Display Windows

```c
int gi_rename_display_name(current, new)
    char * current;
    char * new;

    Resets the display named current to be named new. Function version returns 1 if successful, -1 otherwise.
```

### 4.15.6.6 Freezing and Unfreezing

```c
int gi_freeze_display_name(name, frozen)
    char * name;
    int frozen;

    Freeze or unfreeze the display window named name. If frozen is 0, the window is unfrozen, otherwise it is frozen. The function version returns 1 if successful, -1 if not.
```
4.15.6.7 Reshowing

\texttt{gi\_reshow()}

Update the current display window appropriately by looking at the various flags to determine what kinds of updates to do so that only screen operations absolutely necessary to make things consistent are done. Controlled by \texttt{gi\_reshow\_flag}. See Section 4.14.4 for the coding of \texttt{gi\_reshow\_flag}.

\texttt{gi\_reshow\_all()}

Update all displays, according to the setting of \texttt{gi\_reshow\_flag}. Only repaints as necessary. See Section 4.14.4 for the coding of \texttt{gi\_reshow\_flag}.

\texttt{gi\_command(\textit{\texttt{gi yr name}})};

\texttt{int gi\_reshow\_display\_name(name)}

\texttt{char * name;}

Causes complete repainting of display window named \texttt{name}, whether or not it is frozen.

\texttt{gi\_command(\textit{\texttt{gi yr all}})};

\texttt{gi\_reshow\_all\_displays()}

Repaint all displays regardless of whether frozen.

\texttt{gi\_change\_origin(x\_displ, y\_displ)}

\texttt{int x\_displ, y\_displ;}

Moves the icons on the current display by \texttt{(x\_displ, y\_displ)}. Note that the y displacement is interpreted as positive meaning downward.

4.15.6.8 Showing, Changing and Erasing Unit Icons

These functions correspond to the \texttt{gi\_command()} strings \texttt{"gi s ..."}, \texttt{"gi c ..."}, and \texttt{"gi e ..."}, described in section 4.10.1. Calling these functions is much faster than passing the command through the string processing routines in \texttt{gi\_command()}, and does not incur the overhead of adding the string to the command history panel.
GiShowNumberedUnits(unit_index, num_units, unit_what, unit_lrange, unit_hrange, 
    target, image, where, spacing, num_cols)

    int unit_index; /* first unit (-ve => auxiliary)*/
    int num_units; /* how many */
    int unit_what; /* aspect */
    FLINT unit_lrange; /* low end of value range */
    FLINT unit_hrange; /* high end of value range */
    char * target; /* unit/site target for link in/out */
    char * image; /* name of icon to use */
    Vertex * where; /* where on display */
    Vertex * spacing; /* spacing between cols and rows */
    int num_cols; /* units per row */

GiShowNamedUnits(name, num_units, unit_what, unit_lrange, unit_hrange, 
    target, image, where, spacing, num_cols)

    char * name; /* initial \\ implies auxiliary aspect*/
    int num_units; /* number of units to show */
    ... etc, as above.

These two functions display one or more units on the current display panel. GiShowNumberedUnits shows units [unit_index,unit_index+num_units]. GiShowNamedUnits shows the first [num_units] described by [name], where name can be a type, set or unit name. The parameters are almost the same as those used in the "gi s ..." string that can be passed to gi.command(), described in section 4.10.1:

- **unit_index** - the absolute value of this parameter is the index of the first unit to display. If this parameter is positive, then the aspect to be displayed is primary. If it is negative, it is an auxiliary aspect (see Section 4.12). GiShowNumberedUnits only.

- **name** - specifies the name of the unit(s) to be displayed, and whether the aspect is primary or auxiliary. If it is primary, just the name is given. If it is auxiliary (see Section 4.12), the name is preceded by the character \\‘. GiShowNamedUnits only.

- **num_units** - the number of units to be displayed.

- **unit_what** - the aspect to be displayed. Aspects are defined as follows: 1: Potential; 2: Output; 3: State; 4: LinkIn; 5: LinkOut; 6: Data.

- **unit_lrange** - the lower bound on the range of values that the selected aspect may assume. This is used to determine which icon to display for a particular aspect value (see Section 4.7.2). FLINT is a type that is defined to be float in the floating point version and int in the integer version.

- **unit_hrange** - the upper bound on the range of values the selected aspect may assume. Values above the upper bound are treated as this value for display purposes.
• **target** - the unit/site target to use for the LinkIn or LinkOut aspects. This parameter is ignored for other aspects. See section 4.7.2 for a description of the format of this string.

• **image** - the name of the family of icons to use to display this aspect of the unit(s). See Section 4.15.2.1 for a description of how icon files are generated and named.

• **where** - the \((x,y)\) pixel location in display space where the first unit is to be displayed. The position of subsequent units depends on the spacing parameter. Vertex is a type defined as shown below, which is also used in other functions:

```c
typedef struct vtx
{
    int x, y;
} Vertex;
```

• **spacing** - the number of pixels between rows and columns of icons.

• **num_cols** - the number of units in a row. The layout pattern for a group of units is in a grid of num_cols columns and num_units/num_cols rows.

Returns 1 if successful, -1 if unsuccessful. If unsuccessful, then nothing is displayed. There is some error checking but not as much as in the “gi s ...” form.

GiChangeNumberedUnits(unit_index, num_units, unit_what, unit_lrange, unit_hrange, target, image)

These two functions change how units are displayed on the current display panel. See GiShowNumberedUnits above for a description of the parameters. This function can change the aspect, the value range, the icon, and the target for LinkIn and LinkOut. The specified units should already be displayed. Returns 1 if successful, -1 if not.
GiEraseNumberedUnits(unit_index, num_units)
    int unit_index;
    int num_units;

GiEraseNamedUnits(name, num_units)
    char * name;
    int num_units;

Erases \([num\_units]\) from the current display, commencing with unit\_index or from the
group described by name which is a set, type or unit name.

4.15.6.9 Drawing Lines, Boxes, Bounding Boxes, and Text

These functions correspond to the \texttt{gi...} commands "gi d ..." and "gi t ...", described
in section 4.10.2. See Section 4.15.6.8 for a description of the \texttt{Vertex} structure.

GiDrawLines(count, vertices)
    int count;
    Vertex * vertices;

Draws lines between count vertices. For \(n\) lines you specify \(n+1\) vertices. \texttt{vertices} is the
address of an array of \(count\) \texttt{Vertex} structures. If \texttt{count} is negative then there are assumed
to be 5 vertices describing a bounding box. Returns 1 if successful, -1 if not.

GiDrawText(text, where, font)
    char * text;
    Vertex * where
    char * font;

Prints the string in \texttt{text} at location \texttt{where} in font \texttt{font}. A \texttt{font} value of NULL means
use the default font. \texttt{font} should be the name of a font file in the default font directory
(usually /usr/lib/X11/fonts/misc), or a complete path name for a font file not in the default
directory. Returns 1 if successful, -1 if not.

4.15.6.10 Moving and Deleting Graphic Objects

These functions correspond to the \texttt{gi...} commands "gi m ..." and "gi x ...", described
in section 4.10.3. See Section 4.15.6.8 for a description of the \texttt{Vertex} structure.

GiMoveObject(from, to)
    Vertex * from;
    Vertex * to;
CHAPTER 4. GRAPHICS INTERFACE USER MANUAL

Moves the object currently at from to position to. Fails if no object is found at from. Returns 1 if successful, -1 if not.

GiDeleteObject(from)

Vertex from;

Deletes the object currently at from. Fails if no object is found at from. Returns 1 if successful, -1 if not.

4.15.6.11 Simulation, Control Flow and X Event Dispatching

gi_do_steps(steps_left, upd_steps)

int steps_left, upd_steps;

Execute steps_left simulation steps, updating the displays every upd_steps steps. May return early if gi.back_to.caller() is called.

gi_raise_control_panel()

Bring the control window to the top, and wait for all X events to be dispatched.

gi_end_session()

Exit program in a controlled fashion, closing files, etc.

gi_get_control()

Dispatch X events until gi.back_to.caller() is called. May be called by controlling programs when XGI is not processing events.

gi.back_to.caller()

Causes return to the topmost caller of gi.get.control().

int gi_process_events()

While there are pending X events, dispatch them. May return early if gi.back_to.caller() is called. Returns 1 if all pending X events were dispatched, 0 if not.
4.15.8.12 Customization

Add an item named *item* to a custom menu and associate it with action *command*. *menu* should be 0, 1 or 2, for menu A (left button), B (middle button) or C (right button). If *item* is omitted, the menu item is constructed from *command*.

```c
Bool AddItemToXMenu(menu, name, type, cmd, argv, argc)
int menu;    /* menu index */
char * name; /* menu item name*/
int type;    /* type of item */
char * cmd;  /* simulator command or function name */
char ** argv; /* parameters for function call*/
int argc;    /* for function call*/
```

This routine adds an item to one of the three custom popups. It returns TRUE if successful, FALSE otherwise. *menu* ranges through (0,1,2). *name* is the name that appears on the popup menu. *type* is either 0, meaning that *cmd* is any string that you could type to the simulator prompt, or 1, meaning that *cmd* is the name of one of your functions that takes *argv* and *argc* as parameters. It isn’t clear just what the latter is useful for. When the item is clicked in the menu, either *gi..£ommand(cmd)* or *cmd(argc,argv)* is called. The widget does NOT copy the argv structure but uses it in place.

```c
Bool DeleteItemFromXMenu(menu, name)
int menu;    /* menu index */
char * name; /* menu item */
```

This routine deletes the item named *name* from the menu 0, 1 or 2. Returns TRUE if successful. If the item corresponds to a function with argv structure, this function deletes the argv structure before exiting.

```c
gi_process_button(button_buf, buttonx, buttony)
char * button_buf;
int buttonx, buttony;
```

Execute command *button_buf*, substituting *buttonx* for $x$, *buttony* for $y$, and the unit index of the unit on the current display window at (*buttonx*, *buttony*), if any, for $u$.

```c
void LogCustomXMenus()
```

This routine logs the custom menus in the log file.
int ReloadCustomXIcon(name, dir)
    char * name;  /* icon name */
    char * dir;  /* directory ending in /, or "" */

This routine loads or reloads a custom icon named name, found in directory dir. If dir is the zero length string, it looks for the icon files in . and ./icon. Returns 1 if successful, 0 otherwise.

4.15.6.13 Miscellaneous

gi_command("gi ye flag");

Toggle command history. If flag is 0, future commands will not be recorded in the command history panel. If it is 1, future commands will be recorded.

gi_command("gi i unitx unity screenx screeny");

Pop up an information window at (screenx,screeny) for unit at (unitx,unity).

gi_command(cmd)
    char *cmd;

Execute command cmd.

gi_show_prev(cmd)
    char *cmd;

Print the message cmd in the command history panel in the control window.

gi_log(string)
    char *string;

Print the string in the current log file.

gi_put_error(string)
    char *string;

Displays the string on the message panel in the control window, prefaced by "ERR:."
4.15. XGI: THE X11 INTERFACE

gi_put_message(string)
char *string;

Displays the string on the message panel in the control window.

gi_update_clock()

Update the clock value shown in the control window, if it has changed.

4.15.7 Integrating User X Code with XGI

The top level XGI widget is called gi_tool and is of class vPanedWidgetClass. This is the control panel widget. User code can add new widgets using gi_tool as parent, so that events are handled correctly. The popup windows are all of class transientShellWidgetClass, with gi_tool as parent. Users wishing to refer to gi_tool in their code should declare it:

extern Widget gi_tool;

Table 4.1 shows code to implement a scrollbar widget that can be used to control parameters used in network simulation (this code is in the file ~/example/xgi/scroll.c). Note that this code does not create a new variable which the scrollbar widget controls, but attaches the scrollbar widget to a pre-existing variable which is known to the simulator Item Table. Suppose you are running a simulation which has two floating point parameters DecayRate and InhibitRate. If these parameter were declared in the code which you linked at makesim time, and were NOT initialized, then they are loaded in the Item Table and so accessible at the user interface. Table 4.2 shows a command script that would compile and load in the scrollbar widget code, and then create scrollbar widgets to control the DecayRate and InhibitRate values.

The code shown in Table 4.1 uses the simulator tables to find the address of the variable to attach the slider control to. The same effect could be achieved without going through the simulator mapping tables by simply passing the address of the variable in the calls to XtAddCallback and XtScrollBarSetThumb. This is how the back-propagation slider controls are set up for the example described in section 4.15.5. The code which implements this is in file ~/example/xgi/BPcontrol.c.

4.15.8 Implementation

XGI evolved in three stages from GI. First, Michael McInerny created a version of XGI to run under the HP widget set using X11 release 2. This version never ran, due to bugs in the widget set. Later Nigel Goddard took the code and translated it to run under the Athena
```c
#include <X11/Intrinsic.h>
#include <X11/StringDefs.h>
#include <X11/Shell.h>
#include <X11/Scroll.h>
#include "sim.h"
extern Widget gi_tool;

static void JumpProc(sbar, param, percent)
    Widget sbar;  /* scrollbar widget */
    caddr_t param;  /* address of parameter (float *) */
    caddr_t percent;  /* percentage from top */
{
    register float * p = (float *) param;  /* address of parameter */
    *p = 1.0 - (* (float *) percent);  /* copy value */
}

MakeControl(argc, argv)
    int argc;  /* one argument, name of variable */
    char ** argv;  /* scrollbar is to control */
{
    Widget shell, scroll;
    NameDesc nte;
    MappingTable mte;
    static Arg scroll_args[] =
    {
        { XtNlength, (XtArgVal) 100},
        { XtNwidth, (XtArgVal) 20 },
    };
    if (FindName(argv[1],nte))  /* Name Table entry for variable */
    {
        IndexToItem(nte.index,&mte);  /* Mapping Table entry for variable*/
        shell = XtCreatePopupShell(argv[1], topLevelShellWidgetClass, gi_tool, NULL, 0);
        scroll = XtCreateManagedWidget("scroll", scrollbarWidgetClass, shell, scroll_args, XtNumber(scroll_args));
        XtAddCallback(scroll, XtNjumpProc, JumpProc, mte.item.floatval);
        XtPopUp(shell, XtGrabNone);  /* popup slider widget, set thumb */
        XtScrollBarSetThumb(scroll, 1.0-*(mte.item.floatval), 0.05);
    } else
        LOGfprintf(stderr, "MakeControl: \%s not found\n", argv[1]);
}
```

Table 4.1: Parameter Scrollbar Widget code
compile scroll
loadcode scroll
call MakeControl DecayRate
call MakeControl InhibitRate

Table 4.2: Creating a Scrollbar Control for a Variable

widget set using X11 release 3. At this point XGI looked like GI to the user. Finally, XGI was hacked to allow multiple display windows and custom popup menus. At no point was the original author of GI, Kenton Lynne, involved in the process, and for both of the authors of XGI this was their first piece of code using X11. In consequence of this history, the implementation of XGI is a mess.

The display windows are implemented in the files Display.c, Display.h and Display.P.h, and the file display_panel.c. In GI there were twenty or so global variables used to implement the display panel, e.g. for the current cursor position, current font, etc. XGI maintains a record of the values for each of these globals. Whenever a new display window is selected to be the current display, XGI copies the current global values into the current display record, and then copies the values from the record for the new display window into the globals. Logically this involves changing some items on the screen, e.g. the layout window and the mode panel in the control window. But changing screen real estate is expensive, so this is not done if the new display window has only been selected for a brief screen update. This occurs at the end of each simulation step, when XGI selects each display window in turn and updates it without changing the mode shown in the control window or the values shown in the layout window. The strategies and mechanisms for updating each display window are the same as those for GI.

The saving and retrieving of global values is done in functions in display_panel.c. Clearly this should have been done with pointers, but it would have required replacing every reference to a global.

The custom menu facilities are implemented in Display.c and its associated files. If you want to change the number of items in each menu, that is the place to look.

4.15.9 Known Bugs

1. Custom menus and the associated editing dialog are not guaranteed to stay exposed. Custom menus may not pop down if the cursor exits the menu window before it is fully realized. Re-entry and exit of the menu window will result in popdown.

2. Text input widgets (dialog boxes, control panel, etc.) do not resize when the text exceeds the current width/height of the widget.
3. Allowing input commands longer than 120 characters may cause custom menu editing to crash the simulator. (Suspect simulator, not XGI).

4. Resizing the display window has been known to cause X protocol errors on some machines because something tries to reconfigure a window to zero width.

4.15.10 Deficiencies

1. There is no mechanism to save and restore the current graphics layout.

2. There is no way to specify a custom icon from the control panel.
Chapter 5

Back Propagation Package

5.1 Introduction

A simple back propagation network consists of a number of layers of units. The first layer is the input layer, the final layer the output layer, and the remaining intervening layers are hidden layers. Links within the network are feedforward: no links within a layer or from a higher layer to a lower layer are allowed.

A back propagation network has two modes of operation. When operating in the forward mode, activation spreads from lower layers to higher layers, using the unit activation functions. When operating in the error-propagation mode, errors are propagated from higher layers to lower layers, using an error-propagation function, units adjusting the weights on their incoming links as the errors propagate. This package allows arbitrary activation and error-propagation functions. The standard UCSD sigmoid activation function and derivative based error-propagation functions (see [Rumelhart et al. 1986]) are provided in the package, but other functions may be written by the user.

Although many researchers have concentrated on looking at a single multi-layer backpropagation network, in general one would like to be able to include a back-propagation network (or several such networks) as sub-parts of a larger network. This package is designed to allow one or more back-propagation modules as part or all of a network.

A cycle is defined to be the process of feeding activation forward until the output layer has settled, then propagation errors back to the first hidden layer. The number of simulation steps this will take is twice the number of layers (only hidden and output layers, explained later) in the back-propagation module (the simulator must be running in synchronous mode).
5.2 Building a back-propagation module

The standard back propagation module has certain required features, and some optional features. The required features are the following: a control unit to monitor and control the running of the module, a bias unit that can be optionally linked to units in the module, and an output layer of units. In almost all cases there will be at least one hidden layer of units, but it is not necessary. The optional features are as follows: an input layer of units which feeds information to the first hidden layer, and a teach layer of units which always has the same number of units as the output layer. It feeds information into the output layer, in effect telling it what the correct output is. Finally there is an optional fire unit which is used to trigger the control unit to tell it to start the module.

A back-propagation module is built with a series of calls to library functions. First the module name is declared, then the hidden and output layers are specified. Finally the end of the module is indicated. Optionally input and teach layers may be declared, the input layer before any hidden layers, the teach layer after all the hidden layers. The functions to do this (in the order in which they should be called) are:

BPmodule declares the beginning of a module.

BPinput creates the optional input layer.

BPhidden creates a hidden layer.

BPoutput creates the output layer.

BPteach creates the teach layer.

BPlink creates a link between two units in the module.

BPfire creates the fire unit and links it to the $fire site of the control unit.

BPendmod indicates the end of the module specification.

Once the module has been constructed, the trial patterns and correct outputs must be presented to the network during simulation. If the module is part of a larger network, other units in the network may feed the trial and output patterns to the module. Alternatively, functions are provided to set the trial and output patterns in the input and teach layers from a user program. These functions are:

BPsetinput sets up a trial pattern in the input layer.

BPsetteach sets up the correct pattern in the teach layer.

BPcycle turns the module on for a number of cycles.
5.2. BUILDING A BACK-PROPAGATION MODULE

5.2.1 BPmodule

The first function that must be called is $BPmodule$. It takes as arguments a string which is the name of the module, and an integer which is the number of layers (including only the hidden and output layers, not input or teach layers). This will set up the building process and create a control unit and a bias unit. If the name of the module is $foo$ the control unit will be named "cont.$foo" and the bias unit "bias.$foo". For example:

$$BPmodule("learn", 3)$$

declares the beginning of a section of code that defines the module $learn$ that will have two hidden layers and one output layer. The units "cont.$learn" and "bias.$learn" will be created. From now on, until the module is completed with the $BPendmod$ routine, there should be no calls to the $MakeUnit$ unit construction function from user code.

5.2.2 BPinput

After the module is started with the $BPmodule$ command an optional input layer may be created. The routine $BPinput$ takes as an argument one integer representing the number of units in the layer. It creates a unit vector of that length with name "$foo(0)" (where $foo$ is the module name). The units are created with the null function. For example:

$$BPinput(7)$$

following the above call to $BPmodule$ would create a vector of seven units with the name "learn(0)" (not to be confused with "$learn[0]"). Thus the third input unit will be called "learn(0)[2]". The outputs of the units in the input layer can be set at any time during simulation using the $BPsetinput$ routine described in section 5.3.1. $BPinput$ returns the index to the vector.

5.2.3 BPhidden

Now the hidden layers are created, each layer being a vector of units. They will be given names "$foo(1)" to "$foo(n-1)" where $n$ is the number of layers specified in the call to $BP-module$ (if $n$ is 1 then there are no hidden layers). The units in these layers have one site called the $learn$ site. The routine $BPhidden$ must be called for each hidden layer. It takes 11 arguments, the first is an integer representing the number of units in the layer. The second is a pointer to a unit function, the standard function is supplied in the library and is called $UFh.o$ (it is the same as the standard output layer function). The third argument is a pointer to a site function for the $learn$ site, the standard function is $SFbpsigmoid$. This function computes a sigmoid curve, whose slope is determined by the global floating point variable $BPtemperature$, which has value 1.0 initially and can vary between 0 and 1.0.
The fourth argument is either 0 or 1 signifying if the bias unit is to be linked to the units in this layer. The fifth is the weight of the link from the bias unit; it must be supplied even if the bias unit will not be linked. The next six arguments specify what certain fields in each unit in the layer should be initially set to. They are all integers (in the floating point version only the last two are integers, the others are floats): initial potential, potential, data, output, initial state, and state. BPhidden returns the index to the vector. For example:

\[
\text{BPhidden}(3, \text{UFh}_0, \text{SFbpsigmoid}, 1, 500, 0, 0, 0, 0, 0, 0, 0)
\]

creates a layer with three units. The units’ unit function is \(\text{UFh}_0\) and their \$learn\ site function is \(\text{SFbpsigmoid}\). The bias unit will be linked to each unit in this layer with a weight of 500. The remaining field values for the units are initialized to 0.

### 5.2.4 BPoutput

After ALL the hidden layers are created the output layer must be created. The units in the output layer are identical to those of the hidden layers except that there is one extra site, the \$error\ site. As a result the routine BPoutput takes identical arguments except there is one more which appears right after the \$learn\ site function pointer, thus moving the remaining arguments one space to the right. This is the \$error\ site function pointer; the standard function given in the library is \(\text{SFerror}\) (see 5.5.1). BPoutput creates a unit vector with name “\text{foo}(n)\” where \(n\) is as above, and returns the index to the vector. For example:

\[
\text{BPoutput}(5, \text{UFh}_0, \text{SFbpsigmoid}, \text{SFerror}, 0, 0, 0, 0, 0, 0, 0, 0)
\]

creates a vector with five units. The units’ unit function is \(\text{UFh}_0\) and their \$learn\ site function is \(\text{SFbpsigmoid}\). The bias unit will not be linked to the units in this layer, and the field values for these units will be initialized to 0. If this call to BPoutput occurred after the above call to BPhidden then the layer would have the name “learn(3)”.

### 5.2.5 BPteach

After the output layer an optional teach layer can be created. The routine to do this is BPteach. It takes no arguments and creates a vector of the same length as the output layer with name “\text{foo}(n+1)\”. Links are made from each unit in the teach layer to the \$error\ site of its corresponding unit in the output layer. The units have no function (null function pointer) and their outputs can be set using the BPsetteach routine described later. BPteach returns the index of the vector.

So calling BPteach after the above call to BPoutput would create a vector with five units and name “learn(4)".
5.2. BUILDING A BACK-PROPAGATION MODULE

5.2.6 BPfire

After the output layer (and the teach layer if there is one) there can be an optional fire unit. The routine that creates this unit is BPfire; it takes no arguments. It creates a unit with name “firefoo” and links its output to the $fire site of the control unit. It returns the index of the fire unit.

So calling BPfire after the above call to BPteach or BPoutput would create a unit called “firelearn”. The unit would be linked to the $fire site of the unit “contlearn”. The unit function for the fire unit is automatically set to be Uffire, defined in the library.

Initially the output of the fire unit is 0. When its output is non-zero it triggers the control unit to start a cycle. Each time the fire unit runs it reduces its output by one. Therefore if a cycle takes 6 steps then running the network with the fire unit outputting 12 at the start would let the module run through 2 cycles.

The output of the fire unit can be set by the user with BPcycle described below. If there is no fire unit then the network will have to activate the control unit on its own.

5.2.7 BPlink

To link units in layers the routine BPlink is called; it takes 5 arguments. The first argument is the layer number of the unit where the link is coming from. It can be from 0 (input layer) to n-1 (last hidden layer). The second argument is the local unit index within the from layer which specifies the actual from unit. The third argument is the layer number of the unit that the link is going to. It can be from 1 (first hidden layer) to n (output layer). The fourth argument is the local unit index within the to layer which specifies the actual to unit. The fifth argument is an integer which is the weight of the link (float if floating point version). For example:

BPlink(0, 2, 1, 0, 250)

links the third unit of the input layer to the first unit of the first hidden layer, with a weight of 250.

Links can only be made in the forward direction, and no links can be made to the teach layer.

NOTE: Because the error-propagation function uses the link function pointer field as a backpointer with which to propagate errors there can not be any link function.

5.2.8 BPendmod

To complete the module the routine BPendmod is called with an argument which is the name of the module (given to BPmodule). After this is called only calls to the simulation functions BPsetteteach, BPsetinput or BPcycle (see Section 5.3) can be made (or BPmodule
to start a new module). For example, to complete the module started by the $BPmodule$
above we would call:

$$BPendmod("learn")$$

This would label the "learn" module as complete. An error will occur if the correct number
of layers have not been created.

5.3 Simulating the module

During simulation the trial patterns and correct output patterns must be provided to the
input and output layers. This can be done by other units if the module is embedded in a
larger network, or by explicit library function calls from user code.

5.3.1 BPsetinput

To set the output values of input units the routine $BPsetinput$ is called. It takes three
arguments. The first is a string which is the name of the module. The second is an integer
which is the local index of the unit within the input layer vector (starting at 0). The third
is an integer (or float in floating point version) which is the value to set the output of the
specified unit to. This routine can be called while the module is being built (after the input
layer has been built) or after the module has been completed. For example:

$$BPsetinput("learn", 6, 900)$$

called any time after the input layer has been created, will set the output of the seventh
unit in the input layer to 900. An error is signaled if the unit index is out of range.

5.3.2 BPsetteach

To set the output values of teach layer units the routine $BPsetteach$ is called. Its arguments
are identical to those of $BPsetinput$. It can be called any time after the teach layer is
created. For example:

$$BPsetteach("learn", 4, 0)$$

will set the output of the fifth unit of the teach layer to 0. An error is signaled if the unit
index is out of range.
5.3.3 BPcycle

BPcycle is called to activate the network for a number of cycles. This can also be done by feeding activation from another unit to the $fire site on the control unit (whenever the control unit gets activation on that site it starts a cycle). BPcycle takes an integer parameter, the number of cycles to be run, and a string, the name of the module to be activated. For example:

    BPcycle("concept-learn", 100)

will activate the module concept-learn for 100 cycles. Note that activating is not the same as simulation. In addition to the call to BPcycle, the simulator must be run for the appropriate number of time steps, i.e. at least \#cycles * \#steps-per-cycle.

5.4 Example: learning 8-3-8 encoding

The goal behind the 8-3-8 encoding problem is to teach a network to encode a number between 0 and 7 into a "3-bit" code, and then decode it back into the original number between 0 and 7. The original number is represented as the activation of one of a group of 8 units. Which number is represented depends on which unit is activated. In other words, there is a unit for 0, a unit for 1, etc., up to 7. The "3-bit" representation consists of three units. When the network learns the encoding there will be a unique pattern of activation over these three units when given the activation of each of the original 8 units. The decoded number is represented in the same way as the original number.

5.4.1 Constructing the network

We will create a module with one hidden layer, one output layer, an input layer, a teach layer, and a fire unit. The input layer will consist of the 8 units which represent the original number. The hidden layer will consist of the 3 units which represent the "3-bit" code. The output layer will have 8 units like the input layer to represent the decoded number. The teach layer will be used to feed the correct result to the output layer. The standard functions will be used for both the hidden and the output layers.

Links will be made from each of the units in the input layer to each of the units in the hidden layer. Then from each of the units in the hidden layer to each of the units in the output layer. First the initial definitions are given. Both sim.h and bp.h must be included.
#include "sim.h"
#ifdef FSIM
#define BP_ONE 1.0
#define BP_ZERO 0.0
#define CAST float
#else
#define BP_ONE 1000
#define BP_ZERO 0
#define CAST int
#endif

#include "bp.h"

The above code can be compiled for either the floating point or integer version of the simulator. When the floating point version is made it is compiled `-DFSIM` so `FSIM` will be defined and `BP_ONE` will be 1.0 (rather than 1000), and `BP_ZERO` will be 0.0 (rather than 0). Table 5.1 shows the top level code for building the 8-3-8 back propagation network.

```c
build(argc, argv)
int argc;
char *argv[];
{
    register int i, j;

    srand(random(getpid())); /* seed random number generator */
    AllocateUnits(40); /* upper limit on units */
    BPmodule(argv[1], 2); /* start 2 layer module */
    BPinput(8); /* create input layer with 8 units */
    BPhidden(3, UFh_o, SFbpsigmoid, 1, /* create hidden layer */
            BP_ONE/2, BP ZERO, BP ZERO, BP ZERO, BP ZERO, 0, 0);
    BPoutput(8, UFh_o, SFbpsigmoid, SFerror, 1, /* create output layer */
             BP_ONE/2, BP ZERO, BP ZERO, BP ZERO, BP ZERO, 0, 0);
    BPtrace(); /* create teach layer */
    BPtrace(); /* create fire unit */
    for(i = 0; i < 8; i++) /* create all links */
      for(j = 0; j < 3; j++)
        { BPlink(0, i, 1, j, (random())%700)*(BP_ONE/1000)+(BP_ONE/10));
            BPlink(1, j, 2, i, (random())%700)*(BP_ONE/1000)+(BP_ONE/10));
        } /* weights random .1 -.8 */
    BPEndmod(argv[1]);
}
```

Table 5.1: Back propagation build code

The call to `AllocateUnits` is standard for building any network. The number of units
allocated must include the units in the input layer (if there is one), in all hidden layers, in the output layer, and in the teach layer (if there is one). It also must include the bias unit, the control unit, and the fire unit (if there is one). The call to BPmodule names the module being built and defines it as having 2 layers (since one layer must be an output layer that means there is only one hidden layer). The call to BPinput creates the optional input layer with 8 units.

Then the hidden layer is created. It has 3 units, each unit having the function UFli.o, which is the standard unit function. The site function for these units is SFbpsigmoid, again the standard function. The “1” in the fourth argument position signifies that the bias unit is to be linked to each unit in this layer. The weight of the link is BP_ONE/2, which is 500 for the integer version, and .5 for the floating point version. The remaining 6 arguments set the initial potential, potential, data, output, initial state, and state values of each unit in this layer to 0. (The last two arguments are integers in either version of the simulator.)

The output layer is created with 8 units. It is created like the hidden layer except it has an extra site function, SFerror, for the $error site which hidden layer units do not have.

The call to BPteach sets up a teach layer with 8 units (because the output layer has 8 units), and links the output of each unit in the teach layer to the $error site of the respective units in the output layer.

Then linking is done, in the manner described above, between the input layer and the hidden layer, and then to the output layer. The weight for each link is a random number between 100 and 799 for the integer version, and between .1 and .799 for the floating point version.

The end of the module is signified by the call to BPendmod.

5.4.2 Running a simulation

To run a simulation, we shall use a function that sets up trial patterns and correct outputs in the input and teach layers. For example, unit 0 of the input layer is activated, as well as unit 0 of the teach layer; then the network is run through one cycle. The function below does this for all eight input and teach units. This function can be called from the simulator command interface. The code for this cycle function is shown in Table 5.2.

The first argument to the cycle function is the name of the module and the second is the number of times each of the eight units should be activated. Originally all the units in the input and teach layers are outputting 0. The function goes through each unit in those layers, one by one, and sets their outputs to 1000 (or 1) using BPsetinput and BPsetteach. The call to BPssetcyle sets up the module to be run for one cycle. Then the network is actually run, using the Step function. Then the units are reset to zero. This process is repeated as many times as the user specified in the function call.

The argument to Step is “4” because the number of steps in a cycle is 2 times the number of layers (hidden and output only). This example is contained in the example/838encode subdirectory.
cycle(argc, argv)
int argc;
char *argv[];
{
    register int i, j, k;

    if(argc != 3)
    {
        fprintf(stderr, "Wrong # args\n");
        return;
    }
    j = atoi(argv[2]);
    for(k = 0; k < j; k++)
        for(i = 0; i < 8; i++)
        {
            BPsetinput(argv[1], i, BP_OHE);
            BPsetteach(argv[1], i, BP_OBE);
            BPcycle(argv[1], 1);
            Step(4);
            /* 2 layers times 2 */
            BPsetinput(argv[1], i, BP_ZERO);
            BPsetteach(argv[1], i, BP_ZERO);
        }
}

Table 5.2: Code for running a back propagation simulation

5.5 Activation and error-propagation functions

There is a standard format for the unit functions to be used in the hidden and output layers. It is of the form:

In forward state (activation):
    Output some function of activation from $learn site (usually identity);
    call BPendfwd passing the unit pointer as argument.
In reverse state (error-propagation):
    For each link into $learn site:
        Change weight of link.
        Propagate error to unit at other end of link.
    call BPendrev passing the unit pointer as argument.

The library provides the standard UCSD functions for activation and error-propagation. Additionally, the standard unit activation function has a floating point value BPtemperature which divides the weighted sum activation before exponentiation. Also the back propagation
of errors function has an additional floating point value $BPlearn$ which multiplies the normal error propagation signal before it is propagated. Both these variables are user-settable, and both should be in the range (0-1). Initially both have the value 1.0, which gives the same effect as the version 4.0 library.

Of course, other activation and error-propagation functions may be written by the user.

5.5.1 Writing error propagation functions

Errors are propagated through links between units by the link function pointer. When the links are made with $BPlink$ the link function pointer is set to point to the data field of the $learn$ site where the link is originating from. Suppose unit $A$ is linked to unit $B$ via link $I$. When $B$ propagates its error down to $A$ it puts the error where $I$'s link function pointer is pointing; this will be to the data field of $A$'s $learn$ site. Any user-written error-propagation functions should use this method. As an example consider the code shown in Table 5.3 for the standard error-propagation function in the library.

First note that the symbol $FLINT$ is defined to be float if compiled for the floating point version, and int if compiled for the integer version of the simulator. In addition, remember that $BP\_ONE$ is defined as 1.0 for the floating point version and 1000 for the integer version.

In the first part of the error-propagation section $delta$, the error to be propagated is computed. It is a function of the error passed down from an upper layer, stored in $up\_sites->data$, and the current output of the unit, $up->output$. The specific function used is described in [Rumelhart et al. 1986].

For each link coming into the $learn$ site of the unit the weight change for each link, $deltaw$, is computed. It is a function of $delta$ and the output of the unit where the link originates, together with a learning rate $BPlearn$. In addition a momentum factor can be added in. The global variables $BPmomentum$ and $BPlearn$ are floating points between 0 and 1. Initially $BPlearn$ is set to 1 (maximum learn rate) and $BPmomentum$ to 0.5. $lp->data$ is set to the last weight change of the link (initially 0).

After the link weight change is computed the change is made to the current weight, and then it is stored in $lp->data$ to be used to compute the momentum factor during the next error-propagation stage.

Finally the error $delta$ is passed down to the unit at the other end of the link as a function of the weight of the link. In other words the propagated error is in proportion to the significance of the link. Note that the address $lp->link_f$ must be type-cast as a float pointer or an int pointer (depending on the version of the simulator) so that the compiler will not interpret it as a function pointer. Also note that the propagated error is added; this is because a single unit might output to a number of units, and error should be totaled from all these units.

The library function $BPendfwd$ switches the state of the unit from forward to reverse and sets the NO\_UNIT\_FUNC\_FLAG. The library function $BPendrev$ switches the state of the
Chapter 5. Back Propagation Package

```c
FLINT
UFh_o(up)
Unit *up;
{
    FLINT delta, nudelta, deltaw;
    Link *lp;

    /**-------------------**
    ** activation code **
    **-------------------**/
    if(TestFlagP(up, BP_FORWARD_FLAG))
    {
        up->potential = up->output = up->sites->value;
        BPendfwd(up);
    } else
    /**------------------------**
    ** error-propagation code **
    **------------------------**/
    {
        delta = ((up->sites->data) * (up->output) * 
                   (BP_ONE - up->output))/(BP.ONE * BP.ONE);
        nudelta = delta * BPlearn;
        for(lp = up->sites->inputs; lp != NULL; lp = lp->next)
        {
            deltaw = ((nudelta * *(lp->value))/BP.ONE) 
                       + BPmomentum * lp->data;
            lp->weight += deltaw;
            lp->data = deltaw;
            *(FLINT *)(lp->link._f) += (delta * lp->weight)/BP.ONE;
        }
        BPendrev(up);
    }
}
```

Table 5.3: Code for back propagation error function
unit from reverse to forward, sets the NO_UNIT_FUNC_FLAG, and clears the accumulated error.

Errors originate at the $error sites of output units. The $error site function calculates the error of the output unit by comparing its output with data coming from the teach layer. It puts the error in the data field of the $learn site of the output unit, thus allowing for standard error-propagation procedures. The library's $error site function is SFerror.

5.6 Module unit layout and operation

Back-propagation module units appear in the UnitList in a particular order. The control unit comes first, then the bias unit, followed by the units in the input layer (if there is one). Next come the units of all of the hidden layers, the output layer, and the teach layer (if there is one). The last unit is the fire unit (if there is one).

If the name of the module is learn then the control unit would be named contJearn, the bias unit bias.learn, and the fire unit fire.Jearn. Each layer would be named learn(n), where n specifies the layer number of the particular layer. The input layer number is 0, hidden layers start at 1. The indices of these units may be found using the NameToInd function. Each layer is a vector, while the control, bias and fire units are scalars.

Initially all layers units have the NO_UNIT_FUNC_FLAG set, as well as the BP_FORWARD_FLAG. To start simulation the control unit turns on the first hidden layer of units by unseting the NO_UNIT_FUNC_FLAG. After these units run they reset their NO_UNIT_FUNC_FLAG's. In addition, if the BP_FORWARD_FLAG is set it is then unset so that the next time the unit is turned on it will be in the reverse direction (if the flag was unset then it is set again). Then the control unit turns on the next layer of units. The process continues until the output layer has run. At this point the output layer is run again (this time in the reverse, error-propagating direction) and the control unit then turns on layers in the reverse order. When it reaches the first hidden layer the whole process is ready to start again.

NOTE: The unit flag BP_FORWARD_FLAG is defined as 12. Care should be taken that there is no conflict with flags defined in other packages.
Bibliography
